Alvis
Handbuch für Entwickler

Jan Bauerdick, Colin Benner, Eduard Boos, Dominik Dingel,
Simon Groth, Sebastian Schmitz, Frank Weiler

1. September 2011

Inhaltsverzeichnis

Wir wünschen viel Spaß bei der Weiterentwicklung von Alvis.

1 Benötigtes Equipment

1.1 Software

1.1.1 Eclipse

Alvis als RCP Projekt kann am besten mit Eclipse weiterentwickelt werden, wir haben mit mindestens der Version 3.6 gearbeitet.

Es besteht die Möglichkeit bei eclipse.org das Paket „Eclipse for RCP and RAP Developers" herunterzuladen, dies ist ein komplett vorkonfiguriertes und mit allen Abhängigkeiten bestücktes Eclipse.

Falls ein bereits vorhandenes Eclipse für die Entwicklung von Alvis benutzt werden soll, kann man die benötigten Plug-ins auch über die Update Funktion von Eclipse beziehen.

1.1.2 Subversion

Für die Versionskontrolle und zur zentralen Ablage benutzen wir ein zur Verfügung gestelltes SVN. Entsprechendes Plugins und Programme sind für Eclipse und jedes Betriebssystem verfügbar.

1.1.3 ANTLR

Zum generieren von großen Teilen des Compilers benutzen wir den Compiler-Generator ANTLR 3 (`http://www.antlr.org`). Auch hierzu gibt es Eclipse-Plug-ins.

1.2 Kenntnisse

Für das (Weiter-)Entwickeln der Plug-ins für Alvis sind Kenntnisse im Umgang mit RCP erforderlich. Diese sollte man sich aneignen, bevor man mit der Entwicklung von einem Plug-in beginnt. Dabei waren folgende Quellen für uns hilfreich.

- Vogella.de – Sehr zu empfehlen für Einsteiger.
  Lars Vogels Tutorial für Eclipse RCP bietet einen guten Einstieg in die Welt von RCP. Anfänger sollten hier beginnen.
  `http://www.vogella.de/articles/EclipseRCP/article.html`

- RalfEbert.de – RCP Buch für fortgeschrittene.
  Ralf Ebert hat ein Buch über RCP in Form einer Webseite veröffentlicht. In diesem Buch werden in jedem Kapitel andere Konzepte von RCP erklärt.
  http://www.ralfebert.de/eclipse_rcp/

- Einstieg in Eclipse 3.5 – Künneth, Thomas
  Wer lieber etwas in der Hand hat, der ist mit diesem Buch gut bedient. Hier wird Eclipse RCP gut erläutert.
  Verfügbar in der Uni-Bibliothek, ISBN: 978-3-8362-1428-5

- The Definitive ANTLR Reference: Building Domain-Specific Languages – Terence Parr
  In diesem Buch ist praktisch alles erklärt, was man zur Benutzung von ANTLR 3 braucht. ISBN: 978-0-9787-3925-6.

2 Alvis

2.1 Übersicht über die Komponenten

Bei den einzelnen Komponenten wird die Bezeichnung „Plug-in" vielfältig eingesetzt. Wir sprechen von einem Plug-in für Alvis, wenn wir eine neue Algorithmen Art hinzufügen, oder das Hauptplug-in um andere Funktionalitäten erweitern. Dabei ist Alvis selbst auch nur ein Plug-in.

Alle Plug-ins befinden sich im SVN – Ordner „trunk/alvis".

2.1.1 Das Alvis Plug-in

„de.unisiegen.informatik.bs.alvis" – Das Haupt Plug-in, welches die graphische Oberfläche beinhaltet.

2.1.1.1 Extension Points

- „de.unisiegen.informatik.bs.alvis.extensions.runpreferences"
- „de.unisiegen.informatik.bs.alvis.runvisualizer"
- „de.unisiegen.informatik.bs.alvis.export"
- „de.unisiegen.informatik.bs.alvis.extensionpoints.datatypelist"
  An diesem Extension Point registrieren Plug-ins, die Datentypen bereitstellen, welche Datentypen sie bereitstellen und in welchen Paketen diese enthalten sind.
- „de.unisiegen.informatik.bs.alvis.extensionpoints.fileextension"
  An diesem Extension Point registrieren sich Plug-Ins, die eine Algorithmen-Art bereitstellen, dies wird durch ihre Dateiendung gekennzeichnet. Dazu muss das Interface de.unisiegen.informatik.bs.alvis.extensionpoints.IFileExtension im jeweiligen Plug-in implementiert werden.

