

# 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

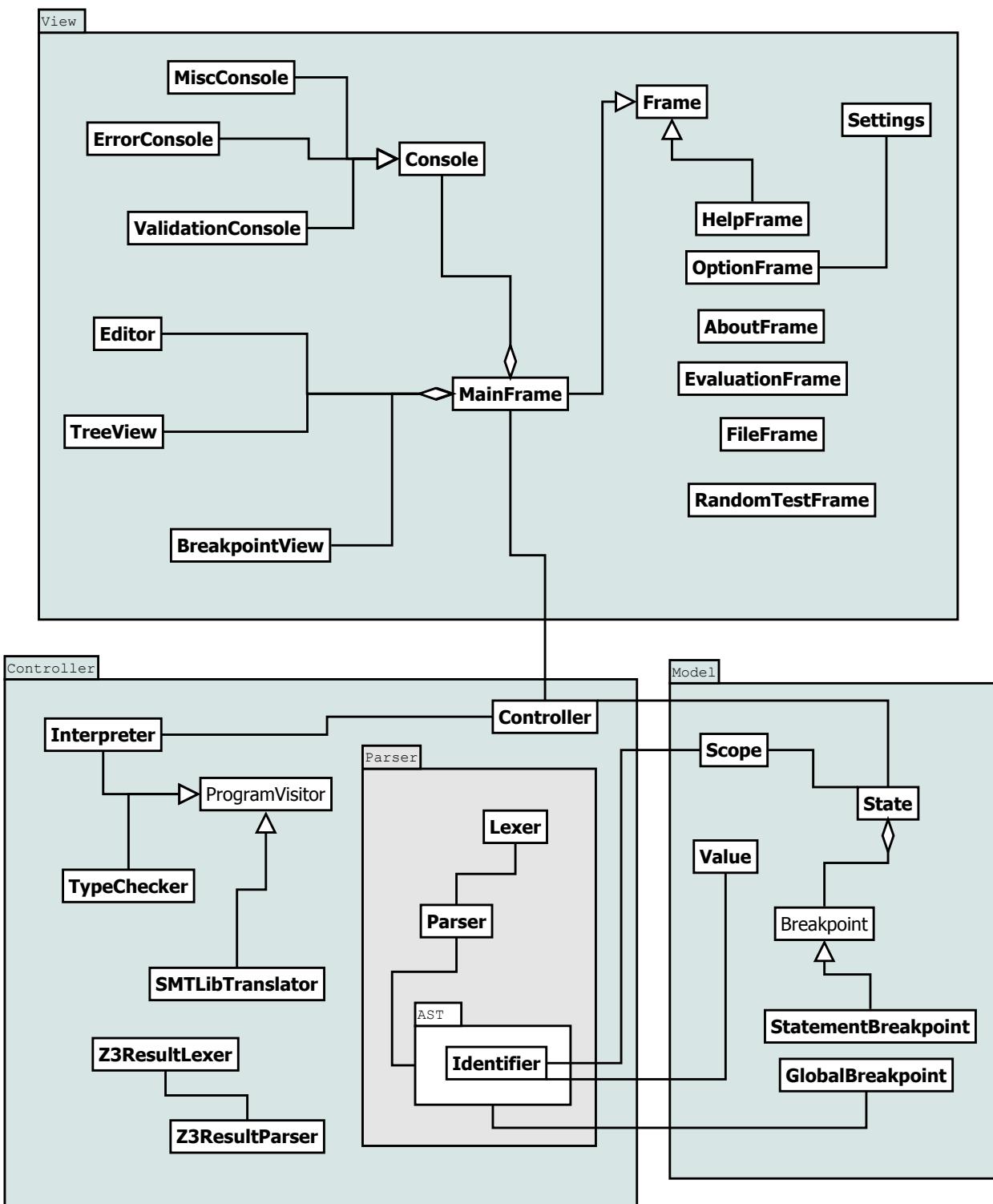
<b>1 Einleitung</b>	<b>3</b>
<b>2 Klassendiagramme</b>	<b>3</b>
2.1 Übersicht . . . . .	3
2.2 Feinstruktur der Komponenten . . . . .	5
2.2.1 Programmstruktur . . . . .	5
2.2.2 Besucherklassen . . . . .	7
2.2.3 Parser . . . . .	8
2.2.4 Modell . . . . .	9
2.2.5 Benutzeroberfläche . . . . .	11
<b>3 Aktivitätsdiagramme</b>	<b>13</b>
3.1 Parser/Type-Checker . . . . .	13
3.2 Z3-Anbindung . . . . .	14
<b>4 Zustandsdiagramm</b>	<b>15</b>
<b>5 Sequenzdiagramme</b>	<b>16</b>
5.1 Besucher-Muster . . . . .	16
5.2 Breakpoints . . . . .	17
5.3 Interpreter-Initialisierung . . . . .	18
5.4 Start-Button . . . . .	19
5.5 Single-Step-Button . . . . .	20
<b>6 Erweiterbarkeit</b>	<b>20</b>
<b>7 Struktur der While-Sprache</b>	<b>22</b>
<b>8 Syntax der While-Sprache</b>	<b>23</b>

# 1 Einleitung

## 2 Klassendiagramme

### 2.1 Übersicht

Das folgende Klassendiagramm zeigt die Grobstruktur der Anwendung.

