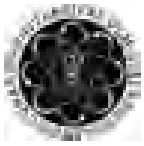


Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

Konzeption und Entwicklung eines Expression Verification Frameworks für ein objektzentriertes Prozessmanagementsystem

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Maik Schönfeld
maik.schoenfeld@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Sebastian Steinau

2017

Fassung 25. Januar 2018

Satz: PDF-LaTeX 2$_\varepsilon$

Kurzfassung

Prozessmanagementsysteme dienen dazu, betriebliche Prozesse dynamisch zu planen und auszuführen. Mit der objektzentrierten Sichtweise, welche sich über die Interaktion von Objekten definiert, sollen hier die Limitierungen der etablierten, aktivitätenzentrierten Prozessmanagementsysteme überwunden werden. Dazu werden Prozessdaten in Objekten zusammengefasst und verwaltet. Objekte bestehen aus Attributen und einem Lebenszyklus, welcher die Zustände eines Objekts beschreibt. Relationen zwischen Objekten definieren die Abhängigkeiten zwischen Objekten und deren Kardinalität. Interaktionen zwischen Objekten werden über einen Koordinierungsprozess gesteuert. PHILharmonicFlows ist eine Implementierung eines solchen objektzentrierten Prozessmanagementsystems.

Um die Koordination von Objektbeziehungen und den Lebenszyklus von Objekten dynamisch verwalten zu können, setzt PHILharmonicFlows intern auf ein Expressionframework, was die flexible Formulierung von ausdrucksstarken Bedingungen ermöglicht. Des Weiteren werden Expressions auch für die Rechteverwaltung verwendet.

Wenn fehlerhafte Expressions in das laufende System gelangen, kann dies zu schwer nachvollziehbaren Fehlersymptomen führen, was die Fehlersuche für den Modellierer erschwert und unter Umständen die Konsistenz des Systems gefährdet. Um zu verhindern, dass fehlerhafte Expressions zur Laufzeit Probleme verursachen können, ist es nötig, die erstellten Expressions auf Korrektheit zu verifizieren, bevor sie von der Modellierungsumgebung oder externen Schnittstellen in das System gelangen. Mit einer detaillierten Auswertung der Expressions soll es darüber hinaus auch möglich sein, dem Modellierer genauere Angaben bezüglich der Fehlerquelle zu machen.

Das Ziel der vorliegenden Arbeit ist die Konzeption und Entwicklung eines Expression Verification Frameworks für das objektzentrierte Prozessmanagementsystem PHILharmonicFlows. Zu diesem Zweck wurden die von PHILharmonicFlows verwendeten Expressions auf ihre Eigenschaften untersucht und mögliche Problemfälle isoliert und ausgewertet. Aus den gewonnenen Erkenntnissen wurden Anforderungen formuliert, welche als Basis für die Erstellung eines Entwurfs dienten. Auf Grundlage dieses Ent-

wurfs wurde schließlich das Expression Verification Framework vollständig implementiert und umgesetzt.

Da sich das PHILharmonicFlows-Framework noch in der Entwicklungsphase befindet, wurde neben einem effizienten und modularen Systemaufbau bei der Planung besonderer Wert auf die Erweiterbarkeit und Wartbarkeit des Expression Verification Frameworks gelegt.

Danksagung

Ich danke Sebastian Steinau für sein Engagement, für seine geduldige Art beim Vermitteln komplexer Zusammenhänge und seine konstante Unterstützung über den gesamten Verlauf der Entstehung dieser Arbeit.

Mein weiterer Dank gilt Saskia Langhammer für die vielen Stunden Korrekturlesen.

Inhaltsverzeichnis

1

Einleitung

Der Einsatz von Prozessmanagementsystemen ist heutzutage für viele Unternehmen ein Erfolgsfaktor, da sie es ermöglichen, Prozesse zu modellieren und an neue Herausforderungen anzupassen, ohne dass ein Software-Entwickler dafür Quellcode anpassen muss. In klassischen aktivitätenzentrierten Prozessmanagementsystemen werden dafür Prozesse als eine Komposition von Aktivitäten organisiert. Die zu den jeweiligen Aktivitäten gehörenden Daten, Funktionen und Prozesse werden im Allgemeinen aber unabhängig voneinander verwaltet, was es Benutzern unmöglich macht, auf benötigte Kontextinformationen während der Prozessausführung zuzugreifen [3].

Objektzentrierung ist ein junger Ansatz in der Prozessmanagementforschung, welcher dem datenzentrierten Zweig zuzuordnen ist. Damit sollen die Limitierungen der klassischen, aktivitätenzentrierten Prozessmanagementwerkzeuge, wie zum Beispiel die Starrheit der dort definierten Prozesse [16], aufgelockert werden. Zu diesem Zweck soll der objektzentrierte Ansatz dabei helfen, semi-strukturierte Prozesse zu erfassen und abzubilden [13]. Diese Prozesse zeichnen sich dadurch aus, dass sie hohe Flexibilität während der Prozessausführung benötigen, was aktivitätenzentrierten Prozessmanagementsystem aufgrund ihrer Limitierungen nicht leisten können.

Mit dem PHILharmonicFlows-Framework wird an der Universität Ulm ein objektzentriertes Prozessmanagement-Werkzeug entwickelt, welches die propagierten Vorteile dieses Ansatzes in der Praxis zeigen soll [7]. So ermöglicht PHILharmonicFlows beispielsweise die dynamische Generierung der Eingabemasken und Datanansichten für den Endnutzer [7].

Expressions sind ein wichtiger Baustein von PHILharmonicFlows. Sie kommen unter anderem im Autorisierungskonzept zum Einsatz, sie werden genutzt, um semantische

Beziehungen zwischen Objekten abzubilden, sind aber auch dazu geeignet, komplexe Berechnungen durchzuführen [17].

Von menschlichen Benutzern mit Modellierungswerkzeugen erstellt, können verschachtelte Expressions schnell komplexe Ausmaße annehmen, welche vom Benutzer nur noch schwer zu überschauen sind. Derzeit hat der Benutzer im Fehlerfall keine Unterstützung zur Modellierungszeit.

1.1 Problemstellung

Expressions können über unterschiedliche Wege ins System gelangen. Zu diesen Möglichkeiten zählen dabei Expressions, welche vom System selbst erstellt werden und keiner direkten Benutzerinteraktion zugrunde liegen. Des Weiteren schließt dies auch Expressions ein, welche über Modellierungswerkzeuge von Benutzern erstellt werden oder auch Expressions, welche über externe Systemschnittstellen angelegt werden.

Die unterschiedlichen Wege, Expressions im System anzulegen, bieten auch unterschiedlich gut ausgeprägte Möglichkeiten, die Korrektheit der Expressions sicherzustellen. So ist beispielsweise in der Modellierungsoberfläche von PHILharmonicsFlows eine bestimmte Struktur vorgegeben, was gewisse Probleme wie beispielsweise eine nicht initialisierte Expression von vornherein ausschließt. Externe Schnittstellen, welche von Fremdsystemen angesprochen werden können, haben solche Absicherungen nicht, was eine zentrale Verifikation im Server von PHILharmonicFlows erforderlich macht.

Werden Probleme mit einer Expression nicht während der Modellierungszeit erkannt, sondern erst zur Laufzeit abgefangen, kann dies für Benutzer zu schwer nachvollziehbaren Fehlerzuständen führen. Des Weiteren müssen in diesem Fall alle Systemkomponenten, welche Expressions verwenden, gegen ungültige Expressions abgesichert sein, was wiederum zu erhöhtem und unnötigem Wartungsaufwand führt.

1.2 Zielsetzung

Mit dem Entwurf und der Implementierung eines Verifikations-Frameworks für PHILharmonicsFlows soll ein Modul geschaffen werden, das die Gültigkeit einer Expression schon zur Modellierungszeit verifizieren kann. Konsumenten dieser Verifikation sollen dabei neben dem Resultat im Fehlerfall auch detaillierte Informationen zur Art des Problems und der verursachenden Teilexpressions bereitgestellt werden.

1.3 Struktur der Arbeit

Kapitel 2 stellt das PHILharmonicFlows-Framework vor und gibt eine kurze Einführung in das Thema Expressions. Kapitel 3 setzt sich mit den Eigenschaften von Expressions auseinander und stellt heraus, welche dieser Eigenschaften einer Verifizierung bedürfen. Das 4. Kapitel stellt das Konzept für die Lösung des Problems vor. Kapitel 5 beschreibt die Umsetzung des Entwurfs in Software. In Kapitel 6 werden alternative Ansätze für den Umgang mit Expressions gezeigt. Das 7. und letzte Kapitel fasst die gewonnenen Erkenntnisse zusammen und liefert einen Ausblick auf mögliche Weiterentwicklungen dieses Themas.

