

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

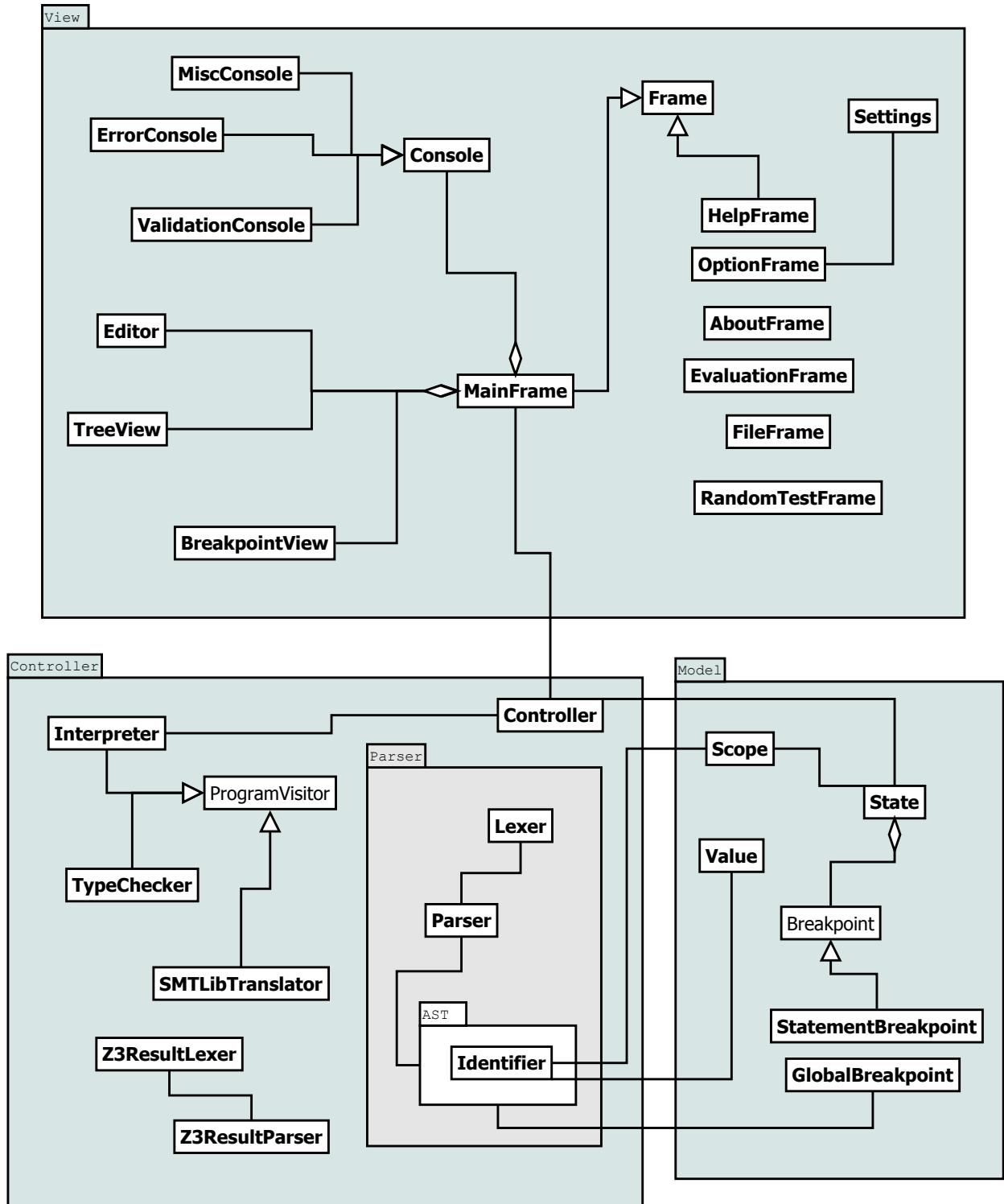
Inhaltsverzeichnis

1 Klassendiagramme	3
1.1 Übersicht	3
1.2 Feinstruktur der Komponenten	4
1.2.1 Programmstruktur	4
1.2.2 Besucherklassen	6
1.2.3 Parser	6
1.2.4 Modell	7
1.2.5 Benutzeroberfläche	7
2 Verhaltensdiagramme	8
2.1 Aktivitätsdiagramme	8
2.1.1 Parser/Type-Checker	8
2.1.2 Z3-Anbindung	9
2.2 Zustandsdiagramm	10
3 Syntax der While-Sprache	11
3.1 Übersicht der Schlüsselwörter und Sonderzeichen	11
3.2 Startsymbol	12
3.3 Produktionsregeln	12