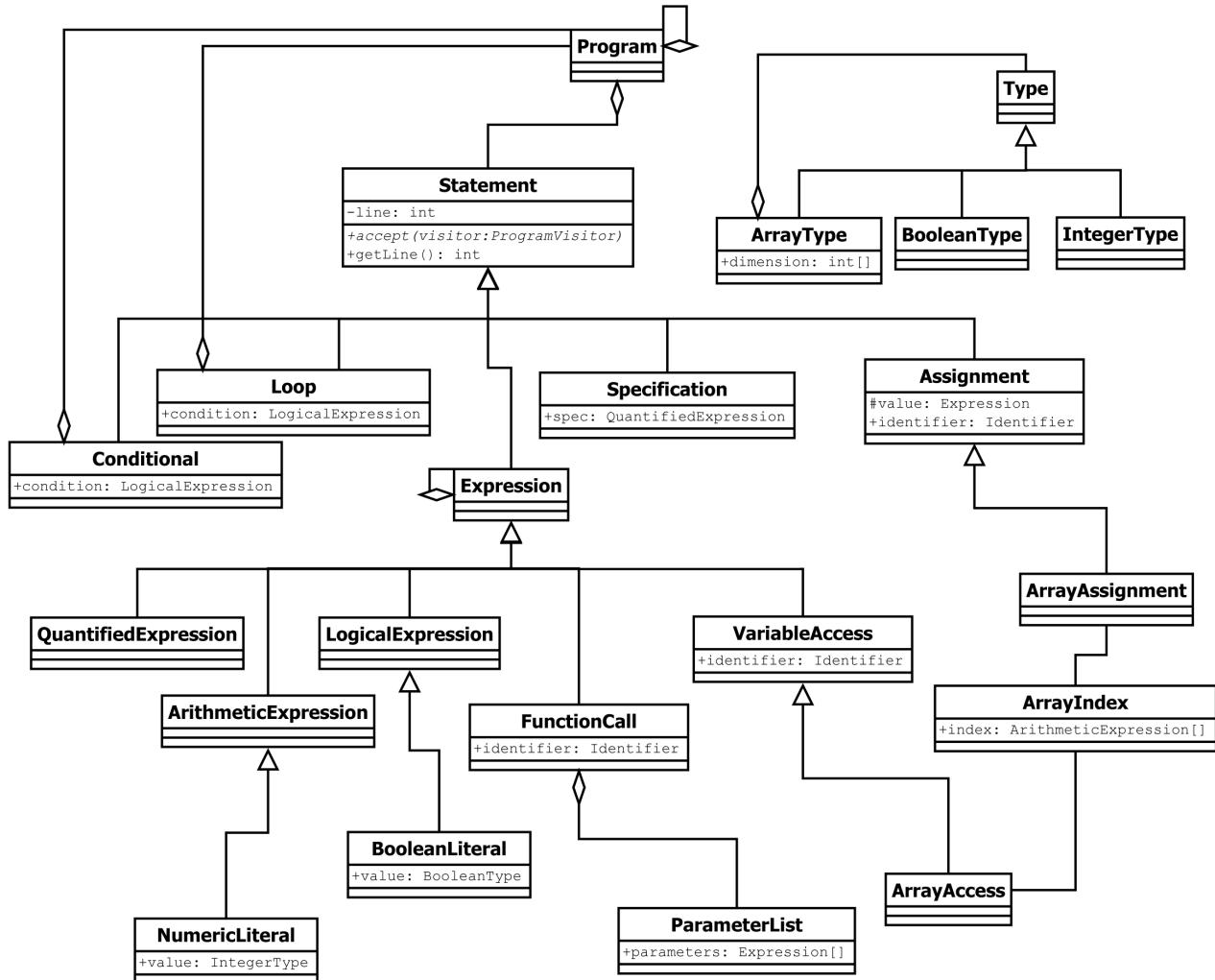


Die gesamte Architektur basiert auf dem Entwurfsmuster Model-View-Controller. Dies ermöglicht uns einen flexiblen Programmentwurf, der eine spätere Änderung oder Erweiterung erleichtert und eine Wiederverwendbarkeit der einzelnen Komponenten ermöglicht.

- Das **Modell** enthält die darzustellenden Daten, z.B die Zustände der Variablen während der Programm-ausführung und die vom Benutzer gesetzten Breakpoints
- Die **Präsentation/View** stellt die Daten aus dem Modell auf der Benutzeroberfläche dar und nimmt Benutzerinteraktionen entgegen
- Um die Zusammenarbeit der ersten zwei Komponenten kümmert sich die **Steuerung**. Außerdem führt sie die Hauptaktionen (Parlsen, Interpretieren, usw.) aus und bearbeitet die Eingaben des Benutzers

## 2.2 Feinstruktur der Komponenten

### 2.2.1 Programmstruktur



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.

- **Program**

Die Klasse **Program** stellt eine Funktion im Quelltext dar.

- Attribute:

**program**

eine Instanz von sich selbst. Diese sind Unterprogramme, die aufgerufen werden können.

**statement**

eine Liste von Instanzen der Klasse **Statement**

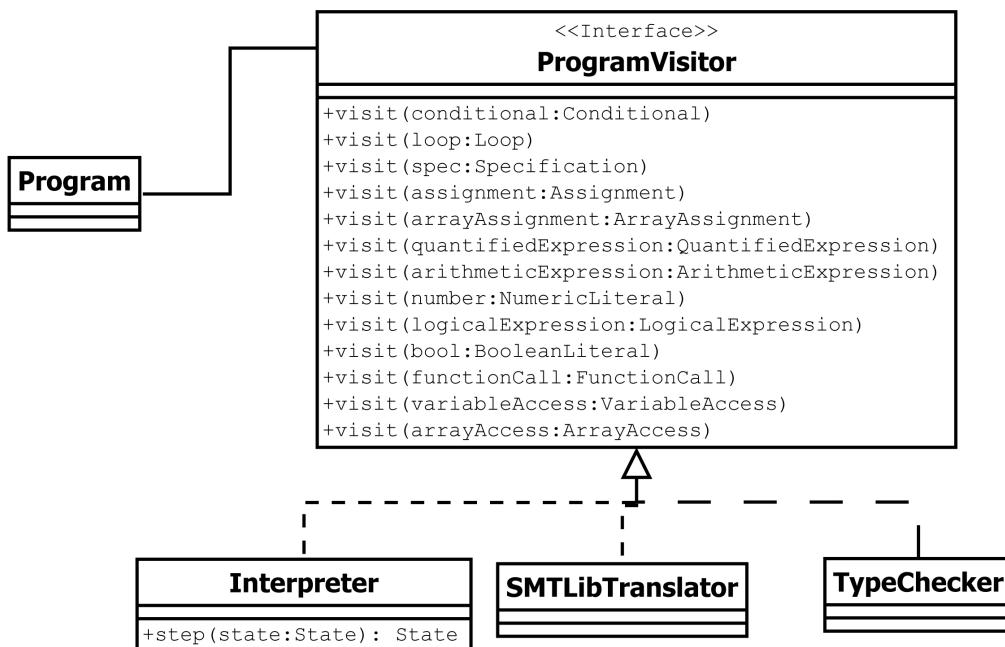
- **Statement**

Die abstrakte Klasse **Statement** steht für eine Anweisung des Programms. Die Anweisungen sind eingeteilt in fünf Klassen: **Conditional**, **Loop**, **Specification**, **Assignment** und **Expression**.