Umgang mit Code-Konstrukten

Das PHILharmonicFlows-Framework wird für .net mit C#[1] entwickelt. Um einen guten Lesefluss zu ermöglichen, werden verwendete Sprachelemente aus C# in Fußnoten erklärt. Sollten Sprachelemente in den Code-Ausschnitten eingeführt werden, wird die Fußnote an den Titel des Code-Ausschnitts angefügt.

[1] C# ist eine objektorientierte und imperative Programmiersprache von Microsoft® mit funktionalen und deklarativen Elementen.

2

Grundlagen

In diesem Kapitel wird das PHILharmonicFlows-Framework vorgestellt und anschließend auf Expressions eingegangen.

2.1 PHILharmonicFlows

PHILharmonicFlows[1] ist ein objektzentriertes Prozess-Management-Framework, welches im Gegensatz zu aktivitätenzentrierten Prozess-Management-Systemen die Zusammenhänge zwischen Prozessen, Daten, Funktionen und Benutzern berücksichtigt [6]. Dazu wird das Objekt in den Mittelpunkt der Betrachtung gestellt. Anwender modellieren Prozesse über Objekte und Objektbeziehungen, Daten werden ausschließlich als Attribute in den Objekten verwaltet. Beziehungen zwischen Objekten werden über Relationen abgebildet, welche zwei Objekttypen im Objektgraphen durch eine gerichtete Kante miteinander verbinden und die Kardinalität der Beziehung definieren. Relationen legen somit auch eine hierarchische Struktur der Objekttypen fest. Diese Art der Beziehung wird als syntaktische Beziehung bezeichnet und diese kann sogar transitiv definiert werden, sodass sich zwischen zwei in Beziehung stehenden Objekten andere Objekte befinden können. Dem gegenüber stehen semantische Beziehungen, welche die Koordinierungsbedingungen zwischen den Objekten definieren und in verschiedene Kategorien eingeteilt werden. Die Kategorien sind: Top-Down (der Fortschritt mehrerer untergeordneter Prozesse hängt vom Status eines übergeordneten Prozesses ab), Bottom-Up (der Fortschritt eines übergeordneten Prozesses hängt von mehreren untergeordneten Prozessen ab), Self (Beziehung zwischen zwei Objekten des gleichen

[1] PHILharmonicFlows = Process, Humans and Information Linkage for harmonic Business Flows

Typs), Transverse (mehrere Prozesse von zwei unterschiedlichen Objekttypen stehen über ein gemeinsames Bezugsobjekt miteinander in Beziehung) und Self-Transverse (ein Prozess hängt von mehreren Prozessen desselben Typs ab).

2.2 Expressions

Expressions bestehen aus einer fest definierten Anzahl von Operanden, einer Funktion und einem Rückgabewert. Die Operanden und der Rückgabewert sind von einem bestimmten Typ, vorgegeben durch die Funktion. Operanden können atomare Expressions sein, welche nur einen Wert vorhalten, wie zum Beispiel Konstanten oder Variablen. Alternativ können sie eine durch die Funktion vorgegebene Anzahl von Kindexpressions haben. Diese Anzahl an erwarteten Kindexpressions wird auch Arität genannt.

Da es möglich sein muss, Expressions in PHILharmonicFlows zu beliebig komplexen Ausdrücken zu kombinieren, werden diese in einer Baumstruktur angeordnet. Bei der Auswertung werden zunächst die Kindelemente ausgewertet und deren Resultate dann entsprechend der Funktion zu einem neuen Resultat verarbeitet. Diese Resultat-Werte können unterschiedliche Datentypen haben [5]. Aktuell sind im PHILharmonicFlows-Framework vier Typen implementiert: `Boolean` stellt Wahrheitswerte (Richtig, Falsch) dar, `String` ist eine Zeichenkette, `Number` bezeichnet eine Gleitkommazahl mit doppelter Genauigkeit und `Date` bildet ein Datum und/oder eine Uhrzeit ab. Darüberhinaus existieren für jeden dieser Datentypen noch Listentypen.

2.2.1 Beispiel

Als Beispiel soll hier eine Regel aus einem Bewerbungsprozess gezeigt werden, in welchem für jede eingehende Bewerbung auf eine ausgeschriebene Stelle Gutachten von Mitarbeitern zu erstellen sind. Die Regel besagt:

> *Eine Bewerbung darf nur abgelehnt werden, wenn mindestens die Hälfte der Gutachter die Bewerbung abgelehnt hat.*

Die Prüfung, ob diese Bedingung für eine Bewerbung erfüllt ist, kann mit einer Expression beschrieben werden. Dazu ist es zunächst nötig, die Bestandteile der Expression aus der Regel zu extrahieren. Um zu prüfen, ob mindestens die Hälfte der Gutachter die Bewerbung ablehnen, wird die Anzahl der negativen Gutachten sowie die Gesamt-Anzahl der Gutachten benötigt. Mit diesen Eigenschaften kann die Regel etwas formaler definiert werden:

$$\texttt{if} \quad (\#\text{negativer Gutachten}) \geq [(\#\text{Gutachten})/2] \\ \texttt{then} \quad [\text{Bewerbung ablehnen}] \quad \texttt{else} \quad [\text{weiter im Prozess}] \tag{2.1}$$

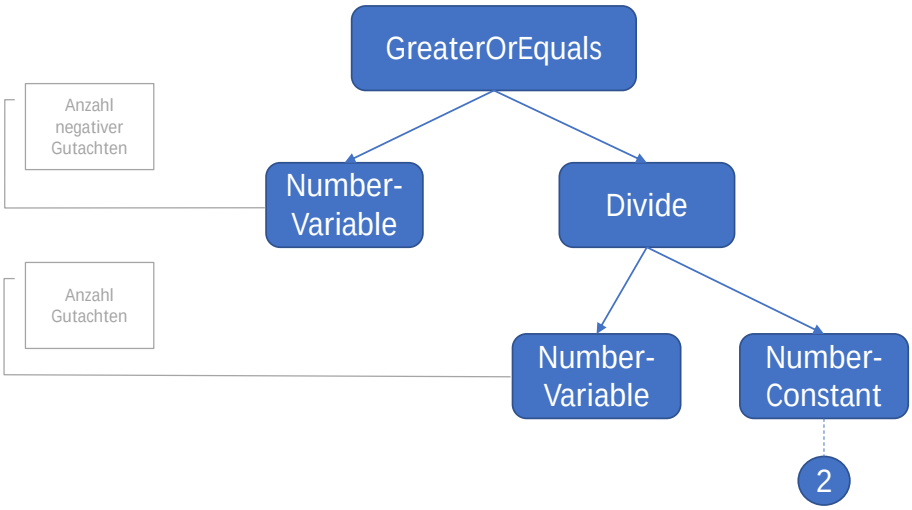


Abbildung 2.1: Expressionbaum für die Ablehnung von Bewerbung

Die eigentliche Expression ist hier:

$$(\#\text{negativer Gutachten}) \geq [(\#\text{Gutachten})/2] \tag{2.2}$$

Die Blätter eines Expressionbaumes sind dabei immer tatsächliche Werte, wie Variablen (im Beispiel die Anzahl der Gutachten und die Anzahl der negativen Gutachten) oder Konstanten (im Beispiel die Zahl 2). Teilexpressions sind dann die `Divide`-Funktion und an der Wurzel die `GreaterOrEquals`-Funktion. Abbildung 2.1 zeigt die Expression als Expressionbaum.

Obwohl besagter Expressionbaum die Hierarchie korrekt darstellt, ist er dennoch nur bedingt geeignet, die Expression in ihrer Gesamtheit zu beschreiben.

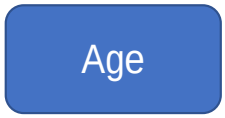


Abbildung 2.2: Age-Expression

Um dies zu verdeutlichen, wird in Abbildung 2.2 die Age-Expression gezeigt. Allein mit dieser Darstellung wird nicht deutlich, wie viele Parameter die Expression erwartet, welche Datentypen diese Parameter besitzen und welchen Rückgabetyp die Expression hat oder gar, welche Abhängigkeiten zwischen Parametern untereinander beziehungsweise mit dem Rückgabetypen bestehen. Daher wird für diese Arbeit eine erweiterte Darstellung verwendet, welche die zuvor genannten Eigenschaften darstellen kann.