1 Klassendiagramme

1.1 Übersicht

Dieses Klassendiagramm zeigt die Grobstruktur der Anwendung.



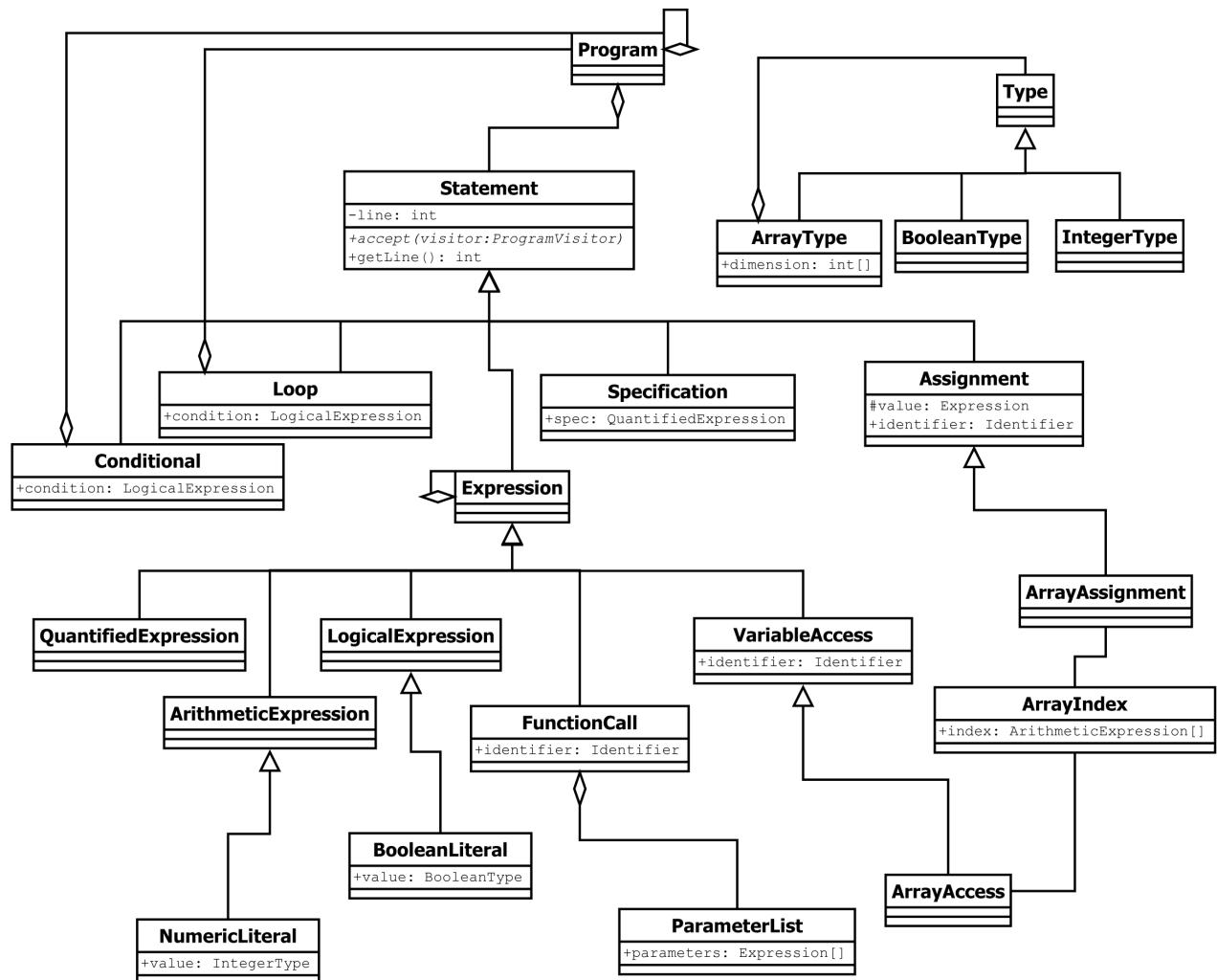
Die gesamte Architektur basiert auf dem Entwurfsmuster Model-View-Controller. Dies ermöglicht uns einen flexiblen Programmumentwurf, der eine spätere Änderung oder Erweiterung erleichtert und eine Wiederverwendbarkeit der einzelnen Komponenten ermöglicht.