2.1.2 Plug-ins für Alvis

Diese Plug-ins erweitern das Haupt Plug-in um notwendige Funktionen. Die Teilung des Hauptprogramms in verschiedene Plug-ins fördert die Modularisierung und die Wiederverwendbarkeit, ggf. auch für neue Projekte.
Dabei können die Plugins in optionale Plugins und nicht optionale Plugins unterschieden werden. Optionale Plugins sind nicht zwingend zum Ausführen von Alvis erforderlich, während nicht optionale Plugins Kernbestandteile von Alvis sind.

Die Plug-ins befinden sich im Ordner „trunk/alvis/plugins".

nicht optionale Plug-ins:

- „de.unisiegen.informatik.bs.alvis.primitive" – Primitive.
- „de.unisiegen.informatik.bs.alvis.primitive.datatypes" – Primitive Datentypen.
- „de.unisiegen.informatik.bs.alvis.ui.navigator" – Navigator (Package Explorer).
- „de.unisiegen.informatik.bs.alvis.vm" – Virtuelle Maschine.

Optionale Plug-ins:

- „de.unisiegen.informatik.bs.alvis.vm.test" – Tests der virtuellen Maschine.
- „de.unisiegen.informatik.bs.alvis.editors.xml" – XML Editor.
- „de.unisiegen.informatik.bs.alvis.help"
- „de.unisiegen.informatik.bs.alvis.graph.help"
- „de.unisiegen.informatik.bs.alvis.sync.help"

2.1.3 Graph Plug-in

Das Graph Plug-in erweitert Alvis um die Datentypen Graph, Knoten und Kante.

- „de.unisiegen.informatik.bs.alvis.graph" – Graph Plug-in.
- „de.unisiegen.informatik.bs.alvis.graph.datatypes" – Graph Datentypen.

2.1.4 Betriebssysteme Plug-in

2.1.5 Virtual Machine Plug-in

Die Virtual Machine steuert den Ablauf eines Algorithmus. Sie ist für die Verwaltung der einzelnen Schritte der Threads verantwortlich.

2.1.5.1 Klassen

Im Folgenden werden die einzelnen Klassen kurz erläutert, alle Einzelheiten sind in der jeweiligen Java-Doc zu finden.

- AbstractAlgo
  Definiert die zu implementierende Schnittstelle der Algorithmen für Verwaltungsfunktionen wie: Starten, Stoppen, Inspizieren, Breakpoints
- Activator
- Algo
  Ist dabei ein Beispiel Objekt das AbstractAlgo implementiert, aber ohne eine besondere Funktionalität.
- AlgoThread
  Verwaltet die Algorithmen und dazugehörigen Threads als eine Einheit, die Virtuelle Maschine arbeitet nur mit AlgoThreads.
- AlvisFileObject
  Nimmt den zu kompilierenden Algorithmus in seiner CharSequence auf und wird an den Compiler, der von ToolProvider bereitgestellt wird, überreicht.
- BPListener
  Die Breakpoint Schnittstelle, beim Erreichen eines Breakpoints wird der entsprechende Listener als eigener Thread aufgerufen.
- DPListener
  Die Decision Point Schnittstelle, beim Erreichen eines DecisionPoints wird der entsprechende Listener aufgerufen um die Situation zu klären.
- DynaCode
  Ist für das Kompilieren und Laden des Java-Algorithmus verantwortlich. Dazu erzeugt sie aus den benötigten Paketen den Classpath und erzeugt eine Liste von Verzeichnissen, in denen Algorithmen liegen. Danach wird der Compiler in Javac mit den erzeugten Parametern angestoßen. Nach dem Kompilieren holt sich DynaCode das class-Objekt und lädt es.
- Javac
  Kompiliert Java-Dateien, die ihm von DynaCode gegeben werden. Dazu baut die Klasse nur ein Array mit den Optionen für den Compiler auf und übergibt sie ihm. Zuerst wird versucht, von ToolProvider einen Compiler zu beziehen. Im Erfolgsfall wird der Algorithmus in einem AlvisFileObject gespeichert und kompiliert. Falls kein Compiler zurückgegeben wird, wird der Compiler aus com.sun.tools.javac.Main benutzt. Dieser Fallback ist nicht schön und soll noch aus Alvis verschwinden. Leider trat das Problem erst gegen Ende der Projektgruppe auf. Da es einen funktionierenden Workaround gibt, wurde der Fehler