2.2.2 Erweiterte Darstellung von Expressions und Expressionbäumen

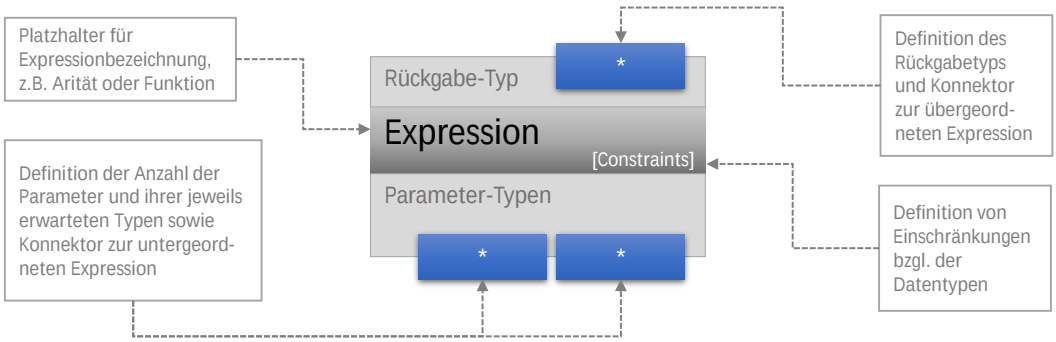


Abbildung 2.3: Eigenschaften der Expressiondarstellung

Abbildung 2.3 zeigt die Definition einer solchen erweiterten Expressiondarstellung. Sie besitzt blaue Konnektoren für Parameter und den Rückgabewert, über welche Expressions miteinander verbunden werden können. Sie ist in der Lage, neben der Bezeichnung

der Expression sowohl die Anzahl der erwarteten Parameter (siehe Abbildung C.1), als auch die Datentypen für Parameter und den Rückgabetypen zu definieren.

Darüber hinaus können mit dieser Darstellung auch Typeinschränkungen für Parameter definiert werden. Dazu wird der betreffende Parameter nicht direkt mit einem Datentyp beschriftet, sondern erhält eine Zahl, zu welcher dann Einschränkungen im Constraint-Feld festgelegt werden. Abhängigkeiten zwischen den Parametern beziehungsweise zwischen einem Parameter und dem Rückgabetypen können so ebenso definiert werden. Dazu erhalten die abhängigen Konnektoren eine Bezifferung anstelle eines konkreten Datentyps, wobei von einander abhängige Parameter jeweils die gleiche Ziffer besitzen.

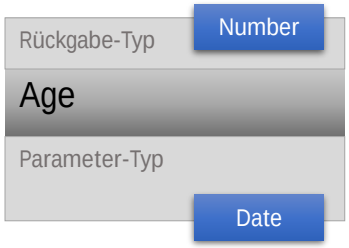


Abbildung 2.4: Age-Expression in Expressiondarstellung

In Abbildung 2.4 wird die zuvor erwähnte `Age`-Expression in der erweiterten Darstellung gezeigt. So ist erkenntlich, dass die `Age`-Expression einen `Date`-Wert erwartet und einen `Number`-Wert zurückliefert. Um eine gültige Expression zu bilden, muss die `Age`-Expression eine Kindexpression besitzen, welche den erforderlichen `Date`-Wert bereitstellt. Wie in dem in Abbildung 2.5 gezeigt wird, könnte die Age-Expression in Kombination mit einer Datumsvariable, die beispielsweise auf das Geburtsdatum einer Person verweist, eine gültige Gesamtexpression ergeben.

Mit dieser Darstellung kann nun die Expression aus dem Unterkapitel 2.2.1 mit allen Eigenschaften dargestellt werden, siehe Abbildung 2.6. In dieser Darstellung ist genau ersichtlich, welche Datentypen für die Parameter der beteiligten Expressions erwartet werden sowie welche Typen von den Expressions zurückgeben werden.

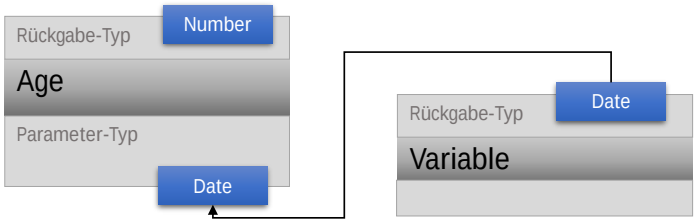


Abbildung 2.5: Beispiel für eine Age-Expression mit Kindelement

2.2.3 Eigenschaften von Expressions

Expressions werden im PHILharmonicFlows-Framework unter dem Namensraum[2] `Shared.Data.Expressions` verwaltet. Alle Expression-Klassen leiten sich von der abstrakten[3] Basisklasse `Expression` ab (siehe Abbildung 2.7). `Expression` hat die Property[4] `ExpressionFunction`, welche die Funktion der Expression beschreibt.

```
1 [DataContract]
2 public enum ExpressionFunction
3 {
4 //...
5 [EnumMember]
6 [Display(Name = "Constant<Bool>")]
7 [Arity(0)]
8 BooleanConstant,
9 //...
```

Listing 2.1: `ExpressionFunction` (Auszug)

[2] In C# werden Namensräume verwendet, um Code zu strukturieren [8]. Mit ihnen können beispielsweise in unterschiedlichen Namensräumen gleiche Bezeichner verwendet werden.
[3] mit dem **`abstract`**-Keyword versehene Klassen können nicht direkt instanziiert werden. Sie dienen als Basis für abgeleitete Klassen.
[4] In C# kapseln Properties Werte einer Klasse für öffentliche Lese- und Schreibzugriffe, genannt Getter- und Setter-Methoden.

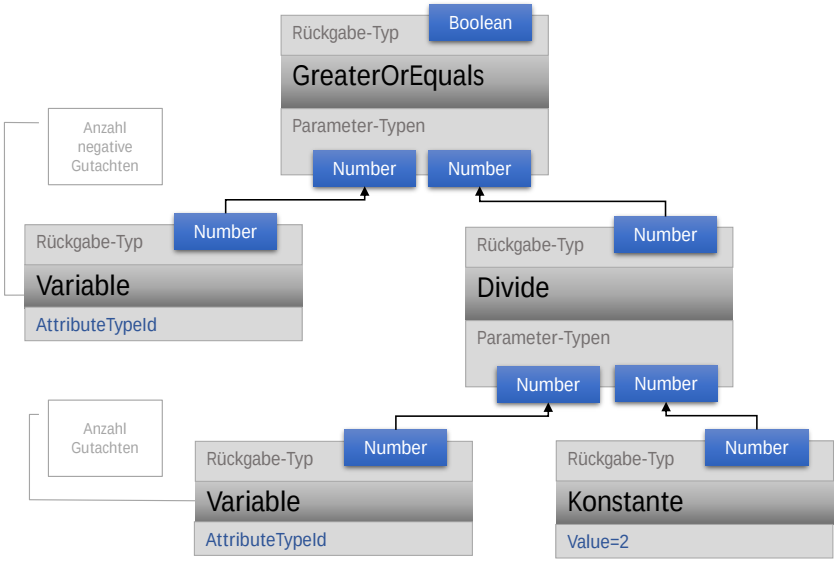


Abbildung 2.6: Expressionbeispiel aus dem Bewerbungsprozess mit allen Eigenschaften

`ExpressionFunction` ist als Aufzählungstyp[5] definiert und stellt darüber hinaus Metainformationen in Form von Attributen[6] über die jeweilige Funktion bereit. Hier relevant ist vor Allem die Arität, welche die Anzahl der Kind-Expressions spezifiziert, die die Expression benötigt, um sinnvoll ausgewertet zu werden. Im Beispiellisting 2.1 hat die Funktion `BooleanConstant` mit `Arity(0)` die Arität null. Eine besondere `ExpressionFunction` ist `None`. Sie wird genutzt, um eine ungültige Expression zu beschreiben.

Von `Expression` abgeleitet werden die Klassen `NullaryExpression`, `UnaryExpression`, `BinaryExpression` und `TernaryExpression`. Diese Klassen stellen Expressions mit unterschiedlicher Arität dar. Die `NullaryExpression` hat die Arität null und ist das Blatt in einem Expressionbaum, sie kann also keine weiteren Kindelemente mehr besitzen.

Die `NullaryExpression` kann eine von drei verschiedenen Funktionen übernehmen:

[5] Ein **`enum`** oder auch **`enumeration`** definiert eine Aufzählung mit einer endlichen Wertemenge.

[6] Attribute in C# werden genutzt, um Code deklarativ mit Metainformationen zu versehen [19]. Siehe Listing 2.1 Zeile 3, 7 und 9. Dabei wird unterschieden zwischen Attributen, die durch das .net-Framework bereitgestellt werden, wie zum Beispiel [`DataContract`], und Custom-Attributen, welche vom Entwickler selbst erstellt werden, wie hier das [`Arity`]-Attribut.

