

Entwurfsdokument

Simon Bischof, Jan Haag, Adrian Herrmann, Lin Jin, Tobias Schlumberger, Matthias Schnetz

Praxis der Softwareentwicklung Projekt 3:

Automatisches Prüfen der Korrektheit von Programmen
Gruppe 1



WS 2011/2012

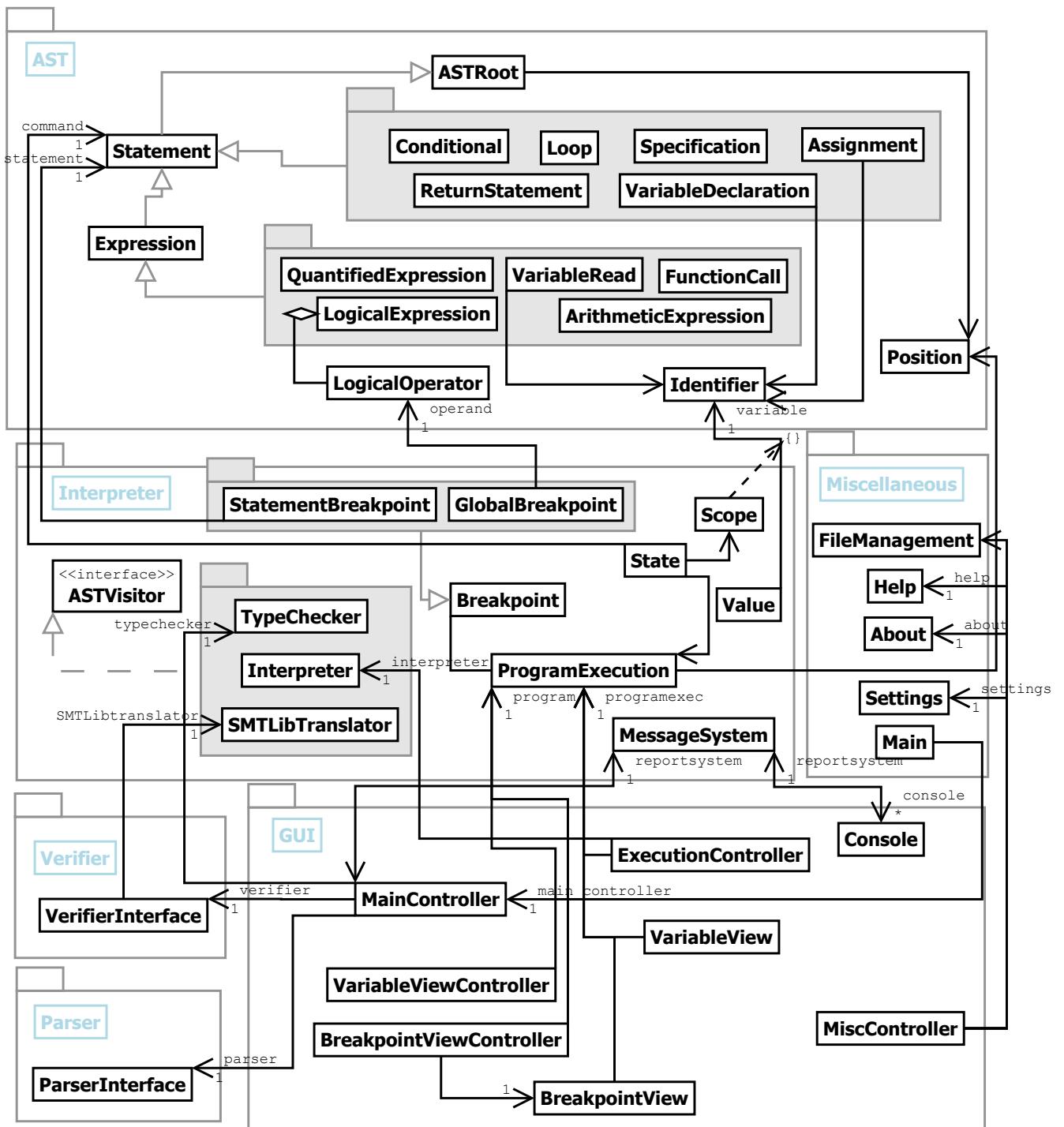
Inhaltsverzeichnis

1 Klassendiagramme	3
1.1 Übersicht	3
1.2 Feinstruktur der Komponenten	4
1.2.1 AST	4
1.2.2 Parser	8
1.2.3 Interpreter	8
1.2.4 Beweiser	11
1.2.5 Misc	12
1.2.6 Benutzeroberfläche	12
2 Aktivitätsdiagramme	15
2.1 Parser/Type-Checker	15
2.2 Z3-Anbindung	16
3 Zustandsdiagramm	17
4 Sequenzdiagramme	18
4.1 Besucher-Muster	18
4.2 Breakpoints	19
4.3 Interpreter-Initialisierung	20
4.4 Start-Button	21
4.5 Single-Step-Button	22
5 Erweiterbarkeit	23
6 Struktur der While-Sprache	24
7 Syntax der While-Sprache	25
8 Implementierungsplan	28
8.1 Vorgangsliste	28
8.2 Gantt-Diagramm	30

1 Klassendiagramme

1.1 Übersicht

Das folgende Klassendiagramm zeigt die Interaktionen zwischen den Komponenten des Systems.



Die gesamte Architektur basiert auf dem Entwurfsmuster Model-View-Controller. Dies ermöglicht uns einen flexiblen Programmumentwurf, der eine spätere Änderung oder Erweiterung erleichtert und eine Wiederverwendbarkeit der einzelnen Komponenten ermöglicht.

- Das **Modell** enthält die darzustellenden Daten, z.B die Zustände der Variablen während der Programm ausführung und die vom Benutzer gesetzten Breakpoints. Aber auch die Komponenten AST und Miscellaneous sind Bestandteile des Modells. Diese Speichern jeweils den Syntaxbaum des Benutzerprogramms und wichtige Daten der Benutzeroberfläche.
- Die **Sicht** stellt die Daten, die sie vom Modell bekommt, auf der Benutzeroberfläche dar und nimmt Benutzerinteraktionen entgegen. Dabei wird sie unterteilt in verschiedenen Views, wie zum Beispiel die VariableView und BreakpointView.
- Um die Zusammenarbeit der ersten zwei Komponenten kümmert sich die **Steuerung**. Da wir mehrere Views haben, gibt es dementsprechend auch mehrere Controller. Diese steuern alle Komponenten des Systems und geben Daten weiter. Wichtige Aufgaben der Steuerung sind beispielsweise das Aufrufen des Parsers und des Interpreters.

1.2 Feinstruktur der Komponenten

1.2.1 AST

Für die Struktur des AST verwenden wir das Composite-Muster. Damit wird die Teil-Ganzes-Hierarchie der While-Grammatik repräsentiert, indem Objekte zu Baumstrukturen zusammengefügt werden. Außerdem ermöglicht uns das Muster, Objekte und Kompositionen einheitlich zu behandeln. Um die Methoden des Interpreters, des Type-Checkers und des Beweisers zu kapseln, verwenden wir hier auch das Visitor-Muster. Somit können diese Methoden leicht für alle konkreten Klassen implementiert werden und auch einfacher modifiziert werden.

- **ASTRoot**

Die Klasse ASTRoot bildet die Wurzel des Baumes. Sie wird direkt vom Parser erzeugt und wird zum Verifizieren an den Beweiser übergeben.

- Attribut:

- `position`
eine bestimmte Position im Quelltext

- Methoden:

- `getPosition()`
gibt `position` zurück
`acceptVisitor(visitor: ASTVisitor)`
nimmt einen konkreten Visitor entgegen

- **Program**

Die Klasse Program ist eine Ansammlung von Funktionen.