als weniger wichtig erachtet und andere Dinge vorgezogen.
Wenn sich tools.jar nicht im Classpath befindet, stoppt Alvis beim Aufruf der Funktion, die etwas aus com.sun.tools.javac.Main benutzt.

- VirtualMachine
  Das Schnittstellen Objekt zur GUI, diese arbeitet nur auf der VirtualMachine.

2.1.6 Die Help-Plug-ins

Die Help-Plug-ins sind dazu da, dem User Hilfestellung zu leisten. Man erreicht sie in Alvis über Hilfe->Hilfeinhalte.
Der Aufbau gestaltet sich wie folgt:
Plug-ins müssen den Extension Point org.eclipse.help.toc implementieren und dort eine Help Table of Contens bereitstellen. Der Pfad zu dieser TOC muss nl/NameDerTOC.xml lauten, damit die Hilfe in mehreren Sprachen zur Verfügung gestellt werden kann. Die deutschen Hilfeseiten – und somit auch die deutsche Table of Contents – befinden sich alle im Ordner nl/de, die standardmäßig angezeigte, englische direkt im Plug-in. Konvention ist, für HTML-Seiten einen Unterordner html anzulegen, um die Übersicht zu bewahren.

2.1.7 Der Pseudocode-nach-Java-Compiler

Der Compiler besteht aus fünf verschiedenen ANTLR-Grammatiken (zu finden in „alvis/compiler/src/main/antlr3/de/uni_siegen/informatik/bs/alvic”):

- TLexer.g
  Enthält, wie der Name schon vermuten lässt, die Definition des Lexers. Zusätzlich zu dem üblichen Inhalt einer Lexer-Grammatik sind noch einige Funktionen enthalten, die für Tests und Autovervollständigung Zugriff auf Tokens bieten.

- TParser.g
  Enthält die Definition des Parser. Dieser erzeugt aus dem Eingabe-Token-Strom einen abstrakten Syntaxbaum (AST), der anschließend an FunctionPass weitergereicht wird.

- FunctionPass.g
  Durchläuft den von TParser erzeugten AST und erstellt eine Liste aller Funktionen und ihrer Typen. Diese Funktionalität separat vom eigentlichen TypeChecker zu haben ermöglicht es uns, Funktionen in beliebiger Reihenfolge zuzulassen.

- TypeChecker.g
  Hier wird überprüft, ob das Programm wohltypisiert ist um böse Überraschungen beim Compilieren des später erzeugten Java-Quelltextes oder gar zur Laufzeit zu verhindern. An einigen Stellen werden zusätzlich noch Typen an Knoten im AST geschrieben, die dann vom Cod-Generator verwendet werden.

- CodeGenerator.g
  Erzeugt mit Hilfe von ANTLRs StringTemplate-Bibliothek je nach Auswahl der Templates LaTeX-Code aus dem Eingabecode, oder den Java-Code, der später compiliert und ausgeführt werden kann. Die Template-Definitionen befinden sich in den Dateien „Java.stg" (fur die Erzeugung von Java-Code) bzw. „LaTeX.stg" (für die Erzeugung von schön formattiertem LaTeX-Code).

Hinzu kommen einige Java-Klassen in
„alvis/compiler/src/main/java/de/uni_siegen/informatik/bs/alvic":

- AbstractTLexer
  Enthält Hilfsmethoden für TLexer.g, unter anderem solche, die dem Lexer mitteilen, hinter welchen Zeichenketten sich Typen verbergen. Die restlichen Methoden sind im Wesentlichen wie die in TLexer.g für Tests und Autovervollständigung gedacht.

- AbstractTParser
  Enthält eine Methode um ANTLRs interne Tokennummern in Tokennamen zu übersetzen und um für einen gegebenen Tokentyp anzugeben, welche Typen als nächstes kommen könnten (für die Autovervollständigung).