- Attribut:  
`line`  
Zeile im Quelltext, in der sich die Instanz befindet
- Methoden:  
`getLine()`  
gibt das Attribut `line` zurück  
`accept(visitor: ProgramVisitor)`  
abstrakte Methode, die von den Unterklassen implementiert wird
- **Conditional**  
Die Klasse **Conditional** steht für eine bedingte Anweisung.
  - Attribute:  
`program`  
Unterprogramme, die bedingt ausgeführt werden  
`condition`  
Bedingung, die überprüft wird
- **Loop**  
Die Klasse **Loop** steht für eine while-Anweisung.
  - Attribute:  
`program`  
Unterprogramme, die bedingt ausgeführt werden  
`condition`  
Bedingung, die überprüft wird
- **Specification**
  - Attribut:  
`spec`  
Ausdruck, der überprüft wird
- **Assignment**  
Die Klasse **Assignment** steht für eine Zuweisung von Variablen. Die Klasse **ArrayAssignment** ist eine Subklasse, die stattdessen für Zuweisung von Arrays steht.
  - Attribute:  
`value`  
der Wert, der zugewiesen wird  
`identifier`  
die Variable, der `value` zugewiesen wird
- **Expression**  
Die abstrakte Klasse **Expression** steht für einen beliebigen Ausdruck. **QuantifiedExpression**, **ArithmeticExpression**, **LogicalExpression**, **FunctionCall** und **VariableAccess** sind Unterklassen dieser Klasse.
- **FunctionCall**  
Die Klasse **FunctionCall** modelliert einen Methodenaufruf.
  - Attribute:  
`identifier`  
Name der Methode  
`paralist`  
Liste von Parametern, die der Methode übergeben werden
- **VariableAccess**  
Die Klasse **VariableAccess** stellt einen Zugriff auf eine vorhandene Variable dar. Von dieser Klasse erbt die Unterklsse **ArrayAccess**.

- Attribut:  
**identifier**  
Name der Variable
- **Type**  
Die Klasse **Type** bildet eine Oberklasse der verschiedenen Typen, die eine Variable haben kann. Dazu gehören **BooleanType**, **IntegerType** und **ArrayType**.
- **ArrayType**  
Die Klasse **ArrayType** definiert einen eigenen Typ der Array-Objekte.
  - Attribute:  
**dimension**  
Dimension des Arrays  
**type**  
Arrayelemente haben „normale“ Typen

## 2.2.2 Besucherklassen

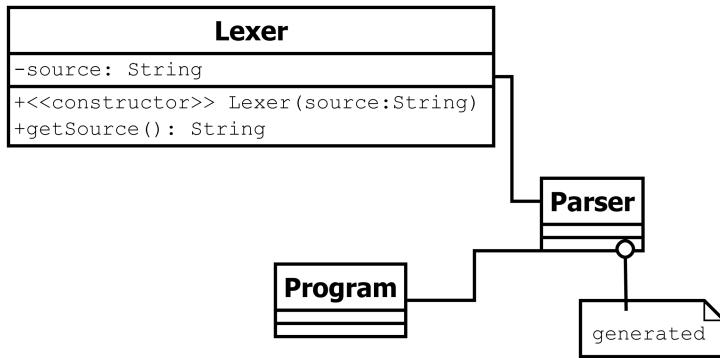


Für die Integration der verschiedenen Methoden, wie zum Beispiel die des Interpreters, verwenden wir das Visitor-Muster. Es kapselt die Operationen, die auf Elemente des AST ausgeführt werden. Das Muster ermöglicht uns, Operationen zu verändern oder neue Funktionen hinzufügen, ohne die Elementklassen zu modifizieren.

- **ProgramVisitor**  
Das Interface **ProgramVisitor** deklariert für jede Klasse konkreter Elemente eine Besuchsfunktion.
- **Interpreter**  
Die Klasse **Interpreter** ist ein konkreter Besucher und implementiert die Aufgaben des Interpretierens als Besuchsfunktionen.

- 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.

### 2.2.3 Parser



Die Klassen der Komponente Parser werden vom Parsergenerator ANTLR erstellt. Der Parser liefert die Struktur des AST.

- **Lexer**

Die Klasse **Lexer** zerlegt den Quelltext in Tokens.

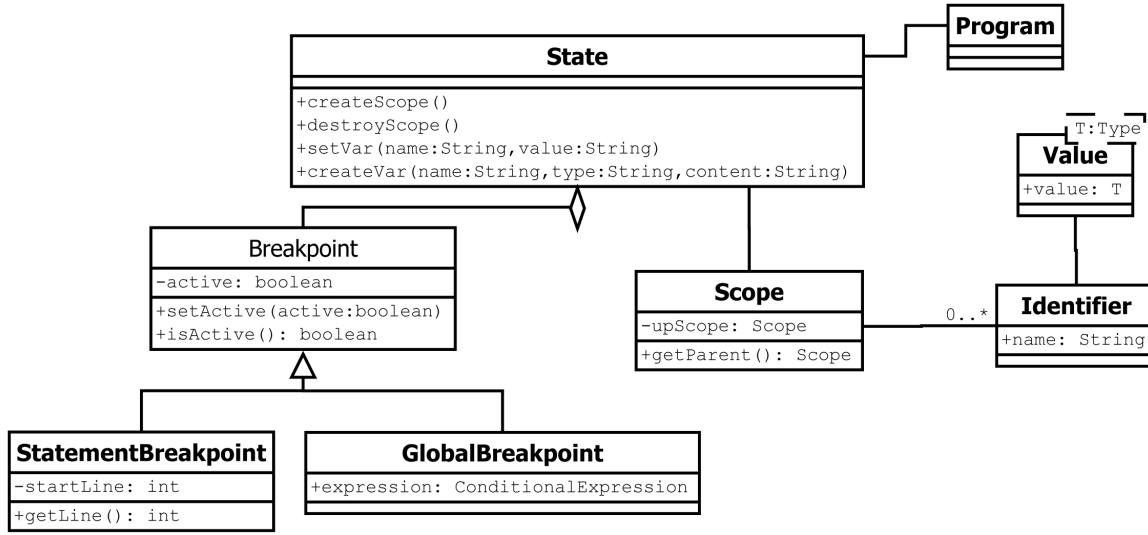
- Attribut:

`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

## 2.2.4 Modell



Die Komponente Modell speichert alle Daten der Anwendung. Diese erhält sie von den Komponenten View (z.B. Benutzereingaben) und Controller (z.B. Zustand der Programmausführung). Außerdem ist es ihre Aufgabe, Daten auf Anforderung der anderen Komponenten bereit zustellen.

- **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.

- Attribute:

`upScope`

eine andere Instanz der Klasse, die in der Hierarchie über diese Instanz liegt

`identifier`

Name der Variable, deren Sichtbarkeit durch diese Instanz dargestellt wird

- Methoden:

`getParent()`

gibt die „Eltern“-Instanz `upScope` zurück

- **Value**

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

- Attribute:

`value`

Attribut mit einem generischen Typ, der `BooleanType`, `IntegerType` oder `ArrayType` sein kann.