- Attribute:

- `functions`
Programm besteht aus einer oder mehreren Funktionen
`mainFunction`
Programm muss aus genau einer Main-Funktion bestehen
`axioms`
Program kann auch Axiome enthalten

- Methoden:

- `getFunctions()`
gibt die Menge der Funktionen zurück
`getMainFunction()`
gibt `mainFunction` zurück
`getGlobalAxioms()`
gibt die Menge der globalen Axiomen zurück

- **Function**

Die Klasse **Function** stellt eine Funktion im Programmcode dar.

- Attribute:

- name**
der Name der Funktion
 - parameters**
beliebig viele Parameter
 - returnType**
Rückgabetyp der Funktion
 - functionBody**
eine Funktion besteht einem **StatementBlock**

- Methoden:

- getFunctionBlock()**
gibt den Körper der Funktion als **StatementBlock** zurück
 - getParameters()**
gibt die Menge der Parameter zurück
 - getName()**
gibt **name** zurück
 - getReturnType()**
gibt **returnType** zurück

- **FunctionParameter**

- Attribute:

- identifier**
der Name des Parameters **type**
 - Parameter** muss einen Typ haben

- Methoden:

- getName()**
gibt **identifier** zurück
 - getType**
gibt **type** zurück

- **StatementBlock**

Die Klasse **StatementBlock** kapselt eine logische zusammengehörende Liste von **Statements**.

- Attribute:

- statements**
eine geordnete Liste von **Statements**

- Methoden:

- getNextStatement()**
gibt das nächste Statement zurück, oder NULL, falls das Ende des Blocks erreicht wurde.

- **Statement**

Die Klasse **Statement** steht für eine Anweisung im Programmcode.

- **Conditional**

Die Klasse **Conditional** ist eine Unterklasse von **Statement** und steht für eine bedingte Anweisung.

- Attribute:

- trueStatements**
es gibt einen **StatementBlock**, der bei Erfüllung der Bedingung ausgeführt wird
 - falseStatements**
es gibt einen **StatementBlock**, der bei Nichterfüllung der Bedingung ausgeführt wird
 - condition**
Bedingung, die überprüft wird

- Methoden:
 - `getCondition()`
gibt `condition` zurück
 - `getTrueConditionBody()`
gibt `trueStatements` zurück
 - `getFalseConditionBody()`
gibt `falseStatements` zurück
- **Loop**
 Die Klasse `Loop` ist eine Unterklasse von `Statement` und steht für eine while-Anweisung.
 - Attribute:
 - `body`
Schleifenkörper, der bedingt ausgeführt wird, solange `condition` wahr ist
 - `condition`
Bedingung, die überprüft wird
 - `invariant`
für die Schleife ist es möglich, eine oder mehrere Schleifeninvarianten anzugeben
- **Specification**
 - Attribut:
 - `expression`
Ausdruck, der überprüft wird
- **ReturnStatement**
 Die Klasse `ReturnStatement` stellt eine return-Anweisung dar.
 - Attribut:
 - `returnValue`
Ausdruck, dessen Wert zurückgegeben wird
 - Methode:
 - `getReturnValue()`
gibt `returnValue` zurück
- **Assignment**
 Die Klasse `Assignment` steht für eine Zuweisung von Variablen.
 - Attribute:
 - `value`
der Wert der zugewiesen werden soll
 - `identifier`
Variable, der `value` zugewiesen wird
 - Methoden:
 - `getValue()`
gibt `value` zurück
 - `getVariable()`
gibt `identifier` zurück
- **VariableDeclaration**
 - Attribute:
 - `name`
Name der Variable
 - `value`
bei der Deklaration muss genau ein Wert eines Ausdrucks zugewiesen werden
 - `type`
Typ der Variable

- Methoden:
 - `getName()`
 - gibt `name` zurück
 - `getValue()`
 - gibt `value` zurück
 - `getType()`
 - gibt `type` zurück
- **Expression**
 Die Klasse `Expression` steht für einen Ausdruck im Programmcode.
- **LogicalExpression**
 Die Klasse `LogicalExpression` steht für einen unären oder binären Ausdruck, der einen Booleanwert zurückgibt.
 - Attribute:
 - `subExpression`
 - ein logischer Ausdruck kann als Operanden wieder Ausdrücken haben
 - `quantifier`
 - ein logischer Ausdruck kann einen forall- oder exists-Quantor haben
 - `logicalOperator`
 - ein logischer Ausdruck muss einen Operator aus `{!, |, &, >, >=, <, <=, !=, ==}` besitzen
- **QuantifiedExpression**
 Die Klasse `QuantifiedExpression` steht für einen Ausdruck, der einen Quantifier enthält.
- **ArithmeticExpression**
 Die Klasse `ArithmeticExpression` steht für einen unären oder binären Ausdruck, der einen Zahlenwert zurückgibt.
 - Attribute:
 - `subExpression`
 - ein arithmetischer Ausdruck kann als Operanden wieder Ausdrücken haben
 - `arithmeticOperator`
 - ein arithmetischer Ausdruck muss einen Operator aus `{+, -, *, /, %}` besitzen
- **FunctionCall**
 Die Klasse `FunctionCall` modelliert einen Methodenaufruf.
 - Attribute:
 - `function`
 - Funktion, die aufgerufen wird
 - `parameters`
 - Menge von Ausdrücken, die als Parameter der Methode übergeben werden
- **VariableRead**
 Die Klasse `VariableRead` stellt einen Zugriff auf eine Variable dar.
 - Attribut:
 - `identifier`
 - Variable, die gelesen wird
 - Methode:
 - `getVariable()`
 - gibt `identifier` zurück
- **Identifier**
 Die Klasse `Identifier` steht für eine Variable.
 - Attribut:
 - `name`
 - Name der Variable

- Methode:
`getName()`
gibt `name` zurück
- **Position**
Die Klasse **Position** stellt eine Position im Programmcode dar.
 - Attribute:
`line`
Zeile im Code
`column`
Spalte im Code

1.2.2 Parser

Der Parser liefert die AST-Struktur für den Interpreter und Beweiser. Einige Klassen dieser Komponente werden vom Parsergenerator ANTLR erstellt.

- **ParserInterface**
Die Klasse **ParserInterface** dient als Schnittstelle zwischen der Komponente Parser und dem restlichen System. Sie wird vom Main-Controller der GUI initialisiert.
 - Attribute:
`parser`
Parser für den Programmcode
`lexer`
Lexer für den Programmcode
 - Methode:
`parse(text: String)`
Initialisiert Parser und Lexer, über gibt ihnen den Quelltext
- **Parser**
Die Klasse **Parser** ist für die Erzeugung des AST zuständig. Es wird eine **ParseException** geworfen, falls kein AST erzeugt werden kann.
- **Lexer**
Die Klasse **Lexer** zerlegt den Quelltext in Tokens.
 - Attribut:
`source`
der vom Benutzer eingegebene Quelltext
 - Methoden:
`Lexer(source:String)`
Konstruktor der Klasse. Bei der Instanzierung wird der Quelltext übergeben.
`getSource()`
gibt den Quelltext zurück

1.2.3 Interpreter

Für die Integration des Interpreters verwenden wir das Visitor-Muster. Es kapselt die Operationen, die auf Elementen des AST ausgeführt werden. Das Muster ermöglicht uns, Operationen zu verändern oder neue Funktionen hinzufügen, ohne die Elementklassen zu modifizieren. Im obigen Klassendiagramm ist auch ein Teil des Modells dargestellt. Diese speichern wichtige Daten bezüglich der Programmausführung.

- **ASTVisitor**
Das Interface **ASTVisitor** deklariert für jede Klasse konkreter Elemente eine Besuchsfunktion.

- **TypeChecker**

Die Klasse **TypeChecker** ist ein konkreter Besucher und implementiert die Aufgabe, die Korrektheit der Typen zu überprüfen. Wenn ein Typfehler gefunden wird, so wird man durch eine `IllegalTypeException` benachrichtigt.

- Methode:

`checkTypes()`

startet den Vorgang des Type-Checks

- **Interpreter**

Die Klasse **Interpreter** ist ein konkreter Besucher und implementiert die Aufgaben des Interpretierens. Ist die Bedingung einer Assert-Anweisung nicht erfüllt, so wird eine `AssertFailureException` geworfen.