- AbstractTreeParser
  Oberklasse von AbstractTypeChecker und FunctionPass, enhält gemeinsame Hilfsmethoden.

- AbstractTypeChecker
  Oberklasse von TypeChecker einhaltet viele Methoden, die von TypeChecker.g benutzt werden. Die meisten dienen dazu Typen zu überprüfen und gegebenenfalls entdeckte Fehler zu melden.

- TypedTree, TypedTreeAdaptor, TypedErrorNode
  Abgeleitet von den entsprechenden Common*-Klassen. Diese Klassen werden benötigt um die Typen von Ausdrücken vom TypeChecker an den CodeGenerator weiterzureichen.

- Compiler
  Die Hauptklasse des Compilers und die Verbindung zur Außenwelt. Wichtig für die Verwendung von außen sind die drei `compile`-Methoden, die tatsächlich übersetzen wenn keine Fehler auftreten, sowie die `check`-Methode, die ausschließlich überprüft, aber nie Code erzeugt. Ebenfalls wichtig ist die `getExceptions`-Methode, die eine Liste aller während des Compilierens erzeugten Exceptions liefert.

- Type, SimpleType, FunctionType, ArrayType
  Klassen, die vom TypeChecker benutzt werden, um die Typen von Ausdrücken, Attributen und Methoden zu beschreiben. Type ist die abstrakte Oberklasse der drei anderen. ArrayType beschreibt Arrays, FunctionType Funktionen/Methoden und SimpleType Typen mit Vererbung, die etwa Typen von Attributen und lokalen

Variablen sind. SimpleType wird auch verwendet, um Container-Typen mit Typ-Parameter für den Elementtyp zu beschreiben, etwa `Vertex Queue`. In Java-Syntax wäre das etwa `Queue<Vertex>`.

SimpleType enthält eine große Menge des Codes, der verwendet wird, um aus Objekten eines Typs eine Beschreibung dieses Typs zu erstellen. Dazu werden mittels Reflection Typen von Attributen und Methoden ausgelesen.

- Member
  Hilfsklasse für die Verwaltung der Typen von Attributen und Methoden. Jedes Attribut und jede Methode eines SimpleTypes wird durch ein Member-Objekt beschrieben. Ein solches Objekt speichert Namen und Typ eines Attributs bzw. einer Methode. Methoden unterscheiden sich von Attributen dadurch, dass bei ihnen als Typ ein FunctionType auftritt.

- *Exception
  TypeException ist die Oberklasse der anderen Exceptions. TypeException ist abgeleitet von ANTLRs RecognitionException um im ANTLR-generierten Compiler all diese Exceptions verwenden zu können. InvalidAssignmentException und InvalidStatementException beschreiben Syntaxfehler, werden also von TParser geworfen, die restlichen Exceptions beschreiben Typfehler und werden daher von FunctionPass und TypeChecker geworfen.

Weiter Informationen zu diesen Klassen sind als JavaDoc verfügbar.

2.1.7.1 Einschränkungen

- Der Compiler kann nicht mit Attributen umgehen, die als einen generischen Typ, d.h. einen Typ-Parameter haben, wie etwa `Stack<String>`. Trifft der Compiler auf ein solches Attribut, so behandelt er es genauso, wie er `Stack` behandeln würde. Insbesondere würde der Compiler davon ausgehen, dass `Stack.pop()` den Rückgabetyp `Object` hat.

- An dieser Stelle kommt die zweite Einschränkung ins Spiel: Alvis unterstützt keinerlei Typecasts. wenn also `pop()` ein `Object` zurückliefert, so kann man dies nur dann wieder als etwas anderes, als ein `Object` verwenden, wenn es in einer in einem Plug-in definierten Methode explizit auf einen anderen Typ gecastet wird.

  Dies ist keine echte Einschränkung, wie etwa Haskell und ML zeigen, die ebenfalls ohne Typecasts auskommen. Es ergibt sich dadurch auch ein Vorteil: es kann kaum passieren, dass im Algorithmus eine ClassCastException auftritt.