- Das **Modell** enthält die darzustellenden Daten, z.B die Zustände der Variablen während der Programm ausführung und die vom Benutzer gesetzten Breakpoints
- 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 (Parse, Interpretieren, usw.) aus und bearbeitet die Eingaben des Benutzers

1.2 Feinstruktur der Komponenten

1.2.1 Programmstruktur

Dieses Klassendiagramm stellt mithilfe des Kompositum-Musters die Struktur des AST dar.



- **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.

Attribut:

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.

Attribute:

value

der Wert, der zugewiesen wird

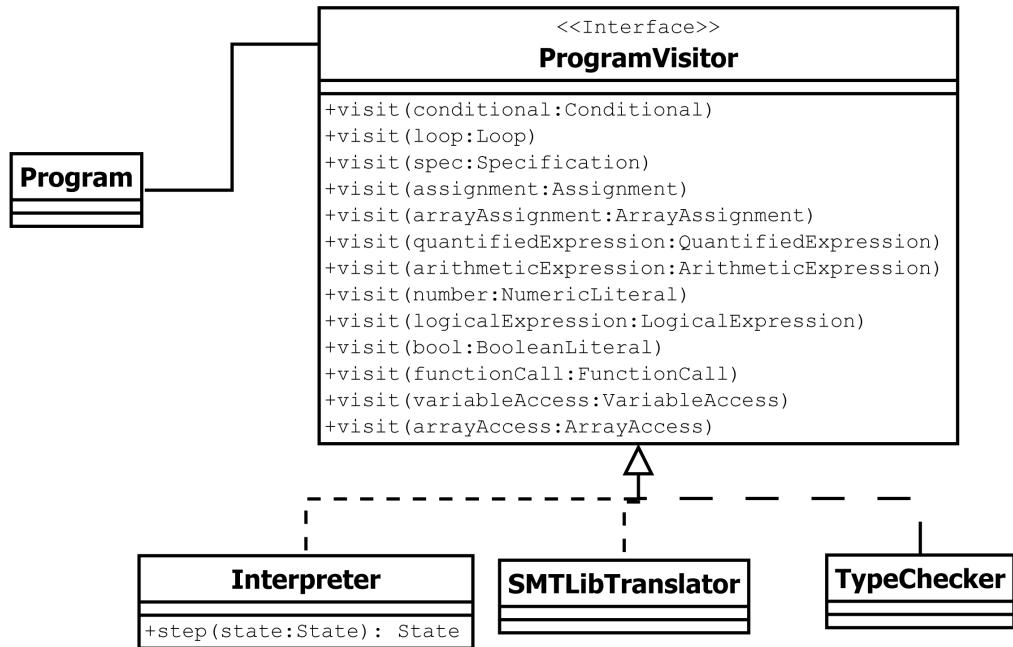
identifier

die Variable, der **value** zugewiesen wird

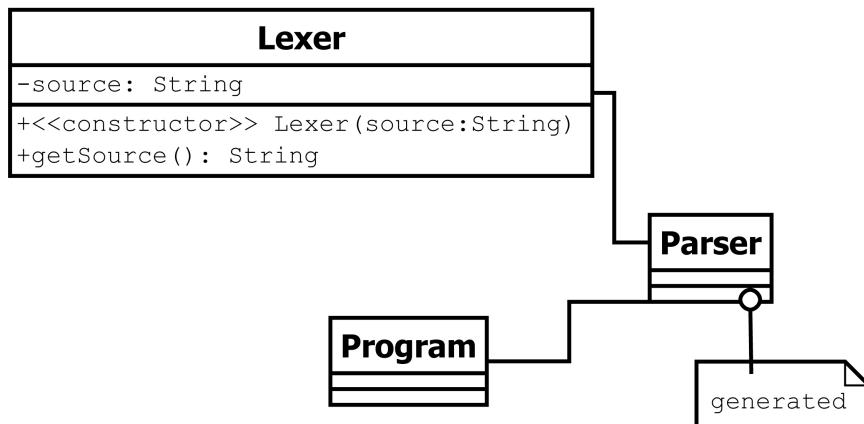
- **Expression**

Die abstrakte Klasse **Expression** steht für einen beliebigen Ausdruck. **QuantifiedExpression**, **ArithmeticalExpression**, **LogicalExpression**, **FunctionCall** und **VariableAccess** sind Unterklassen dieser Klasse.