`identifier`

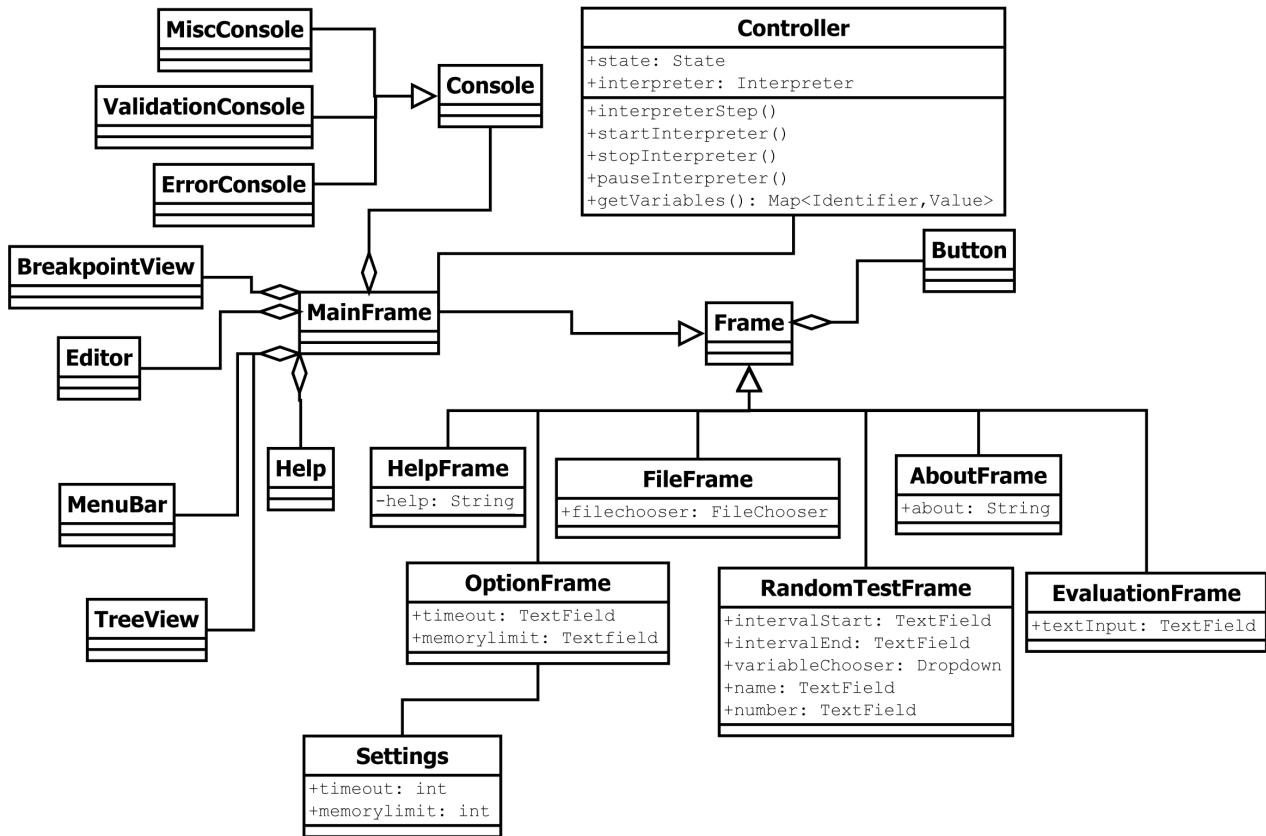
Name der Variable, die den Wert `value` besitzt

- **State**

Die Klasse **State** speichert den aktuellen Zustand der Programmausführung, wie z.B. Werte von Variablen. Außerdem ordnet sie den Variable ihre Sichtbarkeit zu.

- 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 mithilfe eines Stringinputs
  - `createVar(name:String, type:String, content:String)`  
erstellt eine Variable mit Typ und Inhalt mithilfe eines Stringinputs
- **Breakpoint**  
Die abstrakte Klasse `Breakpoint` hat zwei Subklassen: `StatementBreakpoint` und `GlobalBreakpoint`.
  - Attribut:  
`active`  
boolsche Variable, die anzeigt, ob die `Breakpoint`-Instanz aktiviert ist
  - Methoden:
    - `setActive(active:boolean)`  
aktiviert oder deaktiviert diese `Breakpoint`-Instanz
    - `isActive()`  
gibt `active` zurück
- **StatementBreakpoint**  
Die Klasse `StatementBreakpoint` stellt einen an einer Zeile gebundenen Breakpoint dar.
  - Attribut:  
`startLine`  
Zeile, in der sich der Breakpoint befindet
  - Methode:  
`getLine()`  
gibt `startLine` zurück
- **GlobalBreakpoint** Die Klasse `GlobalBreakpoint` stellt einen globalen Breakpoint dar, der an einem Ausdruck gebunden ist.
  - Attribut:  
`expression`  
Ausdruck, an dem die Instanz gebunden ist

## 2.2.5 Benutzeroberfläche



Die Benutzeroberfläche bietet dem Benutzer eine leicht zu bedienende Schnittstelle und nimmt dessen Eingaben entgegen. Um eine flexible Struktur zu gestalten, verwenden wir hier das Facade-Muster in Form der **Controller**-Klasse. Diese vereinfacht die Schnittstelle zu den restlichen Komponenten und fördert die lose Kopplung. Über sie geschehen alle Interaktionen mit anderen Komponenten, z.B. reicht sie Benutzereingaben weiter, damit diese verarbeitet werden können. Der **Controller** stellt der GUI auch die von ihr benötigten Informationen zur Verfügung.

- **Controller**

Die Klasse **Controller** stellt eine Verbindung zwischen dem Modell und der View her.

- Attribute:
  - state**  
Zustand des Programms
  - interpreter**  
die Interpreter-Instanz, die das aktuelle Programm ausführt
- Methoden:
  - interpreterStep()**  
schrittweise Ausführung des Programms
  - startInterpreter()**  
Ausführung des gesamten Programms
  - stopInterpreter()**  
Ausführung abbrechen
  - pauseInterpreter()**

Ausführung pausieren

`getVariables()`

gibt die aktuellen Variablen und deren Werte zurück

- **Frame**

Die abstrakte Klasse **Frame** stellt alle Fenster dar, die vom Benutzer geöffnet werden können. Sie ist Oberklasse der folgenden Klassen: **HelpFrame**, **FileFrame**, **OptionFrame**, **RandomTestFrame**, **AboutFrame**, **EvaluationFrame** und **MainFrame**.

- **HelpFrame**

Die Klasse **HelpFrame** erleichtert die Bedienung der Anwendung.

- `help`

Inhalt der Hilfe

- **OptionFrame**

Die Klasse **OptionFrame** speichert Benutzereinstellungen für den Beweiser.