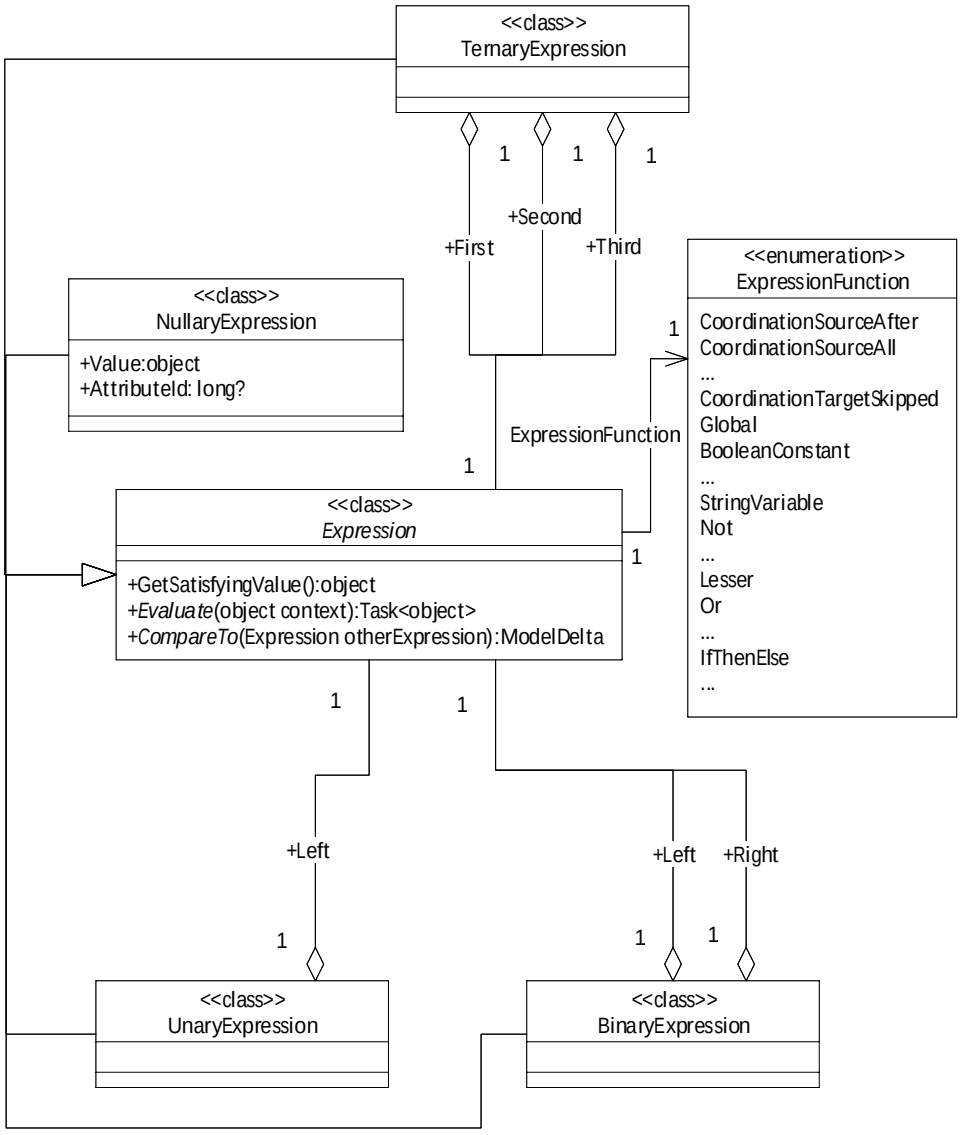


Abbildung 2.7: Klassendiagramm Expressions

- Als **Konstante** speichert die `NullaryExpression` einen direkten Wert, welcher zur Modellierungszeit definiert wird und zur Laufzeit nicht mehr verändert werden kann. Der Wert wird in der Eigenschaft `Value` vom Typ **`object`** gespeichert. Somit kann die Expression beliebige Typen aufnehmen. Anhand der `ExpressionFunction`

wird dann während der Evaluation ein **`Cast`**[7] in den erwarteten Typ vorgenommen und der Wert verarbeitet.

- Als **Variable** oder Referenztyp wird die `NullaryExpression` zur Modellierungszeit auf einen Datentyp festgelegt, der Wert bestimmt sich aber erst zur Evaluationszeit und kann sich zur Laufzeit beliebig ändern. Dazu wird die `AttributeId` auf die ID eines Attributs gesetzt. Zur Evaluationszeit wird dann der Wert des Attributs aus dem Kontext des laufenden Prozesses ausgelesen.
- Als **Koordinationselement** verwaltet die Expression Beziehungen zwischen den Objekten in PHILharmonicFlows. Für diese Expressions ist keine spezifische Verifikation erforderlich.

2.3 Verifikation

Das Ziel der Verifikation von Expressions ist das Erkennen von Fehlern zur Modellierungszeit. Auf diese Weise wird es dem Modellierer ermöglicht, eventuelle Fehler in Expressions direkt in der Modelierungsumgebung zu erkennen. Abgrenzend dazu würde eine Validierung mit einer Expression zur Evaluationszeit stattfinden. Dabei könnte dann zum Beispiel geprüft werden, ob Variablen einen gültigen Wert haben.

Da das PHILharmonicFlows-Framework externe Schnittstellen anbietet, über welche auch neue Objekte importiert werden können, die wiederum Expressions enthalten können, muss hier zwingend eine Verifikation stattfinden, um so einen korrekten internen Zustand des PHILharmonicFlows-Frameworks sicherstellen zu können [2].

[7] Bei einem Cast wird ein Objekt innerhalb der Vererbungslinie seiner Klasse in einen allgemeineren oder spezielleren Typ umgewandelt.

3

Analyse

Um zu ermitteln, welche Eigenschaften der Expressions verifiziert werden müssen, werden zwei Ansätze verfolgt. Zum einen werden Eigenschaften der Expressions selbst untersucht. Zum anderen werden die Eigenschaften der Implementierung von Expressions in PHILharmonicFlows daraufhin untersucht, an welchen Stellen eine Verifikation nötig ist, um fehlerhafte Expressions erkennen zu können.

3.1 Problemquellen

In Kapitel 2 wurden die grundlegenden Eigenschaften der Expressions betrachtet. Zusätzlich dazu bringt eine Implementierung eines Konzeptes immer neue Aspekte mit, die es bei einer Verifikation zu beachten gilt, geschuldet der Art der Implementierung sowie den verwendeten Sprachfeatures selbst.

```
1 public abstract class Expression
2 {
3         public ExpressionFunction ExpressionFunction { get; set; } =
              ExpressionFunction.Equals;
4         //...
5 }
```

Listing 3.1: `Expression` (Auszug)

Wie bereits im Abschnitt 2.2 gezeigt, leiten sich alle Expression-Klassen von der abstrakten Basisklasse `Expression` ab, siehe Listing 3.1.

Alle Expression-Klassen benutzen die von `Expression` geerbte `ExpressionFunction`-Property.

3.1.1 Strukturelle Problemquellen

Um eine grundlegende Auswertung einer Expression ermöglichen zu können, muss die Expression die erwartete Struktur haben. Im Folgenden werden mögliche Verletzungen dieser Struktur aufgeführt.

Falsche Arität

Die `ExpressionFunction` wird bereits in der Basis-Klasse definiert und kann damit von abgeleiteten Klassen nicht mehr eingeschränkt werden. Die abgeleiteten Klassen müssen sich also darauf verlassen, dass sie ausschließlich mit Funktionen erstellt werden, die ihrer Arität entsprechen. Technisch möglich wäre aber auch eine semantisch falsche Objekterstellung wie hier gezeigt in Listing 3.2 und Abbildung 3.1.