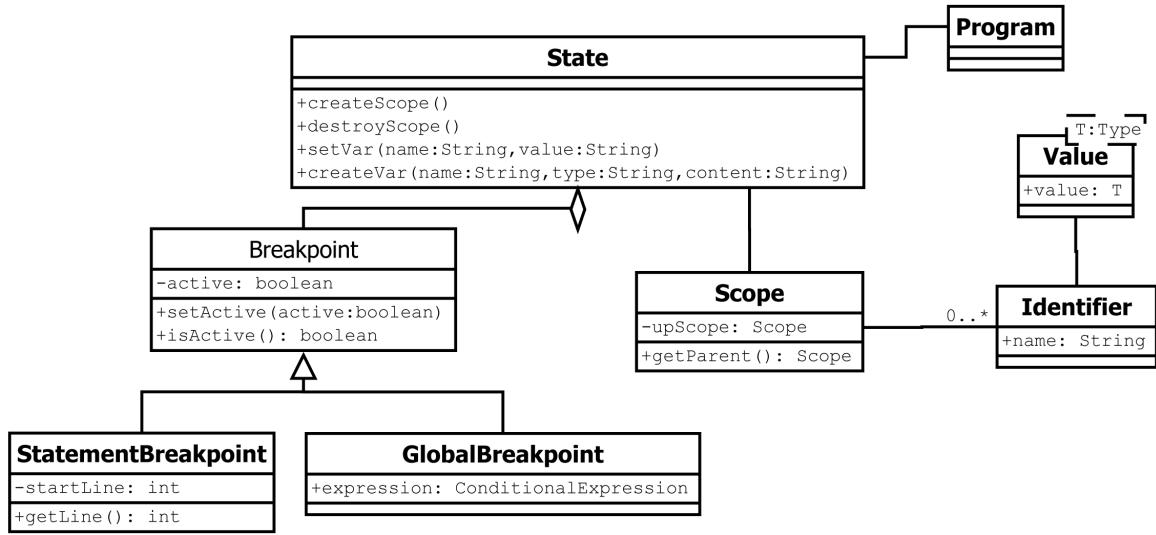
1.2.2 Besucherklassen



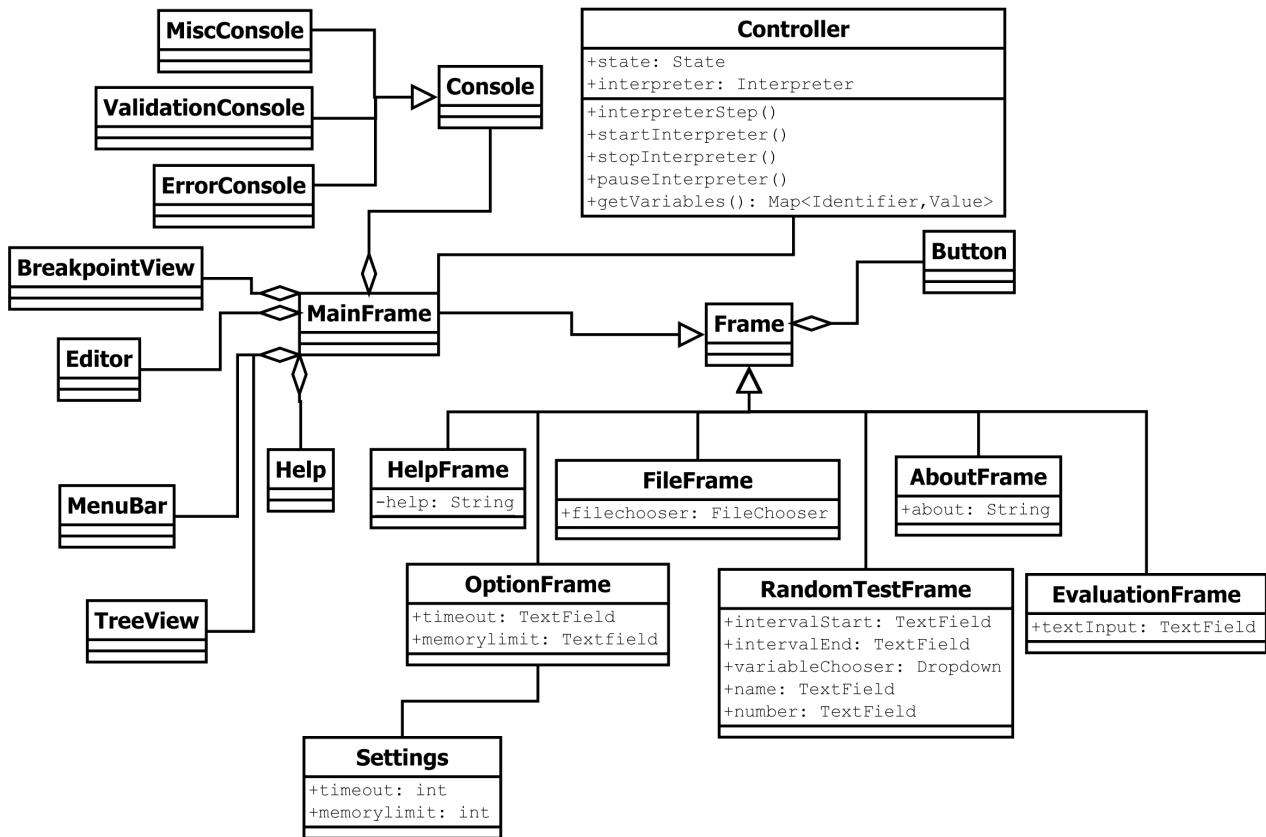
1.2.3 Parser



1.2.4 Modell



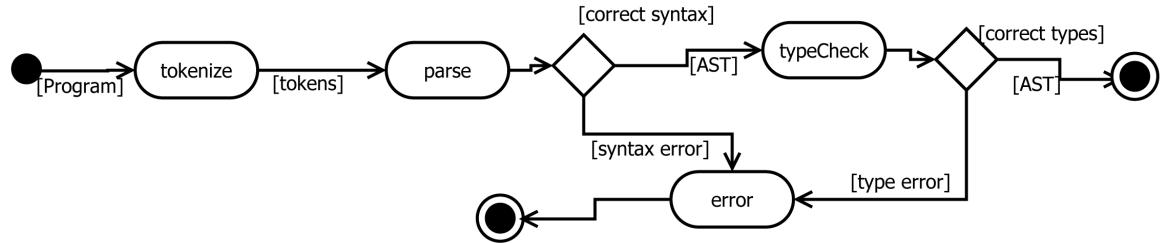