- Attribut:

`settings`

in der **Settings**-Instanz sind die vom Benutzer festgelegte Werte für `timeout` und `memorylimit` gespeichert

- **MainFrame**

Die Klasse **MainFrame** ist das Hauptfenster der Benutzeroberfläche. Sie besteht aus einem **Editor**, einer **MenuBar**, einer **BreakpointView**, einer **TreeView**, mehreren **Consoles** und einem **Help**-Fenster.

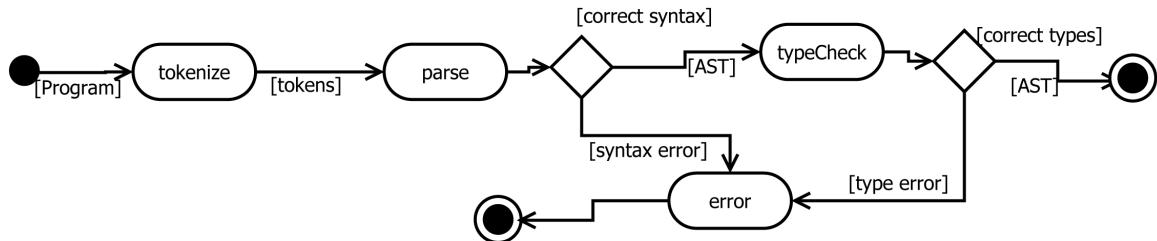
- Attribut:

`controller`

vereinfachte Schnittstelle, leitet Funktionalitäten weiter

### 3 Aktivitätsdiagramme

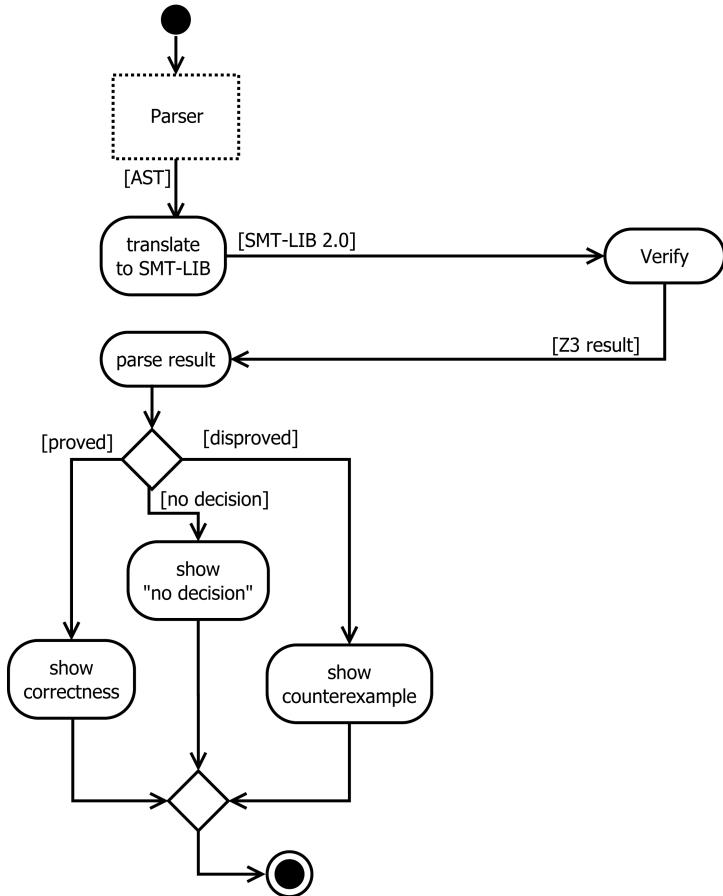
#### 3.1 Parser/Type-Checker



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.

### 3.2 Z3-Anbindung

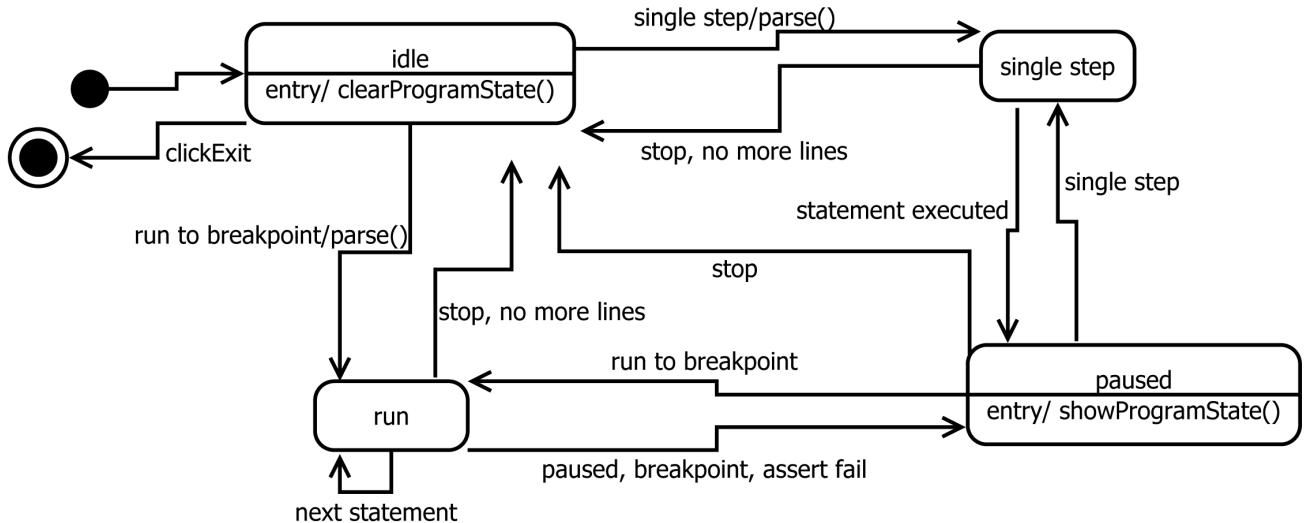


Zur Überprüfung der Korrektheit des Programms wird Z3 benutzt.

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.