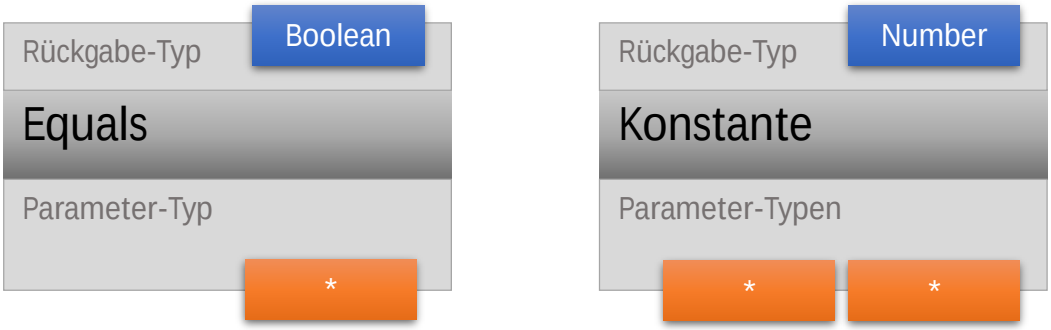


Abbildung 3.1: Expressions mit fehlerhafter Arität

Die `Equals`-Funktion vergleicht zwei übergebene Werte auf Gleichheit. Im ersten hier aufgeführten Beispiel wurde sie aber in einer Expression mit nur einem Parameter deklariert, der `UnaryExpression`. Das zweite Beispiel hat ein ähnliches Problem. Eine Zahlenkonstante wurde hier in einer `BinaryExpression` deklariert. Die Evaluierung

solcher Expressions wird fehlschlagen oder im schlimmsten Fall ein falsches bzw. unerwartetes Ergebnis erzeugen. Auch kann die Stabilität des PHILharmonicFlows-Systems beeinflusst werden.

```
1 var equalsExpression = new UnaryExpression(ExpressionFunction.Equals);
2 var numberConstant = new BinaryExpression{ExpressionFunction =
     ExpressionFunction.NumberConstant};
```

Listing 3.2: Expression mit falscher Arität[1]

Aufrufender Code würde beispielsweise für die zweite Expression aus dem Listing 3.2 ein Objekt der Basisklasse `Expression` erwarten und diese dann anhand ihrer `ExpressionFunction` in die passende Ableitung casten, hier in die `NullaryExpression`. Da aber hinter der Basisklasse eine `BinaryExpression` steckt, wird dieser Cast fehlschlagen, was wiederum eine `NullReferenceException` auslösen wird.

None-Expression

Die `None`-Funktion zeigt an, dass eine Expression nicht richtig konfiguriert wurde oder anderweitig fehlerhaft ist. Ein Fall wie die Konstellation in Listing 3.3 kann daher nicht sinnvoll evaluiert werden.

```
1 var nullaryExpression = new UnaryExpression(ExpressionFunction.None);
```

Listing 3.3: Expression mit ungültiger `ExpressionFunction`

Die `None`-Funktion hat die besondere Arität -1 und sollte im regulären Betrieb nicht auftreten. Da zumindest technisch die Möglichkeit besteht, dass ebendies doch geschieht, muss eine Verifizierung diese Funktion erkennen und als Fehler melden.

Null-Referenz

Wenn eine Expression oder Teilexpression nicht initialisiert ist, kann dies im schlimmsten Fall eine `NullReferenceException` auslösen.

[1] Das **`var`**-Schlüsselwort ersetzt die qualifizierte Angabe der Klasse bei Erhalt der Typsicherheit. Dies ermöglicht es dem Compiler, den Typ für Optimierungszwecke durch andere Typen zu ersetzen.

```
var not = new UnaryExpression(ExpressionFunction.Not)
{
//simplified creation of a boolean-variable expression
Left = new NullaryExpression(ExpressionFunction.BooleanVariable)
};
```

Listing 3.4: Expression mit Kindelement

In Listing 3.4 und Abbildung 3.2 wird die korrekte Erstellung einer Expression mit der Funktion `Not` gezeigt, die eine Expression mit einem $Boolean$-Rückgabewert erwartet und diesen Wert negiert.

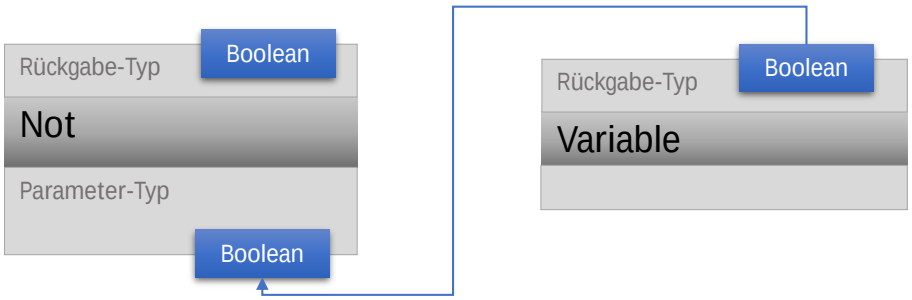


Abbildung 3.2: korrekt initialisierte Expression

Die Kindexpression wird über Objektinitialisierung[2] übergeben. Die direkte Belegung der Kindexpression ist allerdings optional und somit ist auch der Aufruf in Listing 3.5 beziehungsweise Abbildung 3.3 möglich.

```
var not = new UnaryExpression(ExpressionFunction.Not);
```

Listing 3.5: Expression ohne Kindelement

[2]Mit Objektinitialisierung können in C# Eigenschaften eines Objekts direkt bei der Objekterstellung übergeben werden, auch wenn kein Parameter im Konstruktor für sie vorgesehen ist [19]. Dafür wird am Ende des Konstruktoraufrufes ein Block mithilfe einer geschwungenen Klammer geöffnet, in welchen dann die Eigenschaften qualifiziert gesetzt werden können.

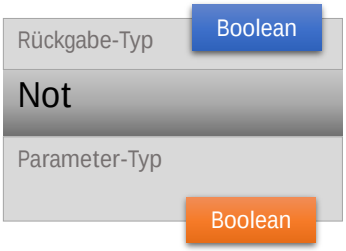


Abbildung 3.3: Expressions ohne Kindexpression

Code, welcher eine solche Expression auswertet, würde erwarten, dass eine `UnaryExpression` eine initialisierte Referenz auf eine Kind-Expression enthält und im schlimmsten Fall ungeprüft darauf zugreifen, was zu einer `NullReferenceException` führen würde. Das gleiche Resultat würde auch eine nicht initialisierte Expression, zu sehen in Listing 3.6, erzeugen.

```
1 UnaryExpression not = null;
```

Listing 3.6: Nicht initialisierte Expression

Eine Unterbrechung im Expressionbaum bei Erhalt des abgeschnittenen Teilbaums, gezeigt in Abbildung 3.4, erzeugt das gleiche Problem. Für die Auswertung der Expression spielt es keine Rolle, ob ein abgeschnittener Teilbaum existiert. Der abgeschnittene Teilbaum würde ignoriert und zu einem späteren Zeitpunkt vom Garbage-Collector[3] gelöscht werden.

Zyklischer Graph

Problematisch sind auch Expressions, deren Kind-Expressions auf das Elternobjekt verweisen, gezeigt in Listing 3.7. Bei der Auswertung einer solchen Expression würde zur Laufzeit eine `StackOverflowException` auftreten, da mehr rekursive Aufrufe stattfinden

[3] Der Garbage-Collector ist ein Konzept von modernen objektorientierten Programmiersprachen, wie Java und C#, um verwaiste Objekte (also Objekte, die in der Laufzeitumgebung nicht mehr referenziert werden) zu löschen und den von ihnen verwendeten Speicher freizugeben [10].

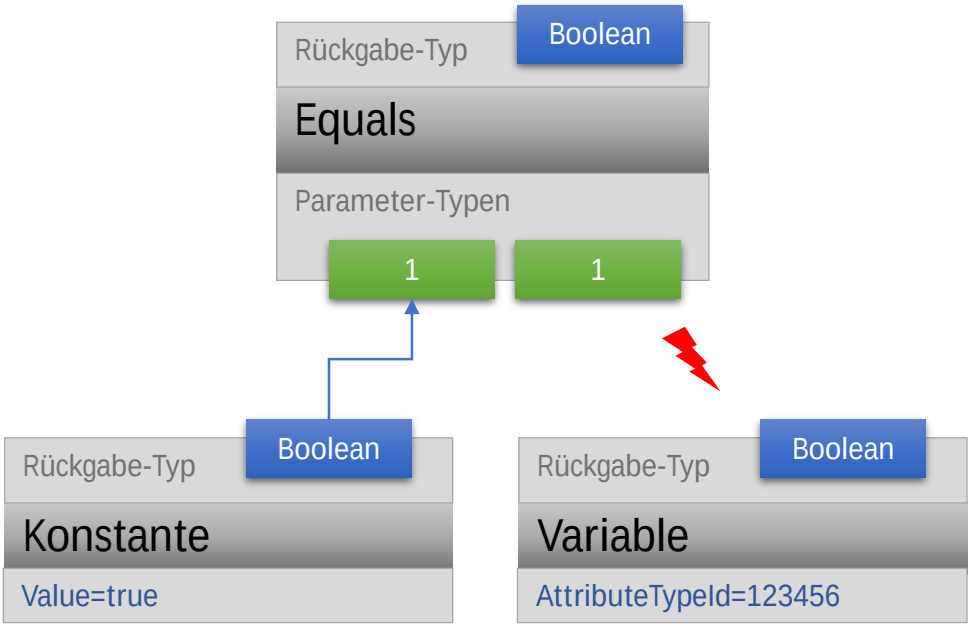


Abbildung 3.4: Expression mit abgeschnittenem Teilbaum

als die Laufzeitumgebung verarbeiten kann. Die Zyklen können direkt oder indirekt auftreten, gezeigt in Abbildung 3.5.

```
1 var not = new UnaryExpression(ExpressionFunction.Not);
2 not.Left = not;
```

Listing 3.7: Rekursive Referenzierung

3.1.2 Semantische Problemquellen

Die Überprüfung semantischer Zusammenhänge erfordert, dass die betreffende Expression strukturell korrekt ist. Dies wird in den folgenden Betrachtungen als gegeben vorausgesetzt.