1.2.5 Benutzeroberfläche



2 Verhaltensdiagramme

2.1 Aktivitätsdiagramme

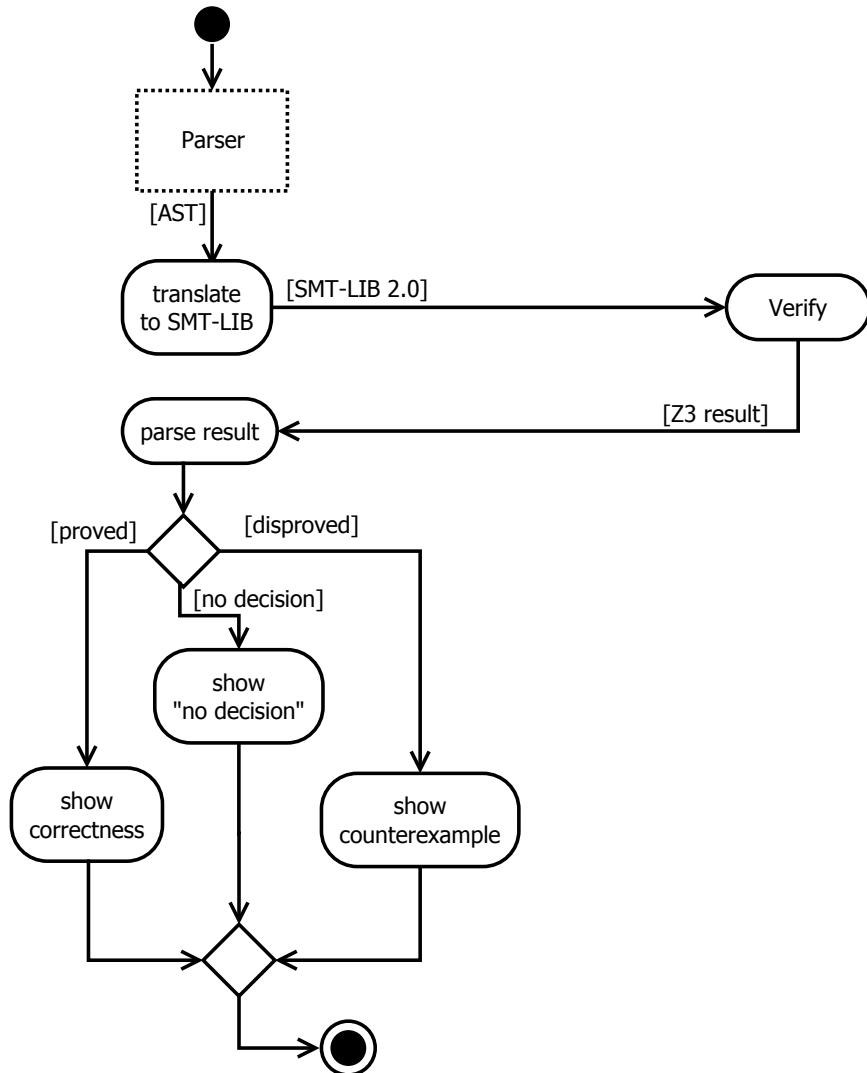
2.1.1 Parser/Type-Checker