- Methode:

`step(state:State)`

schrittweise Ausführung des Programms. Dabei wird der Funktion der aktuellen Zustand `state` übergeben und diese gibt wieder eine Instanz der Klasse `State` zurück.

- **SMTLibTranslator**

Die Klasse **SMTLibTranslator** ist ein konkreter Besucher und implementiert die Aufgabe, den AST in die von Z3 benötigte Sprache zu übersetzen.

- Methode:

`getWPTree(ast: ASTRoot)`

übersetzt den AST in ein `WPProgram`

- **State**

Die Klasse **State** speichert den aktuellen Zustand der Programmausführung, wie z.B. Werte von Variablen.

- Attribute:

`statement`

jedem Zustand ist ein Statement zugeordnet

`currentScope`

die erste Scope-Instanz, die instanziert wurde

- Methoden:

`createScope()`

erzeugt für eine Variable die entsprechende Scope-Instanz

`destroyScope()`

zerstört eine Scope-Instanz

`setVar(name: String,value:String)`

setzt eine Variable und ihren Wert

`createVar(name: String,value:String,type: TypeOfValue)`

erstellt eine Variable mit einem Typ und Wert

- **Scope**

Die Klasse **Scope** bildet eine Hierarchie der Sichtbarkeit von Variablen des Programms. Je weiter unten sich eine Scope-Instanz befindet, desto lokaler ist die damit verbundenen Variable. In der obersten Ebene befinden sich also die globalen Variablen. Bei einem Variablenzugriff wird zunächst die unterste Ebene überprüft, ob diese die gesuchte Variable enthält. Wenn dies nicht der Fall ist, so sucht man in der nächst höheren Ebene weiter. Wenn man bereits die oberste Ebene erfolglos durchsucht hat, ist die Variable nicht vorhanden und eine Exception wird geworfen.

- Attribut:

`upScope`

nur die erste Scope-Instanz wird von `State`, alle weiteren werden von der Klasse selbst verwaltet

- Methode:

`getParent()`

gibt die „Eltern“-Instanz `upScope` zurück

```

getNextStatement()
gibt das nächste Statement zurück
addVars(<Identifier, Value>*)
fügt im untersten Scope alle Variablen hinzu, in den darüberliegenden Scopes nur noch die nicht vorhandenen

```

- **Value**

Die Klasse **Value** speichert den Wert einer zugehörigen Variable.

- Attribut:

```

type
Typ des Wertes
identifier
Name der Variable

```

- Methode:

```

getType()
gibt type zurück

```

- **Breakpoint**

- Attribut:

```

active
boolsche Variable, die anzeigt, ob der Breakpoint aktiviert ist

```

- Methoden:

```

setActive(active:boolean)
aktiviert oder deaktiviert den Breakpoint
isActive()
gibt active zurück

```

- **StatementBreakpoint**

Die Klasse **StatementBreakpoint** ist eine Unterklasse von **Breakpoint** und stellt einen Breakpoint dar, der an einer bestimmten Stelle im Quellcode steht.

- Attribut:

```

statement
Anweisung, an der der Breakpoint gebunden ist

```

- Methode:

```

getStatement()
gibt statement zurück

```

- **GlobalBreakpoint**

Die Klasse **GlobalBreakpoint** ist eine Unterklasse von **Breakpoint** und stellt einen globalen Breakpoint dar, der an einem Ausdruck gebunden ist und nicht an einer bestimmten Stelle im Programmcode.

- Attribut:

```

expression
logischer Ausdruck, an dem der globale Breakpoint gebunden ist

```

- Methode:

```

getExpression()
gibt expression zurück

```

- **ProgramExecution**

Die Klasse **ProgramExecution** modelliert die Programmausführung.

- Attribut:

```

breakpoints

```

`ProgramExecution` ist für die Überprüfung der Breakpoints zuständig
`position`
`ProgramExecution` hat zu einem Zeitpunkt eine bestimmte Position im Quellcode

- Methoden:
`ProgramExecution(interpreter: Interpreter)`
Konstruktor, der Interpreter wird dabei übergeben
`checkBreakpoint()`
überprüft, ob ein Breakpoint getroffen wurde
`setBreakpoint(active: Boolean, pos: Position)`
setzt einen aktiven oder inaktiven Breakpoint an die Position pos
`getVariables()`
gibt alle Variablen und ihren Werten zurück

- **MessageSystem**

Die Klasse `MessageSystem` leitet Rückmeldungen (z.B. Fehlermeldungen) an die GUI weiter.

- Attribut:
`consoles`
Das Message-System hat bestimmte Konsolen gespeichert, denen es Nachrichten übermitteln soll
- Methoden:
`addConsole(console: Console)`
fügt console zur Liste der zu benachrichtigenden Konsolen hinzu
`removeConsole(console: Console)`
entfernt console aus der Liste
`sendUpdate()`
benachrichtigt Konsolen über Veränderungen

1.2.4 Beweiser

Die Komponente Verifier übersetzt den AST in eine Struktur, die die Weakest-Precondition generiert. ANTLR hilft uns auch hier, die benötigten Lexer- und Parserklassen zu erzeugen.

- **VerifierInterface**

Die Klasse `VerifierInterface` stellt die Schnittstelle zwischen der Komponente Verifier und dem restlichen System dar. Sie wird vom Main-Controller initialisiert und startet die Übersetzung des SMTLib-Translators.

- Attribute:
`smtlibTranslator`
Übersetzer, der den AST übersetzen soll
`program`
Programm in der wp-Struktur
`z3lexer`
Lexer für Z3-Ergebnis
`z3parser`
Parser für Z3-Ergebnis
- Methoden:
`notifyConsole()`
benachrichtigt Konsolen über Beweisergebnisse
`verify(ast: ASTRoot)`
startet die Übersetzung

- **S-Expression**

Die Klasse `S-Expression` stellt einen beliebigen Ausdruck dar.

- Attribute:
 - `op`
 - Operator des Ausdrucks
 - `subexpression`
 - Operanden sind wieder S-Expressions
- Methode:
 - `toString()`
 - gibt eine Stringrepräsentation zurück als Rückübersetzung
- **Constant**
 Die Klasse **Constant** ist eine Unterklasse von **S-Expressions** und bildet das Blatt des Composite-Musters.
 - Attribut:
 - `stringRepräsentation`
 - die Konstante als String

1.2.5 Misc

In dieser Komponente werden Daten, die der Benutzeroberfläche betreffen, gespeichert.

- **FileManagement**
 Die Klasse **FileManagement** kümmert sich um das Speichern von Benutzerdateien.
 - Methoden:
 - `loadFile(dir: String)`
 - lädt eine Datei
 - `saveFile(text: String, dir: String)`
 - speichert Programm in eine Datei
- **About**
 Die Klasse **About** speichert nützliche Informationen zu unserem Produkt.
 - Attribut:
 - `text`
 - wichtiger Text
 - `singleton`
 - Singleton-Muster, da der Text nicht mehrmals erzeugt werden muss
 - Methoden:
 - `getInstance()`
 - gibt `singleton` zurück
 - `getText()`
 - gibt `text` zurück
- **Settings**
 Die Klasse **Settings** speichert Einstellungen für den Beweiser.
 - Attribute:
 - `timeout`
 - Zeitlimit für einen Beweisvorgang
 - `memorylimit`
 - Speicherlimit für einen Beweisvorgang

1.2.6 Benutzeroberfläche

Die Benutzeroberfläche bietet dem Benutzer eine leicht zu bedienende Schnittstelle und nimmt dessen Eingaben entgegen. Um die verschiedenen Sichten zu steuern, gibt es für jede Sicht einen eigenen Controller. Über diese Controllers geschehen alle Interaktionen mit anderen Komponenten, z.B. reichen sie Benutzereingaben

weiter, damit diese verarbeitet werden können. Die Controllers stellen der Views auch die von ihr benötigten Informationen zur Verfügung.