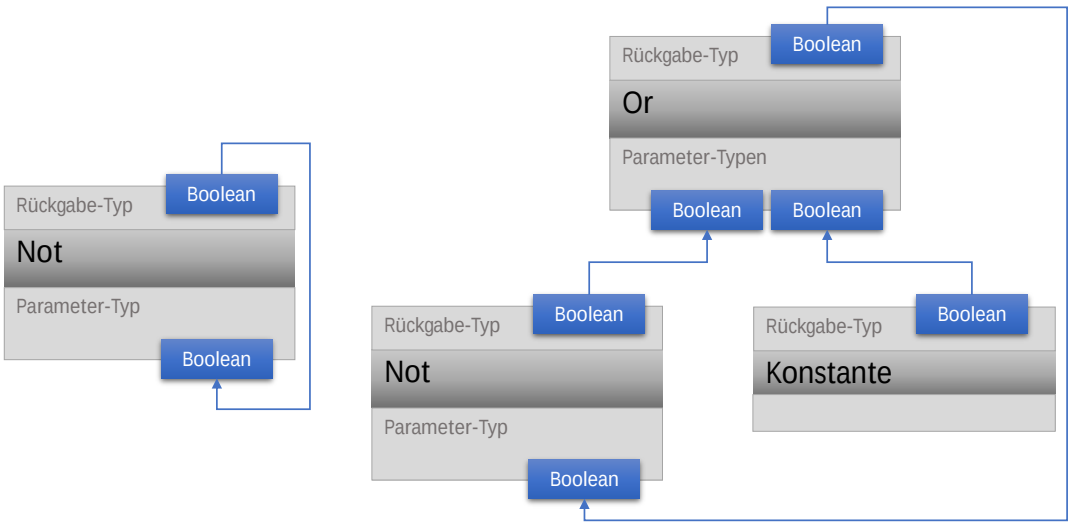


Abbildung 3.5: Expressions mit Zirkelverweisen

Variable-Expressions

Es existiert für jeden Datentyp eine `ExpressionFunction`, die eine Variable für diesen Datentyp bereitstellt: `BooleanVariable`, `DateVariable`, `NumberVariable`, `StringVariable`.

Variablen enthalten ihren Wert nicht selbst, sondern verweisen über ihre Property `AttributeTypeId` auf ein Attribut in einem übergeordneten PHILharmonicFlows-Framework-Objekt (siehe Listing 3.8 beziehungsweise Abbildung 3.6).

Abbildung 3.6: Expressions mit Variablenfunktion

Da diese Property eine `Nullable`-Property[4] ist, muss sichergestellt werden, dass diese Property ungleich **`null`** ist.

```
var numberVarExpression = new
    NullaryExpression(ExpressionFunction.NumberVariable)
{
   AttributeTypeId = 1234567890L
};
```

Listing 3.8: `NumberVariable`-Expression

Konstanten-Expressions

Ebenso wie bei den Variablen gibt es auch Konstanten für jeden Datentyp (siehe Listing 2.1). Konstanten beinhalten ihren Wert bereits zur Modellierungszeit und speichern ihn in der `Value`-Property. Diese ist vom Typ **`object`** und der erwartete Typ hängt von der definierten `ExpressionFunction` ab.

```
var booleanConstantExpression = new
    NullaryExpression(ExpressionFunction.BooleanConstant)
{
   Value = "test"
};
```

Listing 3.9: `NumberVariable`-Expression

In Listing 3.9 wird einer `BooleanConstant`-Expression ein **`string`**-Wert zugewiesen.

Zur Kompilierungszeit ist dies noch kein Problem, da die `Value`-Property durch ihre allgemeine Definition jeden Datentyp speichern kann. Wenn die Expression aber zur Laufzeit ausgewertet werden soll, kann eine solche fehlerhafte Expression eine `InvalidCastException` auslösen.

[4] `Nullable`-Properties erlauben es, primitive Datentypen wie **`int`** mit **`null`** zu belegen. Dies kann nützlich sein, wenn beispielsweise Fälle existieren, für die keine sinnvolle Belegung notwendig oder möglich ist. `Nullable`-Properties erfordern einen **`null`**-Check, bevor man ohne Gefahr, eine `NullReferenceException` auszulösen, auf sie zugreifen kann.

Abbildung 3.7 zeigt neben einer korrekten Definition auch zwei ähnliche Fälle desselben Problems.

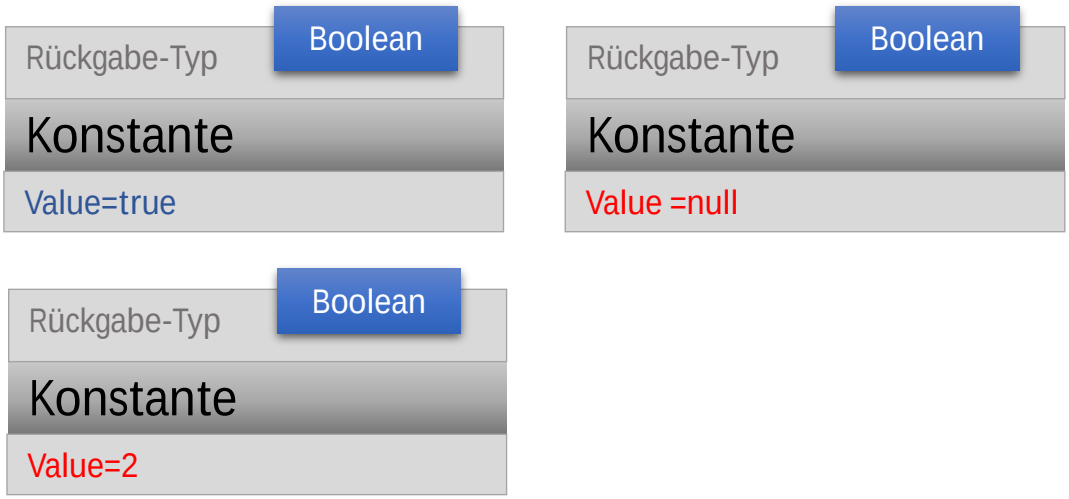


Abbildung 3.7: Fehlerhafte Expressions mit Konstantenfunktion

Parameter-Typ-Einschränkungen

Im Abschnitt 2.2.2 wurde festgestellt, dass die Funktionen der `ExpressionFunction`-Enumeration impliziten Limitierungen bezüglich der Datentypen ihrer Operanden unterliegen. Die im genannten Abschnitt beschriebene Divide-Expression akzeptiert beispielsweise ausschließlich Expressions mit dem Rückgabetyp $Number$ als Operanden.

```
1 var ageOfJanuaryTheFirst = new UnaryExpression(ExpressionFunction.Age)
2 {
3    Left = new NullaryExpression(ExpressionFunction.BooleanConstant) {
         Value = true }
4 };
```

Listing 3.10: Verletzung der inhärenten Parameter-Typ-Einschränkung

Ein anderes Beispiel ist die in Listing 3.10 und Abbildung 3.8 dargestellte `Age`-Expression. Diese erwartet eine Expression mit einem $Date$-Rückgabewert und berechnet dann die Anzahl der vollen Jahre, die seit dem angegebenen Datum bis zum aktuellen Tag vergangen sind, mit anderen Worten also das Alter einer Person.