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.

2.1.2 Z3-Anbindung



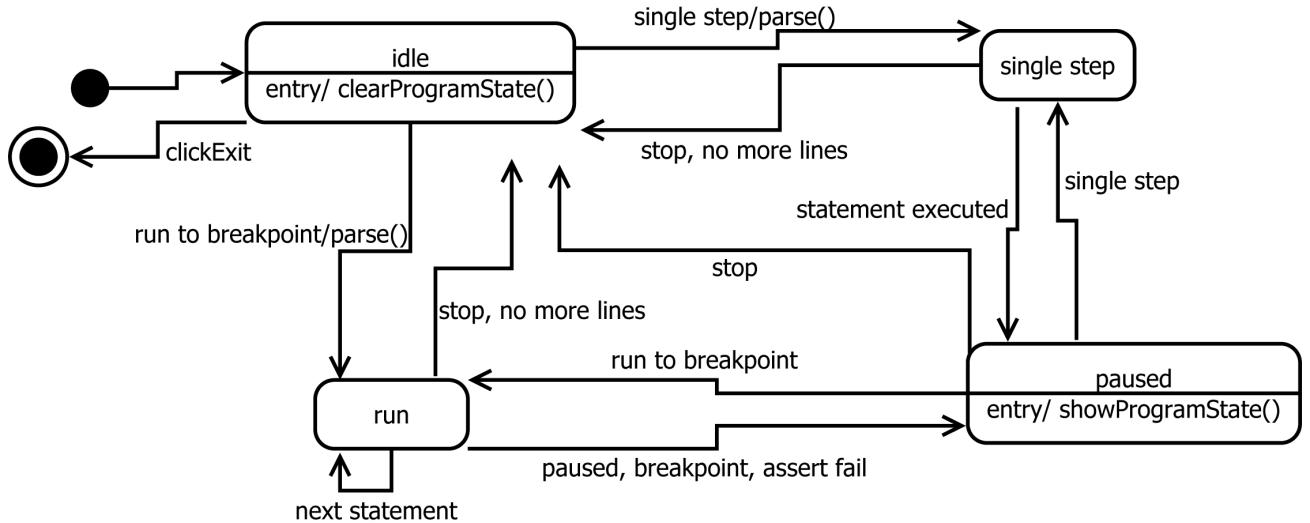
Zur Überprüfung der Korrektheit des Programms wird Z3 benutzt.