- **MainController**

Die Klasse **MainController** ist zuständig für die Initialisierung des Parsers, des TypeCheckers, des Beweisers und des Message-Systems zuständig. Sie erzeugt außerdem auch noch die Main-Frame.

- Attribute:

- parser**

- der initialisierte Parser

- typechecker**

- der initialisierte TypeChecker

- verifier**

- der initialisierte Beweiser

- reportsystem**

- das initialisierte MessageSystem

- execController**

- der MainController muss den ExecutionController kennen

- Methoden:

- initMainFrame()**

- erzeugt die MainFrame

- initVerifier()**

- erzeugt den Beweiser

- initParser()**

- erzeugt den Parser

- **ExecutionController**

Die Klasse **ExecutionController** ist für die Initialisierung der Objekte zuständig, die an der Programmausführung beteiligt sind, also Interpreter und ProgramExecution.

- Attribute:

- state**

- gibt den Zustand des Interpreters an, entweder idle, paused oder running **interpreter**

- der initialisierte Interpreter

- programexec**

- die initialisierte Programmausführung

- Methoden:

- startInterpreter()**

- startet den Interpreter

- stopInterpreter()**

- stoppt den Interpreter

- singleStep()**

- geht in den Single-Step-Zustand

- pauseInterpreter()**

- pausiert den Interpreter

- actionPerformed(e: ActionEvent)**

- beobachtet Aktionen des Benutzers und führt das entsprechende Event aus

- **VariableViewController**

Die Klasse **VariableViewController** ist zuständig für die Steuerung der VariableView, die den Zustand der Variablen anzeigt.

- Attribut:

- varview**

- erzeugte View für die Variablen

- Methode :

- actionPerformed(e: ActionEvent)**

- beobachtet Aktionen des Benutzers und führt das entsprechende Event aus

- **BreakpointViewController**

Die Klasse **BreakpointViewController** ist zuständig für die Steuerung der BreakpointView, die den Zustand der Breakpoints anzeigt.

- Attribut:

brview

erzeugte View für die Breakpoints

- Methode :

itemStateChanged(e: ItemEvent)

beobachtet Aktionen des Benutzers und führt das entsprechende Event aus

- **EditorController**

Die Klasse **EditorController** ist zuständig für die Steuerung des Editors, der den Quellcode mit Syntax-highlighting anzeigt und das Ändern des Codes ermöglicht.

- Attribut:

editor

erzeugter Editor

keywords

Keywörter der While-Sprache für Syntax-Highlighting

memento

gespeicherte Aktionen des Benutzers am Quellcode, für die Undo- und Redo-Funktionen

- Methoden:

keyPressed(e: KeyEvent)

keyTyped(e: KeyEvent)

keyReleased(e: KeyEvent)

beobachtet Aktionen des Benutzers und führt das entsprechende Event aus

- **MiscController**

Die Klasse **MiscController** ist zuständig für die Initialisierung der Misc-Klassen.

- Attribute:

filemanagement

help

about

settings

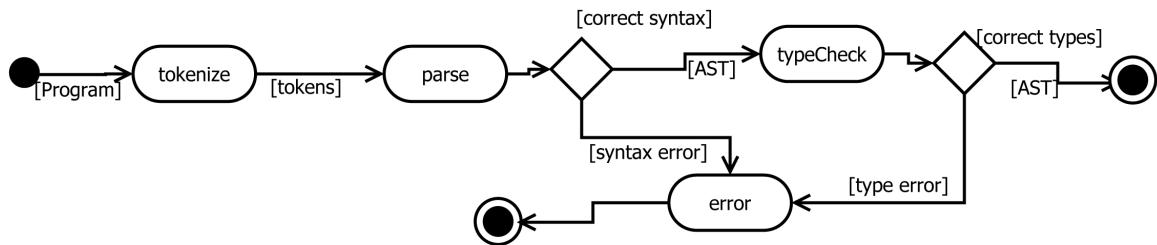
- Methode:

actionPerformed(e: ActionEvent)

beobachtet Aktionen des Benutzers und führt das entsprechende Event aus

2 Aktivitätsdiagramme

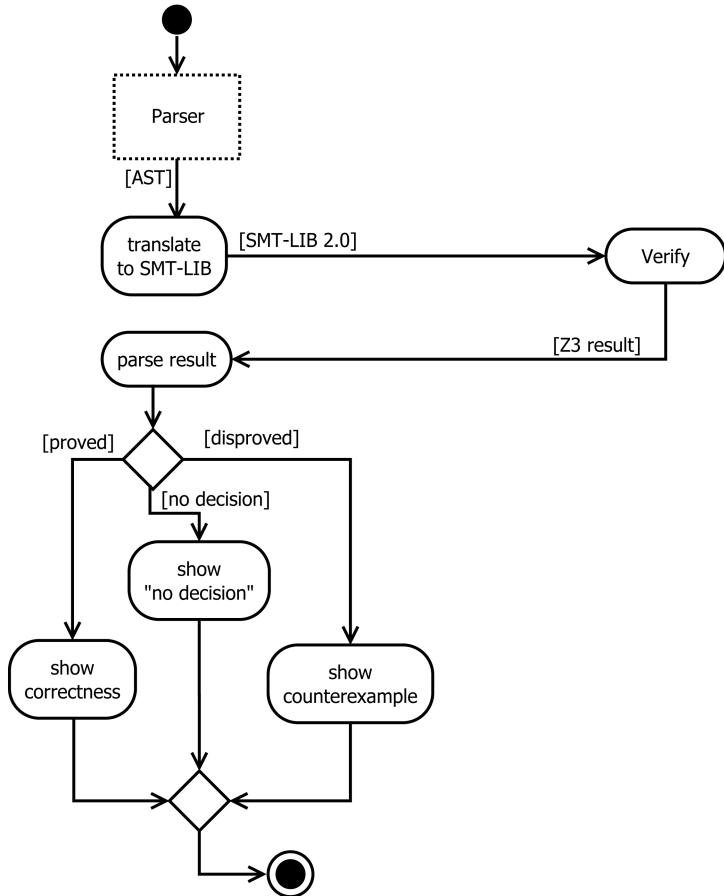
2.1 Parser/Type-Checker



Beim Aufruf des Interpreters wird das Programm in mehreren Schritten geparsst.

1. Der Programmtext wird als einzelner String dem Tokenizer übergeben.
2. Der Tokenizer trennt den String an den wichtigen Stellen und gibt ein Array von Tokens zurück.
3. Der Parser generiert bei syntaktisch korrekten Programmen daraus einen abstrakten Syntaxbaum (AST).
4. Bei Syntaxfehlern bricht der Parser mit einem Fehler ab.
5. Im Erfolgsfall überprüft der Typechecker die Korrektheit der Typen: Sind die Typen korrekt, gibt dieser den vom Parser generierten AST zurück, sonst beendet er sich mit einem Fehler.