- Es gibt weder ein `break`- noch ein `continue`-Konstrukt. Dies nachträglich einzubauen erfordert zwar alle Grammatiken des Compilers zu bearbeiten und vermutich die Template-Definitionen zu erweitern, dürfte aber nicht schwer sein.

In TLexer.g muss man etwa `break` aus den Alternativen in JAVAKEYWORD entfernen und ein eigenes Token daraus machen. Im Parser wird man sicherstellen müssen, dass `break` nur im Körper von Schleifen verwendet werden darf. Dazu empfiehlt sich der Einsatz eines „Gated Sematic Predicate". Die beiden Typechecker-Phasen müssen die neuen Tokens nur weiterleiten, mit Typen haben ja weder `break` noch `continue` etwas zu tun. Im Codegenerator muss dann aus den Tokens per Template schlicht der richtige Java-Code (und wenn man den Quelltext nachher noch nach LaTeXexportieren können möchte auch der richtige LaTeX-Code) erzeugt werden.

- Um die Sprache einfach zu halten haben wir uns auf die nötigsten Operatoren beschränkt. So gibt es keine kombinierten Zuweisungsoperatoren wie $+=$ oder $*=$. Weiterhin gibt es keine bitweisen Operatoren wie etwa Shift-Operatoren, bitweises Und, bitweises Oder, bitweises Nicht oder XOR.

  Das ließe sich aber ebenfalls mit verhältnismäßig geringem Aufwand nachrüsten. Möchte man die Funktionalität der Operatoren nutzen, diese aber nicht in die Sprache einbauen, so kann man natürich ohne weiteres Methoden dafür benutzen. Diese Methoden würde man sowieso benötigen, wenn man die Operatoren, wie die bereits eingebauten, überladbar implementieren wollte.

3 Export

3.1 Erste Schritte

Wir werden den Export des Programms und der Plug-ins über eine Update Seite und über Features bewältigen. Dazu empfehlen wir das Tutorial von Ralf Ebert zur Update Funktionalität in RCP.

http://www.ralfebert.de/blog/eclipsercp/p2_updates_tutorial_36/

Neben den Projekten aus dem vorherigen Kapitel benötigen wir nun auch die Feature-Projekte aus dem Ordner „trunk/export".

3.2 Export des Hauptprogramms für verschiedene Plattformen

Nach Änderungen, Erweiterungen des Hauptprogramms oder der Plugins wollen wir diese jetzt an die Benutzer verteilen. Falls Alvis noch nicht installiert wurde reicht es Alvis neu zu exportieren und auf der Webseite zum Download anzubieten.

Benutzer die Alvis schon installiert haben sollen über die Updatesite eine Möglichkeit bekommen Alvis zu aktualisieren, im folgenden gehen wir auf die einzelnen Schritte dafür ein.

3.2.1 Vorbereitung

Aus dem Tutorial wissen wir, dass unser Export auf Basis von Features läuft. Diese Features müssen wir nun erstmal aus dem SVN in den Workspace auschecken, in dem auch unser Hauptprogramm und die benötigten Plug-ins liegen.

Die Features befinden sich im Ordner „trunk/alvis/export". Wir benötigen:

- „org.eclipselabs.p2.rcpupdate"
- „org.eclipselabs.p2.rcpupdate.utils"
- „de.unisiegen.informatik.bs.alvis.app"

Das Hauptprojekt wird in einen Ordner exportiert. Dabei wird auch ein Ordner „repository” erstellt. Wir erstellen zuerst einen Ordner, welchen wir für den Export benutzen möchten. Hier „D:/Alvis/”. Anschließend noch einen Ordner „repository” für die Update Seite. Hier „D:/Alvis/repository”.

Damit die Änderungen auch als Updates auf die Webseite kommen, brauchen wir die vorhandene Update Seite. Diese befindet sich im Ordner „trunk/alvis/updates” und wird in regelmäßigen Abständen automatisch auf den Server kopiert.

Dazu checken wir den Ordner „updates” aus. Wir wählen „D:/Alvis/repository” als Speicherort.

3.2.2 Export mit einer Target Platform

Um eine RCP Anwendung für verschiedene Plattformen zu exportieren muss die sogenannte „Target Platform” angepasst werden. Mit einer Target Platform kann man festlegen, welche Plug-ins der Anwendung zur Verfügung stehen. Hat man keine spezielle Target Platform gewählt, so bedient sich die Anwendungen aus den Plug-ins, die der installierten Eclipse Distribution zur Verfügung stehen.