## 4 Zustandsdiagramm

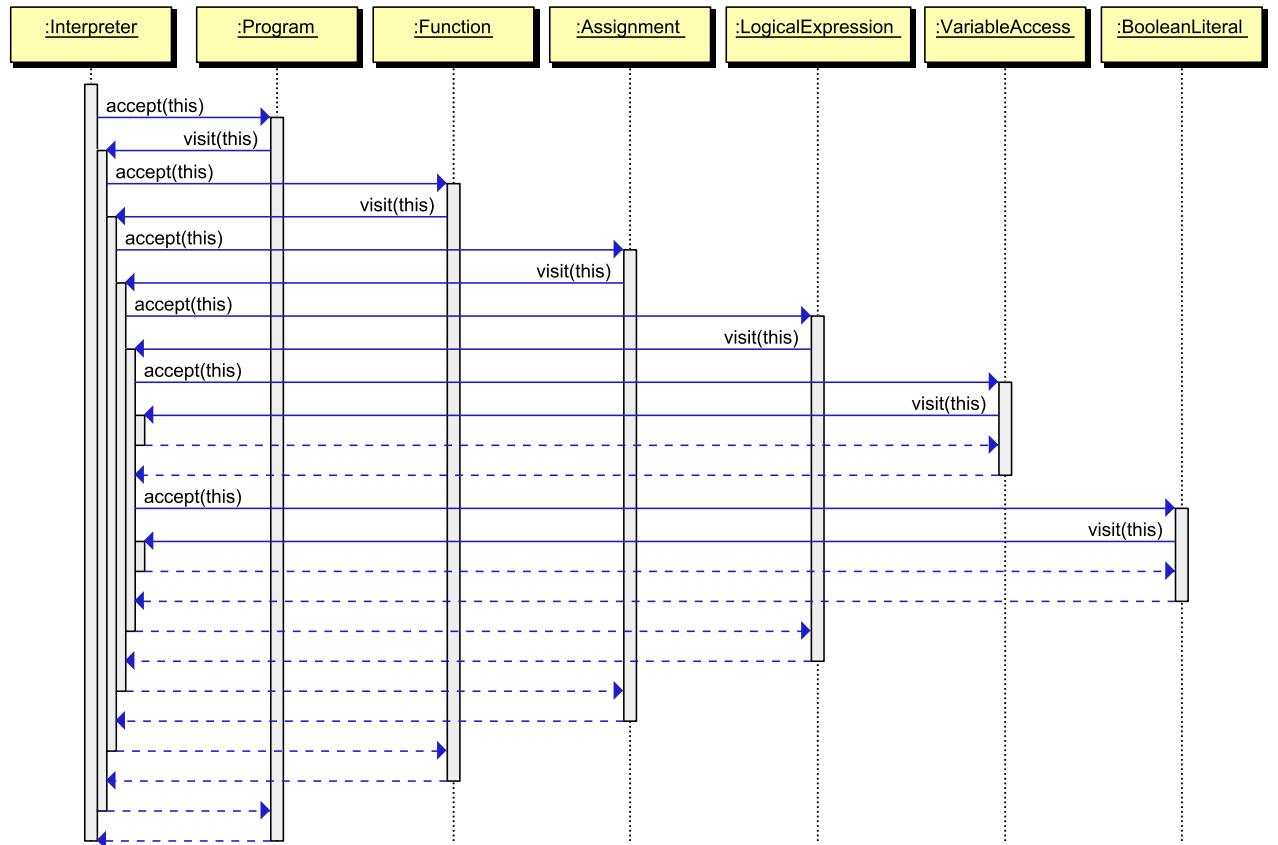
Das folgende Zustandsdiagramm zeigt das Verhalten des Systems bei Benutzeraktionen.



- 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.

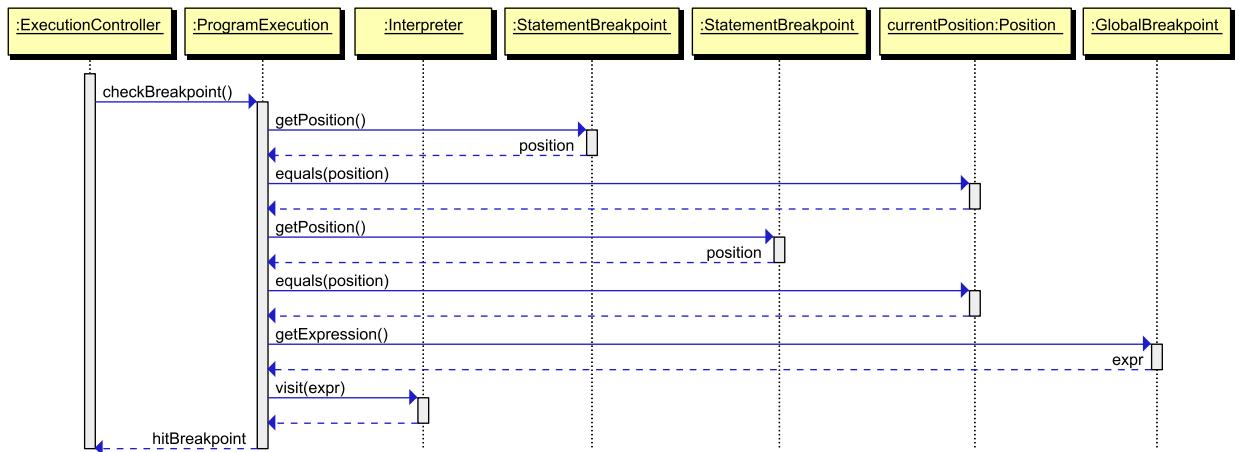
## 5 Sequenzdiagramme

### 5.1 Besucher-Muster



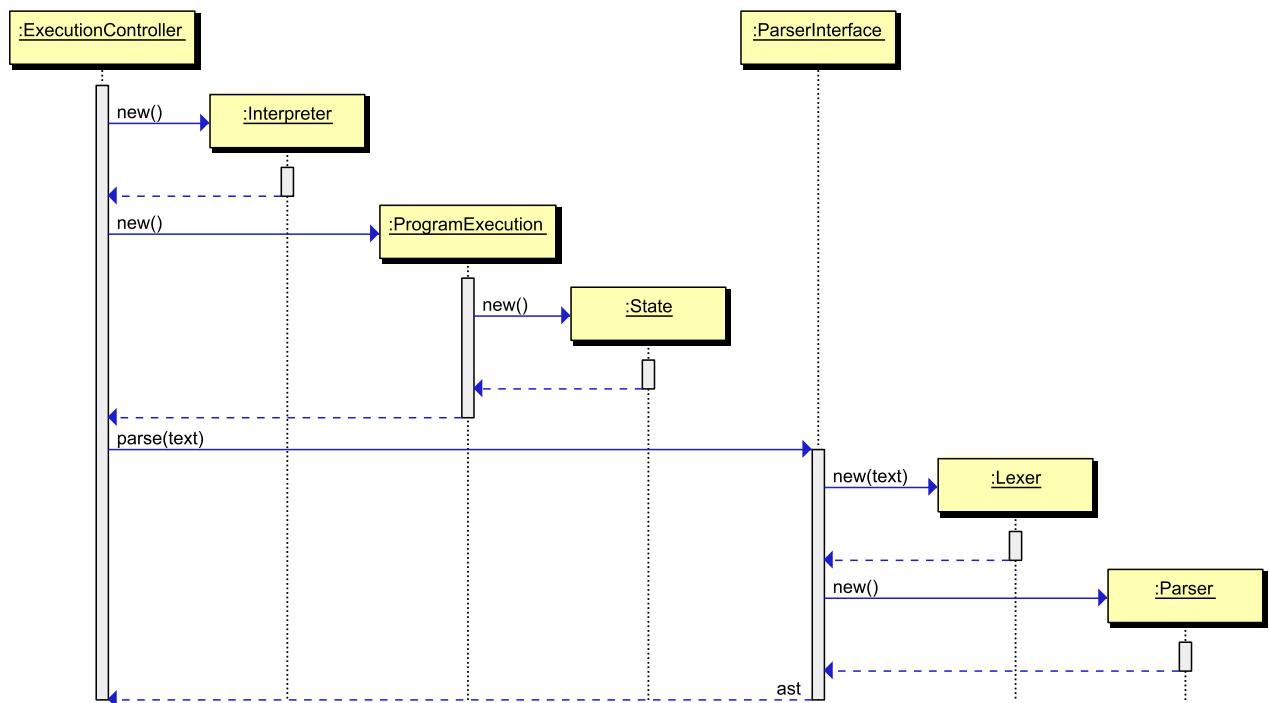
Dieses Sequenzdiagramm zeigt die Arbeitsweise des Besuchermusters an einem Beispiel.