2.2 Z3-Anbindung

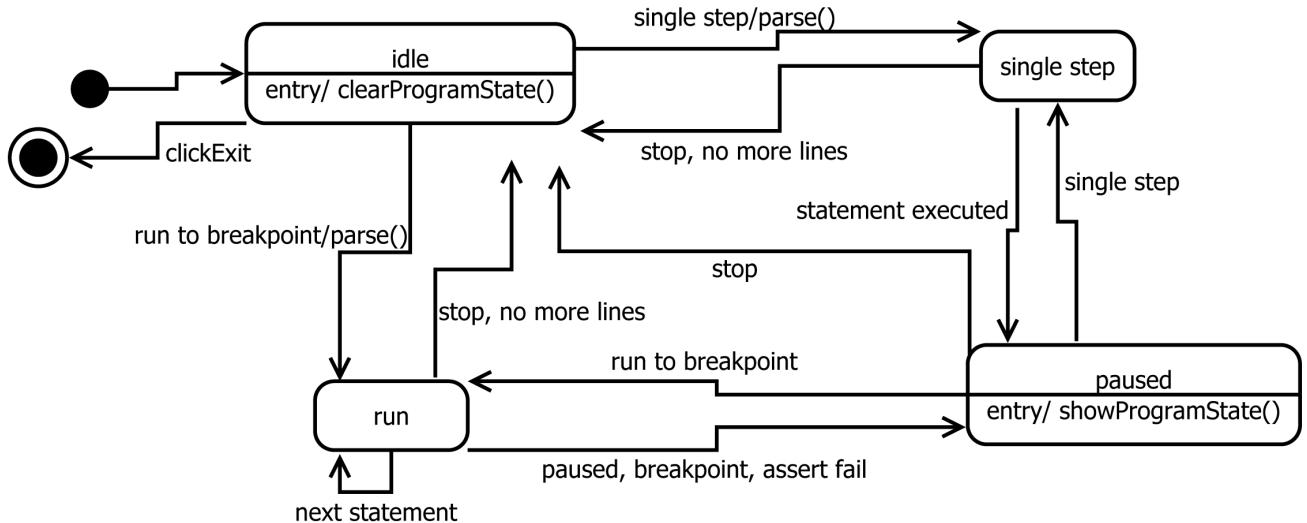


Zur Überprüfung der Korrektheit des Programms wird Z3 benutzt.

1. Zuerst wird das Programm geparsert (siehe Aktivitätsdiagramm Parser/Type-Checker).
2. Im Fehlerfall ist keine Überprüfung durch Z3 möglich. Im Erfolgsfall wird der durch den Parser generierte AST an den SMTLib-Translator gegeben.
3. Der SMTLib-Translator übersetzt das Programm inklusive Spezifikation ins SMTLib-2.0-Format. Dieses bildet die Eingabe für Z3.
4. Die von Z3 zurückgegebene Antwort wird vom Result-Parser analysiert.
5. Meldet der Beweiser die Korrektheit des Programms oder konnte er keine Entscheidung treffen, wird dieses Ergebnis dem Benutzer bekannt gegeben. Falls der Beweiser das Programm falsifizieren konnte, wird dem Benutzer das Ergebnis zusammen mit einem möglichen Gegenbeispiel angezeigt.

3 Zustandsdiagramm

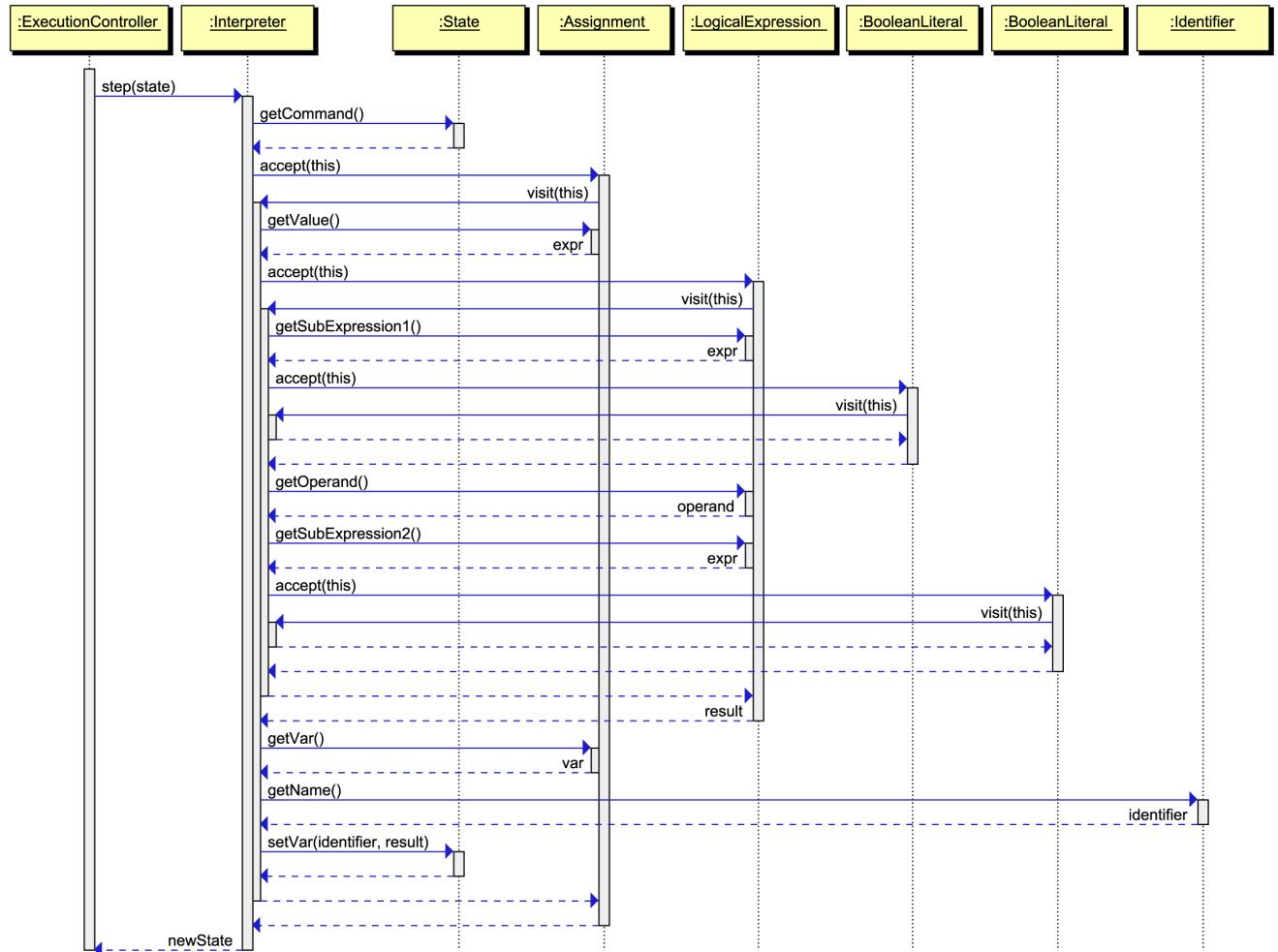
Das folgende Zustandsdiagramm zeigt das Verhalten des Systems bei Benutzeraktionen.



- Beim Starten des Programms geht dieses in den „idle“-Zustand. Hier läuft der Interpreter nicht, und es ist kein Programmzustand gespeichert. Falls einer vorhanden ist, wird dieser beim Eintritt in den „idle“-Zustand gelöscht.
- Beim Auswählen von „single step“ wird das Userprogramm geparsert und ein Statement wird ausgeführt, nachdem der Zustand „single step“ betreten worden ist. Ist kein Statement mehr vorhanden, so beendet sich der Interpreter, das Programm geht zurück in den Zustand „idle“. Sonst wird nach Ausführen des Statements das Programm pausiert, der Zustand „paused“ wird eingenommen.
- Beim Eintritt in den „paused“-Zustand wird der Zustand des Userprogramms ausgegeben. Während der Pausierung läuft der Interpreter nicht. In diesem Zustand stehen die gleichen Möglichkeiten wie im „idle“-Zustand zu Verfügung, das Parsen bei Verlassen des Zustands entfällt aber.
- Wenn im „idle“-Zustand „run to breakpoint“ aufgerufen wird, wird das Userprogramm geparsert und das Programm geht in den Zustand „run“. Das Userprogramm wird solange ausgeführt, bis es zu Ende ist (neuer Zustand: „idle“) oder der Interpreter pausiert, ein Breakpoint getroffen oder eine Assertion falsifiziert wird. In diesen Fällen ist der neue Zustand „paused“.
- In jedem Zustand außer „idle“ ist es zusätzlich möglich, das Userprogramm abzubrechen, wobei der Interpreter beendet wird und alle vorhandenen Variablen-Informationen gelöscht werden. Das Programm geht danach in den Zustand „idle“.
- In jedem Zustand kann das Programm durch einen Klick auf den Exit-Button beendet werden.