Durch den Wechsel der Target Platfrom lassen sich Eclipse Plugins die für andere Betriebssysteme benötigt werden auswählen. So kann mit Eclipse Alvis Cross Plattform gebaut werden.

3.2.3 Einstellen der Target Platform

Wir wollen nun die vorhandene Target Platform als Target Platform für unser Projekt einstellen. Dazu müssen wir das Projekt
„de.unisiegen.informatik.bs.alvis.app” in unserem Workspace haben. Wir öffnen die Datei „allPlatforms.target” und warten bis alle Plug-ins geladen wurden. Dies kann unter Umständen einige Zeit in Anspruch nehmen.

Das vorläufige Ergebnis sieht folgendermaßen aus.

Mit einem Klick auf „Set as Target Platform” lässt sich diese Konfiguration einstellen. Danach wird der Workspace neu gebaut. Die Projekte sollten keine Fehler aufweisen. Sollten Fehler bezüglich nicht aufzufindender Imports sein ist dies ein Indiz für fehlende Plug-ins in der Target Platform. Über den Button „Edit” lassen sich benötigte Plug-ins aus der Update Seite für Helios holen.

Zum Zeitpunkt des Verfassens wurde Alvis unter Eclipse 3.6 entwickelt. Sollte Alvis inzwischen mit einer neueren Version von Eclipse entwickelt werden, so ist es möglicherweise notwendig auch die Target Platform aufzufrischen. Eine genauere

Anleitung findet man im RCP-Buch von Ralf Ebert im Kapitel 16 – Export und Verteilung von RCP-Anwendungen[1]. Noch genauer geht Ebert in seinem Blog[2] auf die Problematik ein. Es muss, in dem Fenster wo wir uns die Plug-ins aus der Update Seite holen, unbedingt ein Haken bei „Include all environments" gesetzt sein.

Nachdem die Target Platform übernommen wurde und wir die Korrektheit des Programms überprüft haben können wir das Projekt exportieren.

3.2.4 Export

Wir möchten nochmals darauf hinweisen, dass es nötig ist die Versionsnummern zu erhöhen, damit ein Update auf die jetzt exportierte Software funktioniert. Es müssen sowohl in der
„MANIFEST.MF" und der
„*.product", im Hauptprogramm, und in der
„feature.xml" im Feature „ .alvis.app",
die Versionsnummer auf einen gemeinsamen neuen Wert erhöht werden.

Um das Hauptprogramm zu exportieren öffnen wir die Datei „build.product" aus dem Plug-in „de.unisiegen.informatik.bs.alvis".

Wir klicken auf den Link „Eclipse Product export wizard" und es erscheint der Export Wizard.

Wir wählen als „Root directory:" einen beliebigen, noch nicht im Ordner „D:/Alvis/" vorhandenen Ordnernamen und stellen „D:/Alvis/" als „Directory:" ein. Wichtig ist, dass der Haken bei „Generate metadata repository" gesetzt ist. Außerdem setzen wir einen Haken bei „Export for multiple platforms".

Wir bestätigen die obigen Einstellungen und klicken auf „Next".

Im nächsten Fenster wählen wir die Plattformen aus, die wir bedienen möchten. Nachdem wir auf „Finish" geklickt haben und der Export abgeschlossen ist, können wir im Zielverzeichnis die Ordner für die einzelnen Plattformen finden.

Beim Export wurden im Ordner „repository" die Java Archive generiert, die eine ältere Version von Alvis bei einem Klick auf „Help" → „Check for Updates" für ein Update

[1] http://www.ralfebert.de/eclipse_rcp/distribution\/
[2] http://www.ralfebert.de/blog/eclipsercp/rcp_builds\/

auf unsere neue Version benötigt. Als Standard Update Pfad ist für Alvis die „Alvis Update Seite[3]" eingetragen also muss der Inhalt des Ordners „repository" in diesen Ordner kopiert werden.

Dies geschieht über einen „commit" der Update Seite.

[3] http://alvis.bs.informatik.uni-siegen.de/update/