## 5.2 Breakpoints



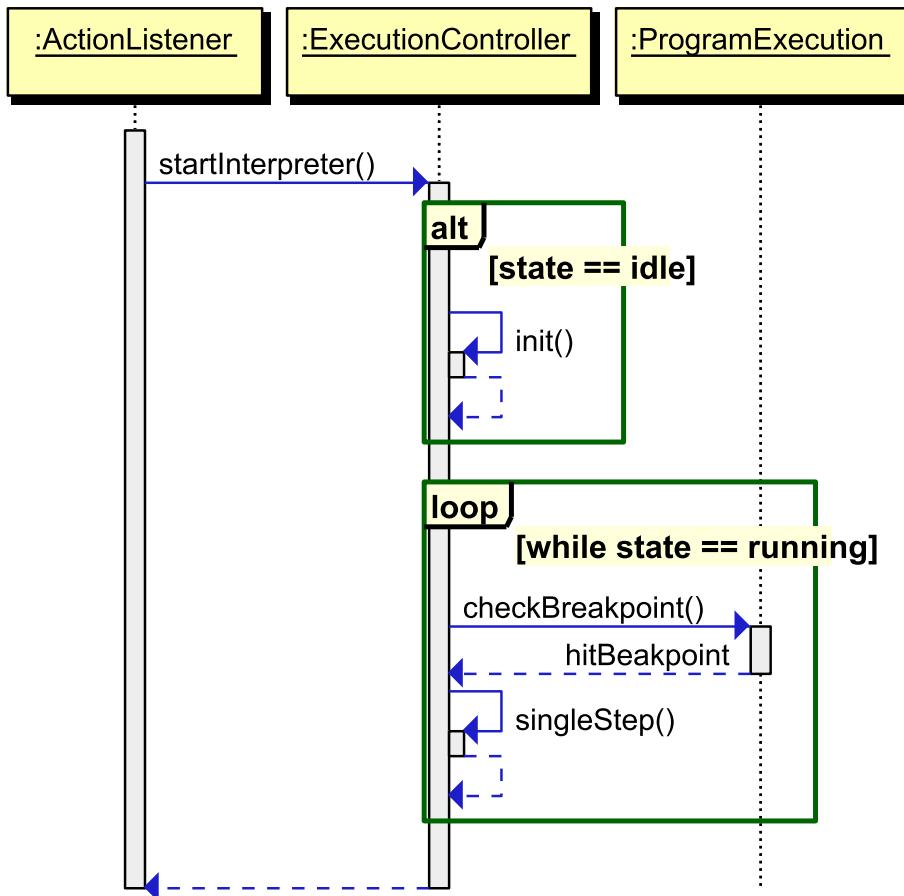
- Dieses Sequenzdiagramm zeigt, wie die Breakpoints ausgewertet werden. Dafür wird angenommen, dass zwei Statement-Breakpoints und ein globaler Breakpoint aktiv sind.
- Zuerst wird für jeden einem Statement zugeordneten Breakpoint die Position abgefragt und mit der aktuellen Programmversion verglichen.
- Danach wird für jeden globalen Breakpoint die ihm zugeordnete Bedingung vom Interpreter ausgewertet.
- Der Rückgabewert von `checkBreakpoint()` ist genau dann ja, falls mindestens ein Breakpoint getroffen wurde.

### 5.3 Interpreter-Initialisierung



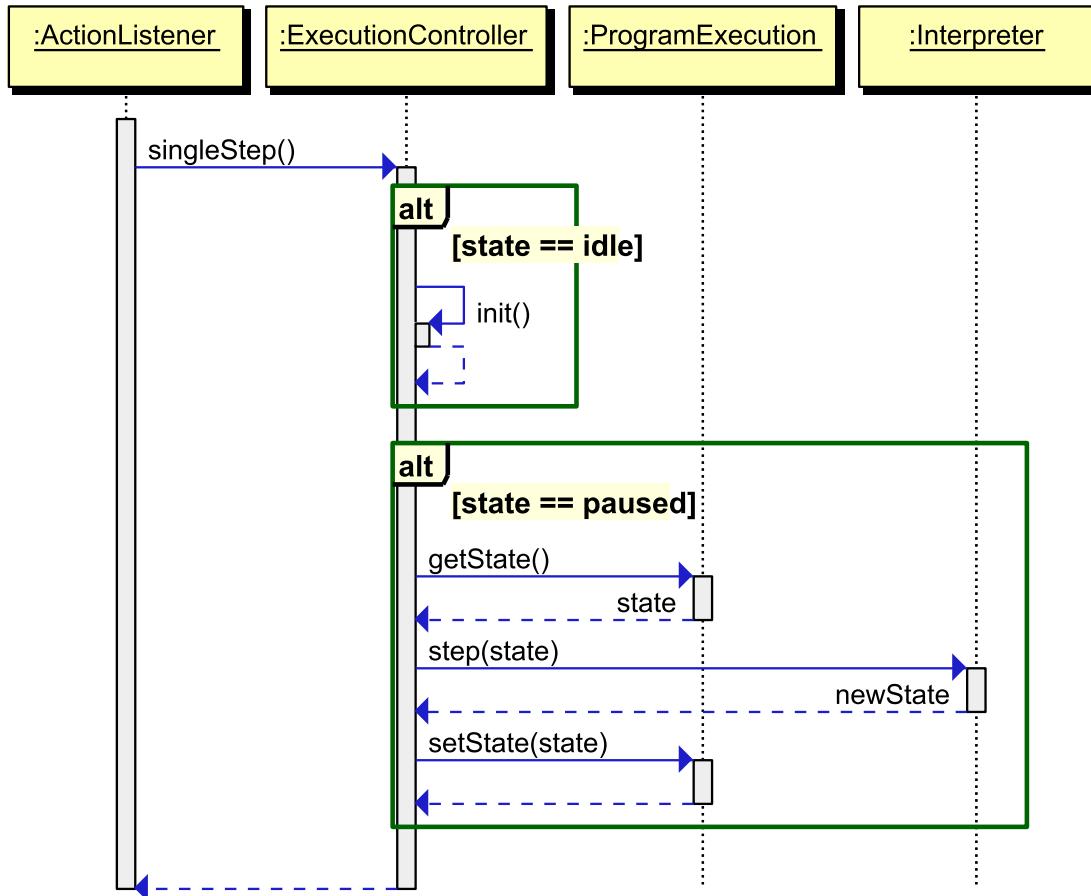
- Dieses Sequenzdiagramm zeigt die Initialisierung des Interpreters bei Programmstart.
- Der ExecutionController erzeugt jeweils eine neue Instanz von Interpreter und ProgramExecution, wobei letztere eine neue State-Instanz erzeugt.
- Dann wird das Parserinterface aufgerufen, welches Instanzen von Lexer und Parser für diesen Parse-Vorgang erzeugt und einen AST zurückgibt.