4 Sequenzdiagramme

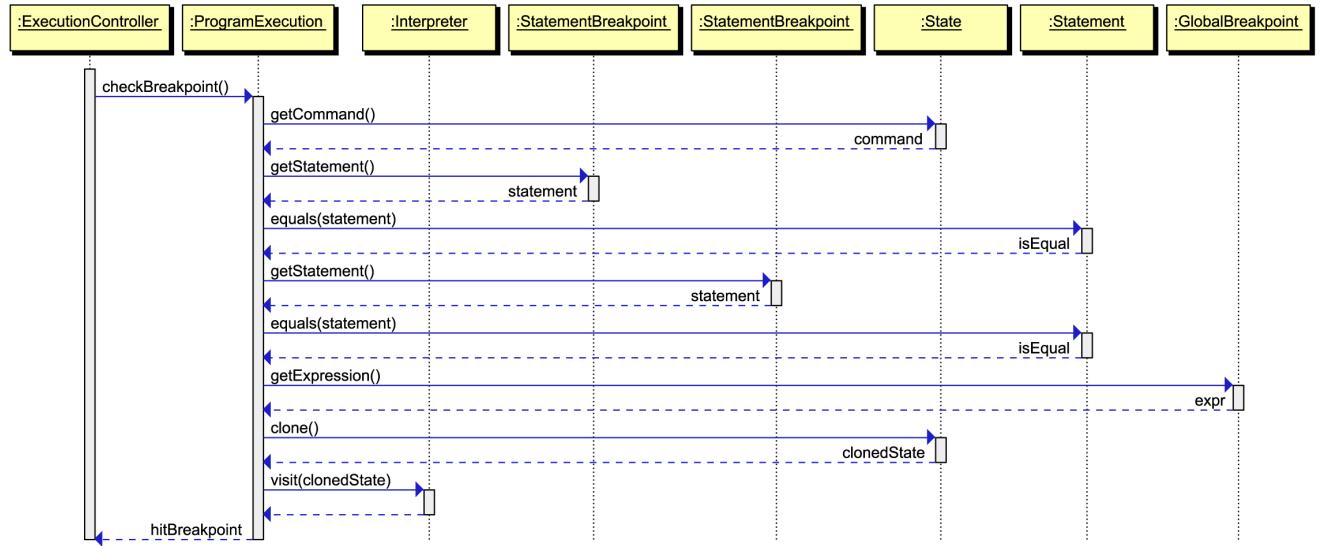
4.1 Besucher-Muster



Dieses Sequenzdiagramm zeigt die Arbeitsweise des Besuchermusters an einem Beispiel. Dies ist zum Beispiel das Statement „flag = true — false“.

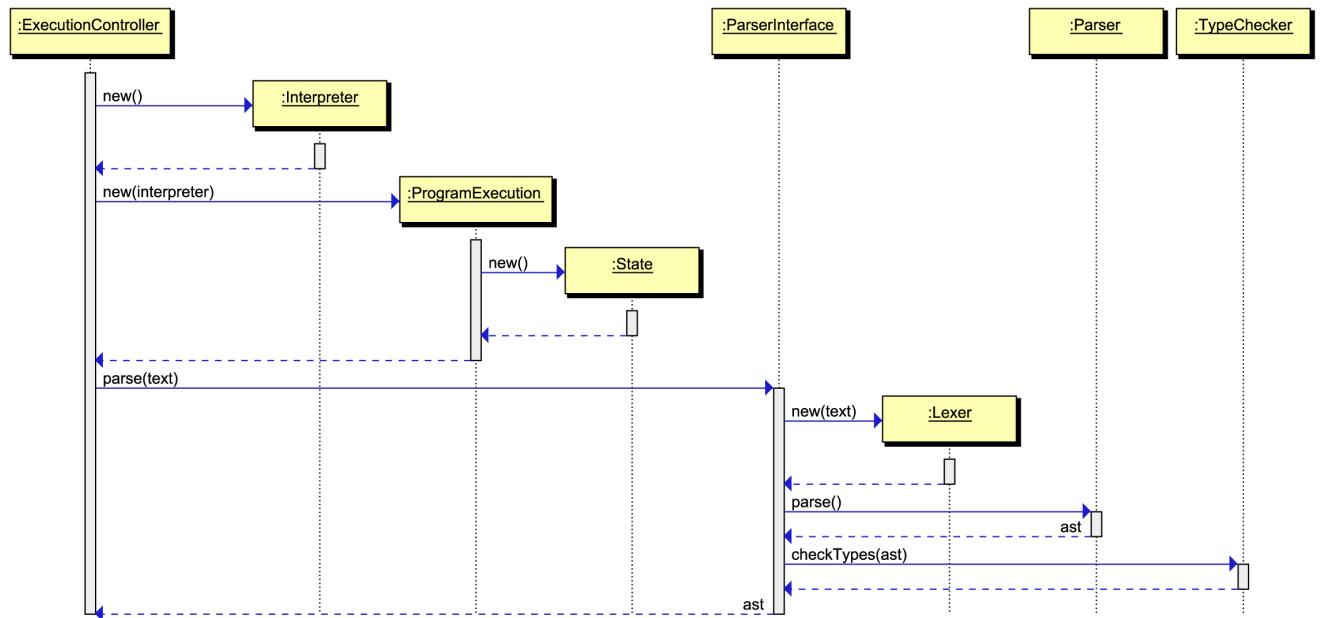
- Der Interpreter wird mit dem momentanen Programmzustand aufgerufen und ruft den Getter für das aktuelle Statement auf.
- Der Besucher ruft die accept-Methode darauf auf, welche ihrerseits die richtige visit-Methode aufruft.
- Bei visit(Assignment) wird zuerst der hintere Ausdruck ausgewertet (hier: „true — false“).
- Für diese Expression wird eine neue visit-Methode aufgerufen. Diese besucht zuerst die erste Subexpression, überprüft den Operanden, ob er unär oder binär ist. Da er binär ist, wird auch die zweite Subexpression besucht.
- Nachdem der Ausdruck ausgewertet worden ist, wird der letzte Bearbeitungsschritt der Zuweisung durchgeführt. Dazu wird der Variablenname gebraucht (assignment.getVar().getName()). Dann wird die Variable im State neu gesetzt.
- Der neue Zustand wird zurückgegeben.