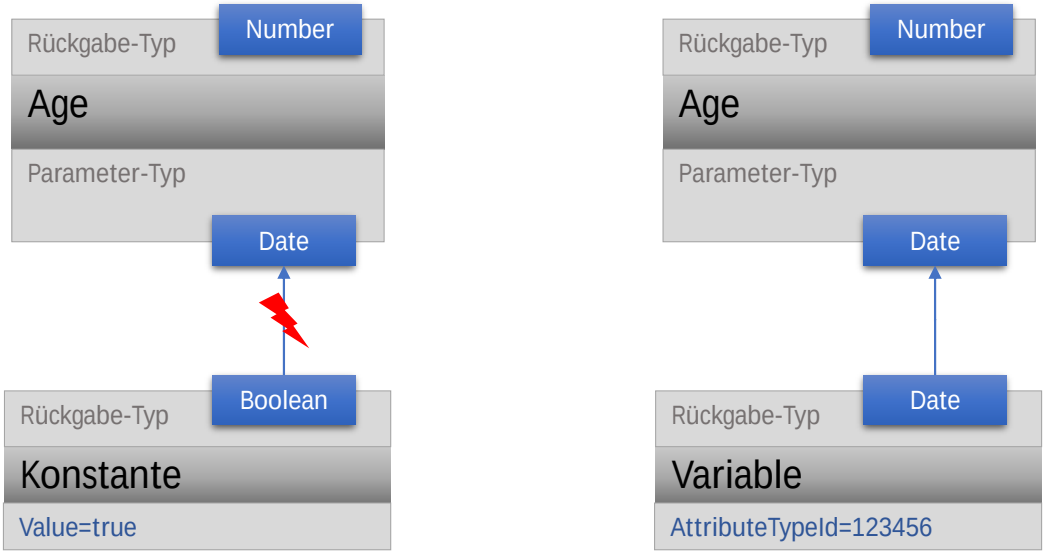


Abbildung 3.8: Verletzung der inhärenten Parameter-Typ-Einschränkung

Im Beispiel wurde versehentlich eine `BooleanConstant`-Expression übergeben, was technisch möglich ist, die `Age`-Expression allerdings ungültig macht.

Abhängige Parameter

Für einige Expressions sind direkte Typeinschränkungen nicht angebracht. Die im Abschnitt 2.2.2 verwendete Equals-Expression kann mit jedem Datentypen umgehen, hat allerdings die Einschränkung, dass die Rückgabetypen beider Operanden übereinstimmen müssen.

```
var equals = new BinaryExpression()
{
        ExpressionFunction = ExpressionFunction.Equals,
        Left = new
            NullaryExpression(ExpressionFunction.BooleanConstant),
        Right = new
            NullaryExpression(ExpressionFunction.NumberVariable)
};
```

Listing 3.11: Verletzung der inhärenten Parameter-Typ-Abhängigkeit

In Listing 3.11 (siehe auch Abbildung 3.9) erhält die `Equals`-Expression zwei Kind-Expressions, deren unterschiedliche Datentypen einen sinnvollen Vergleich ausschließen.

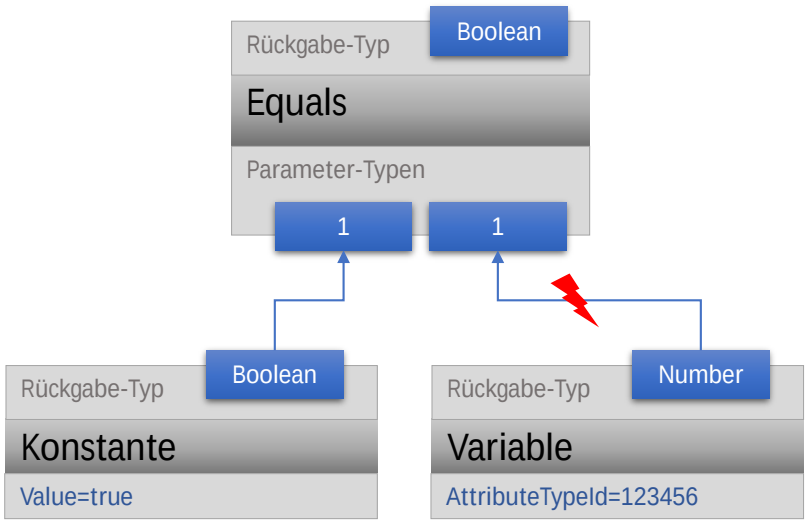


Abbildung 3.9: Verletzung der inhärenten Parameter-Typ-Abhängigkeit

3.2 Anforderungen

In diesem Abschnitt werden die aus der Analyse gewonnenen Anforderungen zusammengetragen.

3.2.1 Funktionale Anforderungen

Die Funktionalen Anforderungen beschreiben die vom System geforderten Features.

A1 Das Verifikation-Framework muss die in Abschnitt 3.1 ermittelten Problemfälle abprüfen und erkennen können. Das heißt, es muss in der Lage sein, die folgenden Fehlerfälle (F1 - F8) zu erkennen:

- F1 Die Arität einer Expression passt nicht zu der definierten `ExpressionFunction` (siehe 3.1.1).
- F2 Die Expression oder eine ihrer Kind-Expressions hat als `ExpressionFunction` die `None`-Funktion definiert (siehe 3.1.1).
- F3 Die Expression selbst oder eines ihrer Kind-Elemente ist nicht initialisiert (siehe 3.1.1).
- F4 Eine Expression hat in einem Kind-Element einen Verweis auf eine übergeordnete Expression (siehe 3.1.1).
- F5 Eine Expression, die eine Variable ist, hat eine ungültige `AttributeTypeId` (siehe 3.1.2).
- F6 Eine Konstanten-Expression hat einen ungültigen Datentyp (siehe 3.1.2).
- F7 Eine Kind-Expression hat einen anderen Datentyp, als die Eltern-Expression für diese Position vorgesehen hat (siehe 3.1.2).
- F8 Eine Kind-Expression verletzt die Typ-Abhängigkeit zu einer anderen Kind-Expression der übergeordneten Eltern-Expression (siehe 3.1.2).

A2 Das Verifikations-Framework muss in der Lage sein, im Fehlerfall jeweils das eigentliche Problem zu lokalisieren (und nicht ein durch jenes Problem verursachte Folgeproblem[5]).

A3 Die Prüfung der einzelnen Fehlerquellen muss deaktivierbar sein und die Reihenfolge der Prüfung der einzelnen Fehlerfälle muss konfigurierbar sein, um so eine flexible Durchführung der Verifikation in unterschiedlichen Szenarien zu ermöglichen.

A4 Das Verification-Framework muss einen Verifikationsreport erstellen, welcher dem Aufrufer eine erfolgreiche Verifikation bestätigt oder alternativ explizite Informationen bezüglich des lokalisierten Problems enthält. Dies beinhaltet die Art des Fehlers und die verursachende Teil-Expression.

3.2.2 Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen beschreiben, unter welchen Bedingungen das System die geforderte Funktionalität liefern muss und welche qualitativen Anforderungen, wie beispielsweise Performance des Systems, eingehalten werden müssen.

Da sich das PHILharmonicFlows-Framework noch in der Entwicklung befindet, muss ein besonderer Schwerpunkt auf die Wartbarkeit und Erweiterbarkeit der zu entwickelnden Lösung gelegt werden.

A5 Die Verifikation soll die Laufzeitkomplexität zur Evaluation einer Expression nur um einen linearen Faktor erhöhen.

A6 Das Verifikations-Framework soll die Definition von Expressioneigenschaften einfach und möglichst zentralisiert umsetzen.

A7 Das Verifikations-Framework soll leicht um weitere Verifikationskomponenten erweiterbar sein.

[5] Es muss also sichergestellt werden, dass die Verifikation einzelner Fehlerarten in der richtigen Reihenfolge ausgeführt wird. Sonst könnte beispielsweise ein Fehler vom Typ F3 gemeldet werden, obwohl die eigentliche Ursache ein Fehler vom Typ F1 ist (also eine falsch eingestellte `ExpressionFunction`).

A8 Um eine gute Wartbarkeit und Erweiterbarkeit des Verifikations-Frameworks zu gewährleisten, sollen folgende Punkte gewährleistet werden:

1 Sinnvoller Einsatz von Entwurfsmustern

2 Orientierung an den Microsoft Coding Conventions, insbesondere bezüglich Code-Kommentierung und -Dokumentation[6]

[6] siehe: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions

4

Entwurf

Aufgabe der Entwurfsphase ist es, aus den vorhandenen Anforderungen ein detailliertes Entwurfskonzept zu entwickeln, welches sich direkt in Code umsetzen lässt [1]. Dieses Kapitel beschreibt den Entwurf, der auf Grundlage der im vorangegangen Kapitel 3.2 ermittelten Anforderungen an das Verifikationsframework erstellt wurde.