## 5.4 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 geparsert worden, wird der Interpreter initialisiert (s. letztes Diagramm).
- Danach wird das Programm solange schrittweise (s. nächstes Diagramm) ausgeführt, bis ein Breakpoint getroffen wird, das Programm pausiert wird oder eine Assertion fehlschlägt.

## 5.5 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.

## 6 Erweiterbarkeit

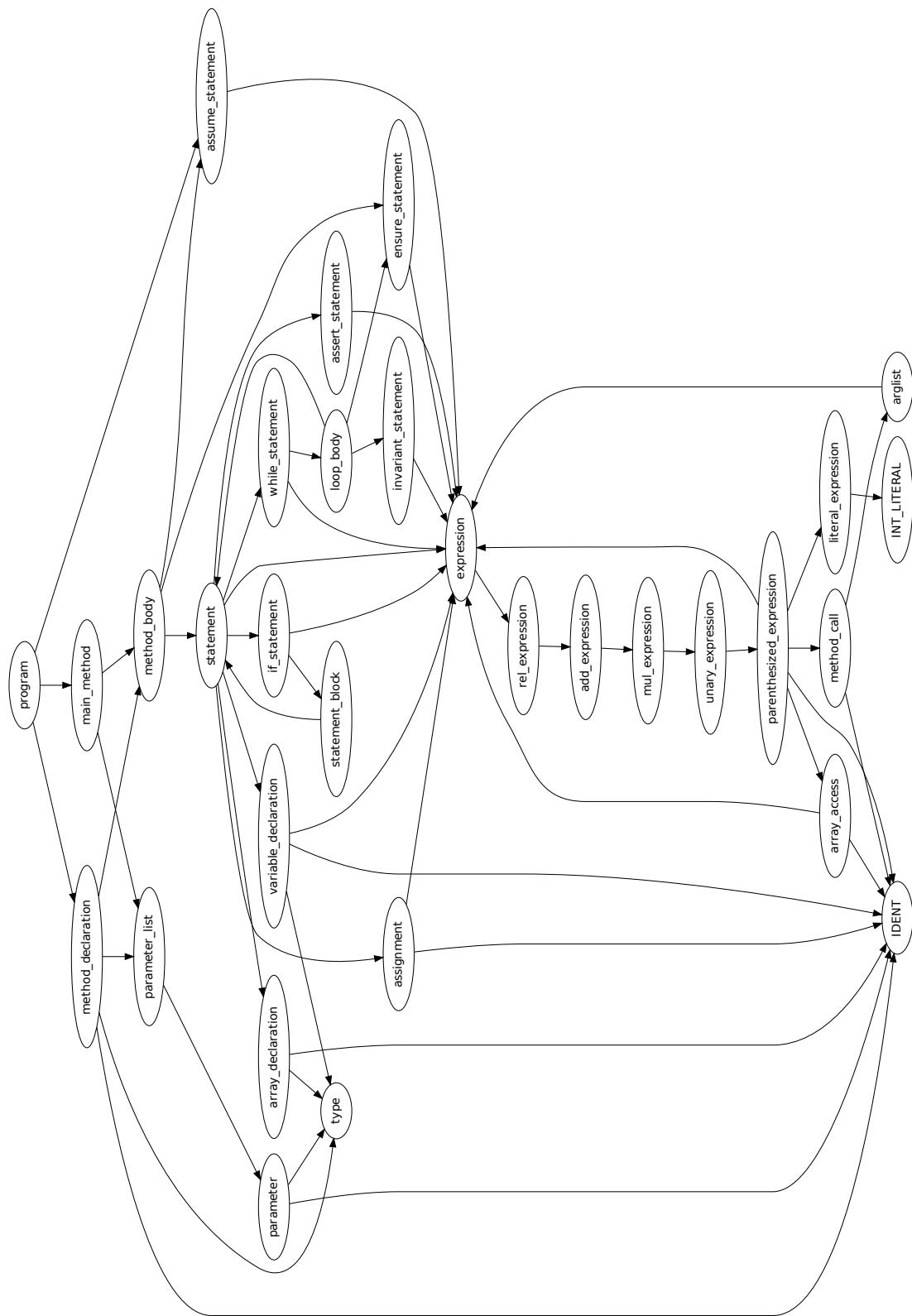
Die Schnittstelle zum Beweiser ist so ausgelegt, dass jeder Beweiser, der das SMT LIB 2.0 Format unterstützt, verwendet werden kann und für eine weitere Anbindungen nur wenige Änderungen nötig sind.

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 Beweisanbindung 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.

## 7 Struktur der While-Sprache



## 8 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_declarator
    | array_declarator
    | assignment
    | if_statement
    | while_statement
    | 'return' expression ';'
    ;

invariant_statement
    : 'invariant' expression ';'
    ;

assert_statement
    : 'assert' expression ';'
    ;

assume_statement
    : 'assume' expression ';'
    ;

ensure_statement
    : 'ensure' expression ';'
    ;

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

```

;

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

array_declaraction
: 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* '}'
;

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')*;

```