4.2 Breakpoints



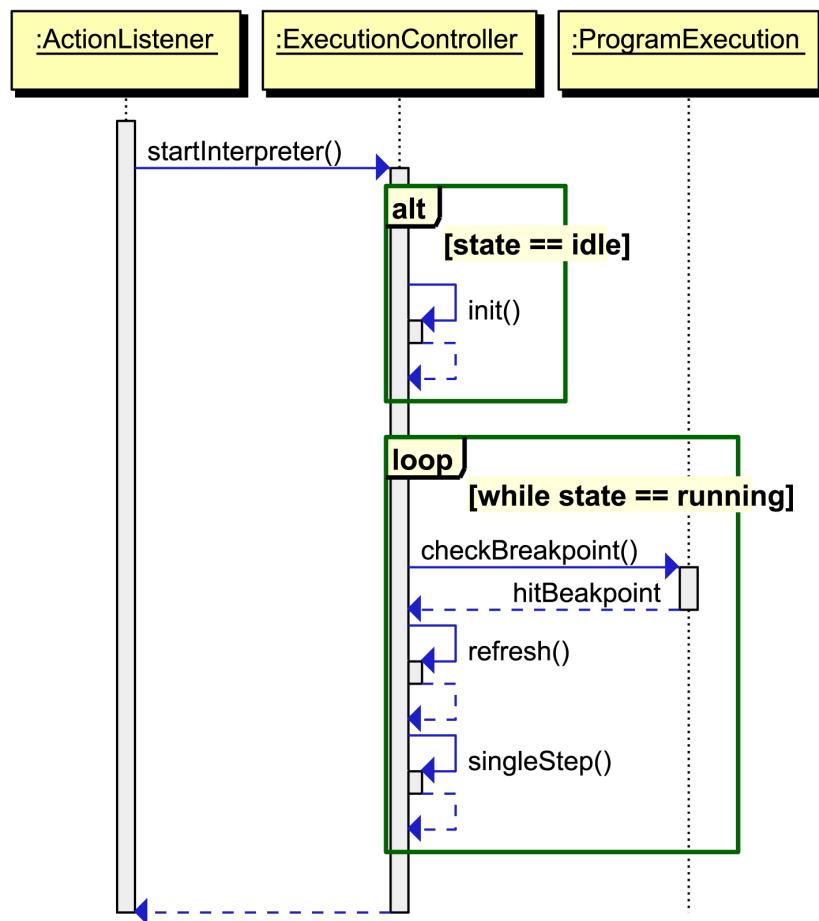
- Dieses Sequenzdiagramm zeigt, wie die Breakpoints ausgewertet werden. Dafür wird angenommen, dass zwei Statement-Breakpoints und ein globaler Breakpoint aktiv sind.
- Zuerst ruft die Instanz von ProgramExecution die Methode `getCommand` von State auf, um das aktuelle Statement zu bekommen.
- Für jeden einem Statement zugeordneten Breakpoint wird dieses Statement abgefragt und mit dem aktuell auszuführenden verglichen.
- Danach wird für jeden globalen Breakpoint die ihm zugeordnete Bedingung angefordert. State wird geklont, und die Bedingung wird mit der geklonten State-Instanz vom Interpreter ausgewertet.
- Der Rückgabewert von `checkBreakpoint()` ist genau dann ja, falls mindestens ein Breakpoint getroffen wurde.

4.3 Interpreter-Initialisierung



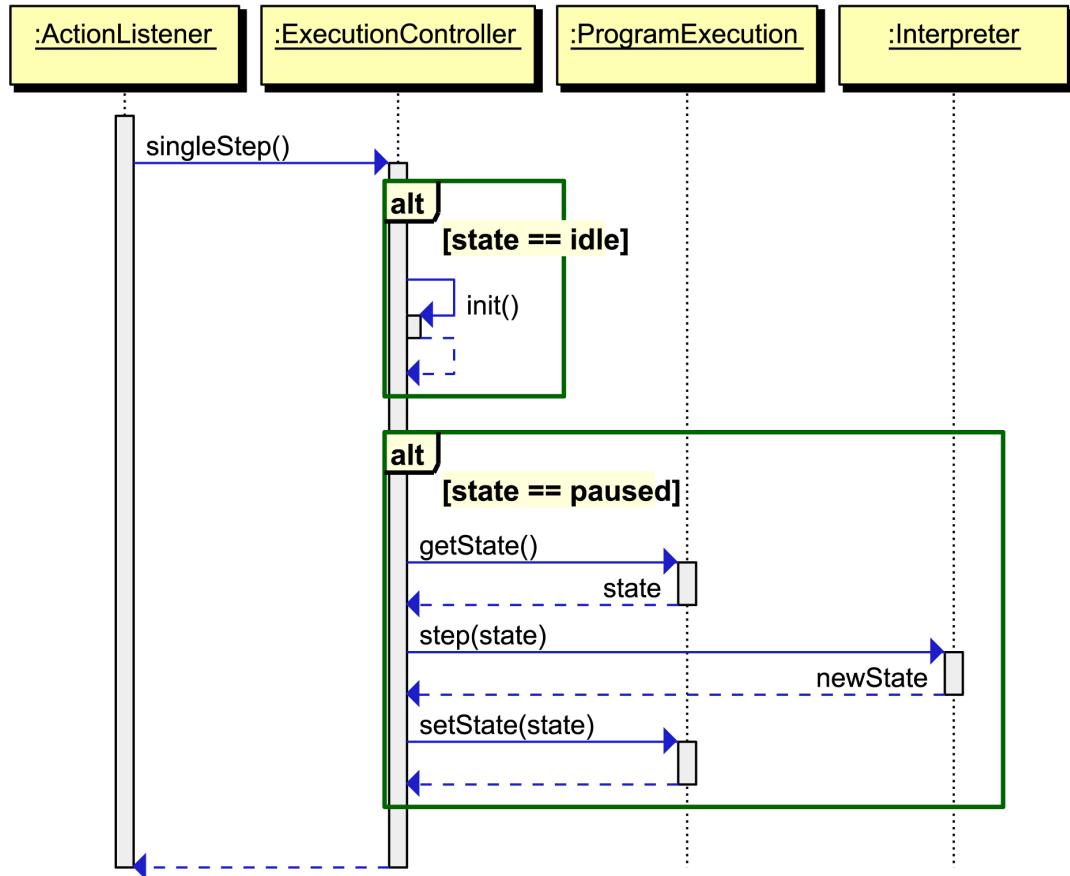
- Dieses Sequenzdiagramm zeigt die Initialisierung des Interpreters bei Programmstart.
- Der ExecutionController erzeugt jeweils eine neue Instanz von Interpreter und ProgramExecution, wobei letztere eine neue State-Instanz erzeugt.
- Dann wird das Parserinterface aufgerufen, welches eine Instanz des Lexers für diesen Parse-Vorgang erzeugt. Der Parser erzeugt daraus einen AST.
- Dieser wird vom TypeChecker auf Typkorrektheit überprüft.

4.4 Start-Button



- Beim Drücken auf den Play-Button wird der entsprechende ActionListener ausgeführt, welcher die startInterpreter-Methode des ExecutionController ausführt.
 - Ist das Programm noch nicht geparsst worden, wird der Interpreter initialisiert (s. letztes Diagramm).
 - Danach wird das Programm schrittweise (s. nächstes Diagramm) ausgeführt, wobei vor jedem Einzelschritt geprüft wird, ob ein Breakpoint getroffen oder das Programm pausiert wurde. Letzteres wird über das Beobachter-Muster realisiert.

4.5 Single-Step-Button



- Beim Drücken auf den SingleStep-Button wird der entsprechende ActionListener ausgeführt, welcher die `singleStep`-Methode des ExecutionController ausführt.
- Ist das Programm noch nicht geparsert worden, wird der Interpreter initialisiert (s. entsprechendes Diagramm).
- Danach wird ein Einzelschritt des Programms ausgeführt. Dafür wird der im ProgramExecution gekapselte Programmzustand abgefragt und an den Interpreter gegeben; der neue Zustand wird dann wieder zurückgespeichert.