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.

2.2 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.

3 Syntax der While-Sprache

3.1 Übersicht der Schlüsselwörter und Sonderzeichen

boolean	→ type_specifier
else	→ if_statement
false	→ logical_literal
if	→ if_statement
int	→ type_specifier
return	→ statement
true	→ logical_literal
while	→ while_statement
0..9	→ integer_literal
a..z,A..Z,-	→ identifier
&	→ mul_expression
	→ add_expression
!	→ unary_expression
!=	→ expression
==	→ expression
<	→ rel_expression
<=	→ rel_expression
>	→ rel_expression
>=	→ rel_expression
+	→ add_expression
	→ unary_expression
-	→ add_expression
	→ unary_expression
*	→ mul_expression
/	→ mul_expression
%	→ mul_expression
,	→ arglist → parameter_list → variable_declaration → variable_initializer
;	→ statement
=	→ variable_declarator → statement
(→ bracket_expression → if_statement → while_statement → methode_call → methode_declaration
)	→ bracket_expression → if_statement → while_statement → methode_call → methode_declaration
[→ array_access → type
]	→ array_access → type
{	→ statement_block → variable_initializer
}	→ statement_block → variable_initializer
#	→ comment

3.2 Startsymbol

compilation_unit

3.3 Produktionsregeln

```
add_expression ::= mul_expression { ( "|" | "+" | "-" ) mul_expression }

arglist ::= expression { "," expression }

array_access ::= identifier "[" expression "]" { "[" expression "]" }

bracket_expression ::= "(" expression ")"
                     | method_call
                     | array_access
                     | identifier
                     | literal_expression

comment ::= "#" .* ( "\n" | "\r" )

compilation_unit ::= { field_declaraction }

expression ::= rel_expression { ( "==" | "!=" ) rel_expression }

field_declaraction ::= ( [ comment ] ( method_declaraction
                                         | statement ) )

identifier ::= "a..z,A..Z,_" { "a..z,A..Z,_,0..9" }

if_statement ::= "if" "(" expression ")" statement_block [ "else" statement_block ]

integer_literal ::= ( "0..9" { "0..9" } )

literal_expression ::= integer_literal
                     | logical_literal

logical_literal ::= "true" | "false"

method_call ::= identifier ( "(" [ arglist ] ")" )

method_declaraction ::= type identifier "(" [ parameter_list ] ")" ( statement_block )

mul_expression ::= unary_expression { ( "&" | "*" | "/" | "%" ) unary_expression }

parameter ::= type identifier

parameter_list ::= parameter { "," parameter }

rel_expression ::= add_expression [ ( "<" | "<=" | ">" | ">=" ) add_expression ]

statement ::= variable_declaraction ";"
            | identifier "=" variable_initializer ";""
            | ( expression ";" )
            | ( if_statement )
            | ( while_statement )
            | ( "return" [ expression ] ";" )
```

```

statement_block ::= "{" { statement } "}"

type ::= typeSpecifier { "[" "]" }

typeSpecifier ::= "boolean" | "int"

unary_expression ::= [ ( "!" | "+" | "-" ) ] bracket_expression

variable_declaration ::= type identifier { "," identifier } [ "=" variable_initializer ]

variable_initializer ::= expression
                      | ( "{"
                        [ variable_initializer { ","
                          variable_initializer } ]
                        "}" )

while_statement ::= "while" "(" expression ")" statement_block

```