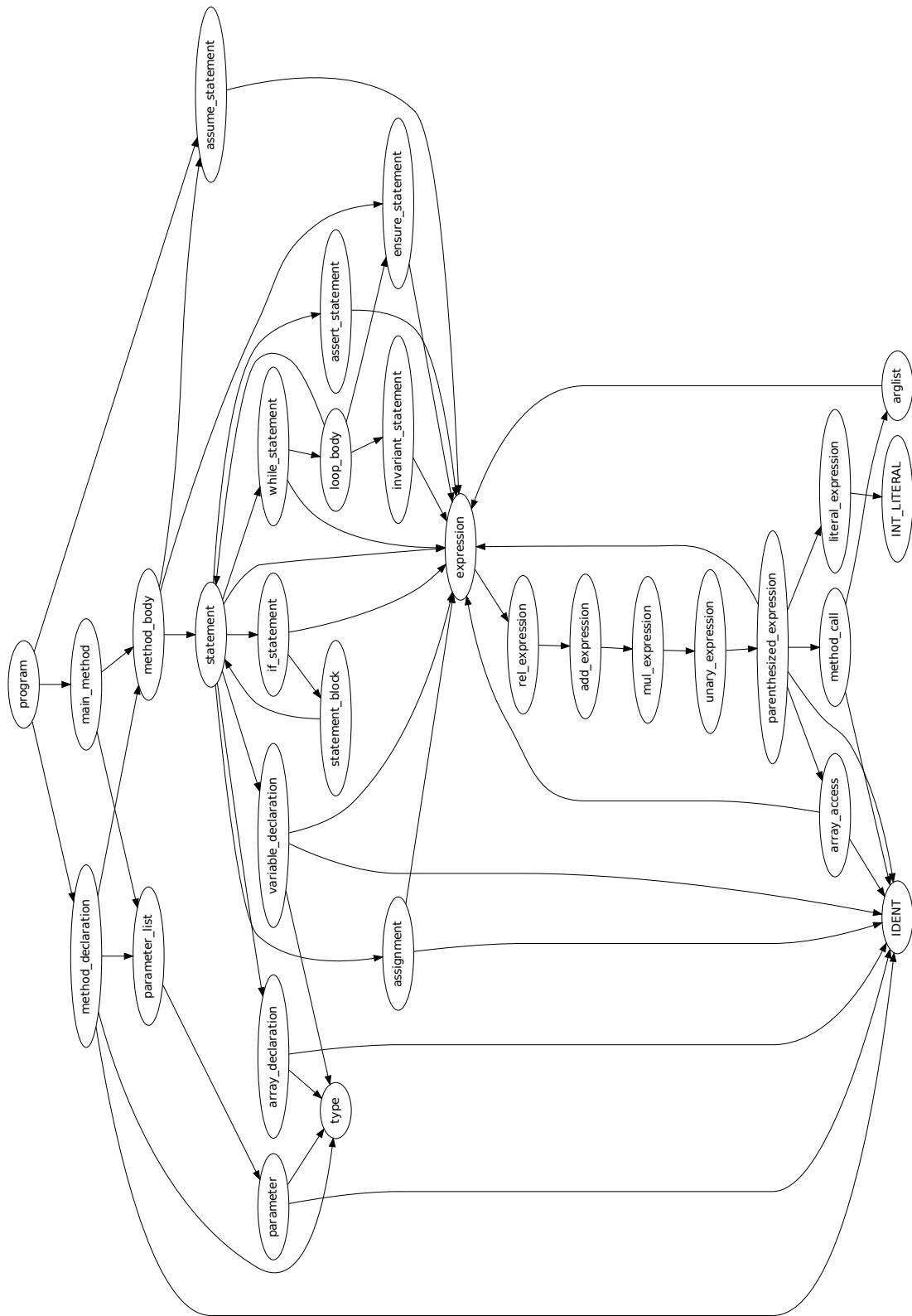
5 Erweiterbarkeit

Die Schnittstelle zum Beweiser ist so ausgelegt, dass jeder Beweiser, der das SMT LIB 2.0 Format unterstützt, verwendet werden kann und für eine weitere Anbindungen nur wenige Änderungen nötig sind.

Durch die gering gehaltenen Schnittstell des Parsers kann dieses Paket leicht geändert oder ersetzt werden, ohne dass der Rest des Programms davon beeinflusst wird.

Die While-Sprache ist durch ihre Grammatik festgelegt. Eine Erweiterung dieser Grammatik ist kein Problem für das Programm. Man kann die Sprache modifizieren und neue Befehle definieren. Dies wird weiter durch den Einsatz des Visitor-Patterns für Interpreter, Typchecker und Beweiseranbindung begünstigt, da hierdurch für jedes weitere Syntaxelement der Sprache nur eine Methode pro Programmelement, dass den abstrakten Syntaxbaum traversiert, implementiert werden muss. Auch das Hinzufügen und Ändern von Typen für diese Sprache ist problemlos möglich.

6 Struktur der While-Sprache



7 Syntax der While-Sprache

```
grammar WhileLanguage;

program
    : assume_statement* method_declaration* main_method
    ;

method_declaration
    : type IDENT '(' parameter_list? ')' method_body
    ;

main_method
    : 'main' '(' parameter_list? ')' method_body
    ;

parameter_list
    : parameter ( ',' parameter )*
    ;

parameter
    : type IDENT
    ;

method_body
    : '{' assume_statement* statement* ensure_statement* '}'
    ;

statement
    : assert_statement
    | variable_declaraction
    | array_declaraction
    | assignment
    | if_statement
    | while_statement
    | 'return' expression ';'
    ;

invariant_statement
    : 'invariant' quantified_expression ';'
    ;

assert_statement
    : 'assert' quantified_expression ';'
    ;

assume_statement
    : 'assume' quantified_expression ';'
    ;

ensure_statement
    : 'ensure' quantified_expression ';'
    ;

assignment
    : IDENT ( '[' expression ']' )* '=' expression ';'
    ;
```

```

;

variable_declarator
: type IDENT ( '=' expression )? ';'*
;

array_declarator
: type IDENT ( '[' ']' )+ ';'*
;

if_statement
: 'if' '(' expression ')' statement_block ( 'else' statement_block )?
;

statement_block
: '{' statement* '}'
;

while_statement
: 'while' '(' expression ')' loop_body
;

loop_body
: '{' invariant_statement* statement* ensure_statement* '}'
;

quantified_expression
: quantifier IDENT '(' range? ')', quantified_expression
| expression
;

quantifier
: 'forall'
| 'exists'
;

range
: expression ',' expression
;

expression
: rel_expression ( ( '==' | '!=') rel_expression )*
;

rel_expression
: add_expression ( ( '<' | '<=' | '>' | '>=' ) add_expression )*
;

add_expression
: mul_expression ( ( '*' | '+' | '-' ) mul_expression )*
;

mul_expression
: unary_expression ( ('&' | '*' | '/' | '%') unary_expression )*
;

```

```

unary_expression
: ('!' | '+' | '-')? parenthesized_expression
;

parenthesized_expression
: '(' expression ')'
| method_call
| array_access
| IDENT
| literal_expression
;

method_call
: IDENT '(' arglist? ')'
;

arglist
: expression ( ',' expression )*
;

array_access
: IDENT '[' expression ']' ( '[' expression ']')*
;

literal_expression
: INT_LITERAL
| BOOL_LITERAL
;

type
: ('int' | 'bool') ( '[' ''])*
;

INT_LITERAL : ('0'..'9')+;
BOOL_LITERAL : 'true' | 'false';
COMMENT : '#' .* ( '\n' | '\r' ) {skip();};
WS : ('\n' | '\r' | ' ' | '\t')+ {skip();};
IDENT : ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*;

```

8 Implementierungsplan

8.1 Vorgangsliste

Automatisches Prüfen der Korrektheit von Programmen
27.12.11–02.02.12

Vorgangsliste			
Vorgang	Anfang	Ende	Ressource
Implementation	27.12.11	02.02.12	Simon Bischof Tobias Schlumberger Adrian Herrmann Jan Haag Matthias Schnetz Lin Jin
Parser & Lexer	27.12.11	28.12.11	Jan Haag
AST	27.12.11	31.12.11	Simon Bischof Jan Haag
Visitor	02.01.12	27.01.12	Simon Bischof Tobias Schlumberger Adrian Herrmann Jan Haag
Type Checker Möglichst bis 5.1.12	02.01.12	07.01.12	Simon Bischof Jan Haag
Interpreter	09.01.12	14.01.12	Simon Bischof Tobias Schlumberger Adrian Herrmann Jan Haag
SMTLib Translator	12.01.12	27.01.12	Simon Bischof Tobias Schlumberger Adrian Herrmann Jan Haag
GUI Prototype	27.12.11	31.12.11	Tobias Schlumberger
GUI Frames	02.01.12	07.01.12	Tobias Schlumberger Lin Jin
GUI Logic	09.01.12	27.01.12	Matthias Schnetz

Verifier Interface
Buffer & Intergration

05.01.12
27.01.12

07.01.12
02.02.12

Automatisches Prüfen der Korrektheit von Programmen
27.12.11–02.02.12

Lin Jin
Adrian Herrmann
Simon Bischof
Tobias Schlumberger
Adrian Herrmann
Jan Haag
Matthias Schnetz
Lin Jin

8.2 Gantt-Diagramm

Automatisches Prüfen der Korrektheit von Programmen
27.12.11-02.02.12