Besonders relevant für diese Phase waren die folgenden Fragen:

- Wie kann man ermittelte Informationen über den Aufbau des Expressionbaums zwischenspeichern, um somit die Anzahl der Traversionen zu minimieren?
- Wie kann man sicherstellen, dass die Verifikation unter allen Umständen den richtigen Fehler findet und nicht nur ein Symptom, das eigentlich von einem anderen Fehler verursacht wird?
- Wie erhalten die Verifikationskomponenten die Information darüber, welche Einschränkungen für die verschiedenen Funktionen der `ExpressionFunction` existieren?

UML-Notation

Für die Diagramme wird der UML-2.0-Standard verwendet. Für die Verwendung des Generics-Sprachfeatures von C# werden parametrisierte Klassen genutzt [11]. Als Abweichung vom Standard werden Eigenschaften von Klassen mit blauer Schrift markiert, um Properties zu kennzeichnen, die auf anderen Eigenschaften beruhen. In Abbildung 4.1 ist `BirthDate` eine eigenständige Property, während `AgeInYears` blau markiert ist und somit aus anderen Daten generiert wird, hier aus dem Geburtsdatum.

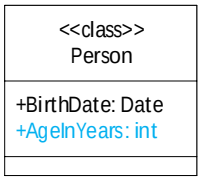


Abbildung 4.1: Notation - Beispiel

4.1 Aufbau der Verifikation

Dieser Abschnitt beschreibt die Bestandteile der entworfenen Lösung und deren Zusammenspiel.

4.1.1 ExpressionWrapper

Für Konsumenten von Expressions ist es ausreichend, diese als die abstrakte Basisklasse `Expression` übergeben zu bekommen, da sie über die `Evaluate`-Methode das benötigte Ergebnis der Expression erhalten, welches intern rekursiv durch die Evaluation der Teilexpressions ermittelt wird. Die Details der Teilexpressions und ihre Zusammensetzung ist dabei nicht von Belang. Das Verifikations-Framework aber ist auf diese Informationen angewiesen.

Um zu erreichen, dass das Laufzeitverhalten zur Verarbeitung einer Expression durch eine Verifikation nur linear beeinflusst wird, muss eine Möglichkeit geschaffen werden, einmal gewonnene Informationen für den Vorgang der Verifikation vorzuhalten. Die `ExpressionWrapper`-Klasse (siehe Abbildung 4.2) kapselt eine `Expression`, ermittelt alle Kind-Expressions und baut so einen Baum von `ExpressionWrapper`-Objekten auf, welcher dem Expressionbaum selbst entspricht. Darüber hinaus liefert diese Klasse zusätzliche Informationen über die jeweilige Expression, wie zum Beispiel, ob die Expression eine Konstante ist oder wieviele Kindelemente sie hat. Dabei bietet sie einen vereinheitlichten Zugriff auf die Kindelemente jeder Teilexpression unabhängig von der Arität der inneren Expression.

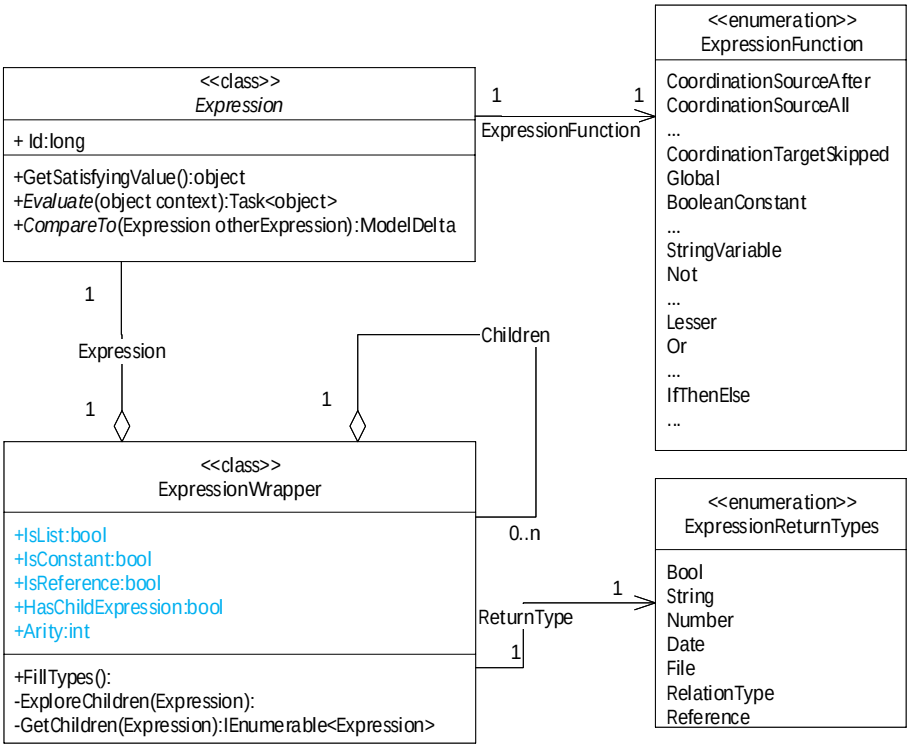


Abbildung 4.2: Klassendiagramm `ExpressionWrapper`

Sobald die strukturelle Korrektheit der Expression verifiziert ist, kann das `ExpressionWrapper`-Wurzelobjekt den Rückgabetyp aller Expressions im Baum mit einer einzelnen Traversion ermitteln und speichern, um so die semantischen Verifikationen zu vereinfachen.

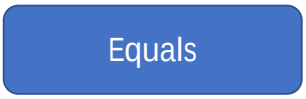


Abbildung 4.3: Expression, wenn sie an das Verifikationsframework übergeben wird

Der Vorgang wird an drei Grafiken illustriert. In Abbildung 4.3 wird gezeigt, welche Informationen über eine Expression verfügbar sind, wenn sie an das Verifikationsframework übergeben wird. Sie wird als die abstrakte Basisklasse `Expression` übergeben und

somit ist nur die `ExpressionFunction` bekannt. Wenn das `Expression`-Objekt an den Konstruktor der `ExpressionWrapper`-Klasse übergeben wird, untersucht dieser das Objekt nach Kindexpressions und erstellt für diese dann rekursiv Kindelemente. Abbildung 4.4 zeigt dies im linken Teil der Grafik. Mit diesen Informationen können die strukturellen Verifikationen (siehe Abschnitt 3.1.1) durchgeführt werden. Wenn die Verifikation die Korrektheit der Struktur bestätigt, kann der `ExpressionWrapper`-Baum als nächstes die Typ-Informationen der Teilexpressions untersuchen. Dazu wird in den Blattelementen begonnen, die Rückgabetypen zu ermitteln. Anschließend wird für jedes Elternelement anhand seiner Typattribute und der ermittelten Rückgabetypen der Kindelemente der Rückgabetyp ermittelt (siehe auch Kapitel 5). Der Zustand nach diesem Schritt wird in der rechten Hälfte von Abbildung 4.4 gezeigt.

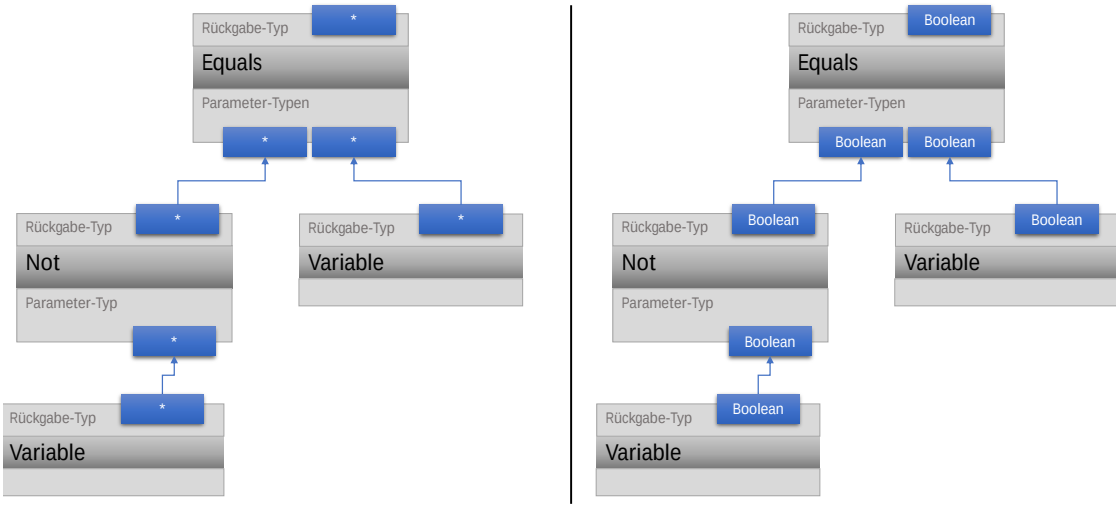
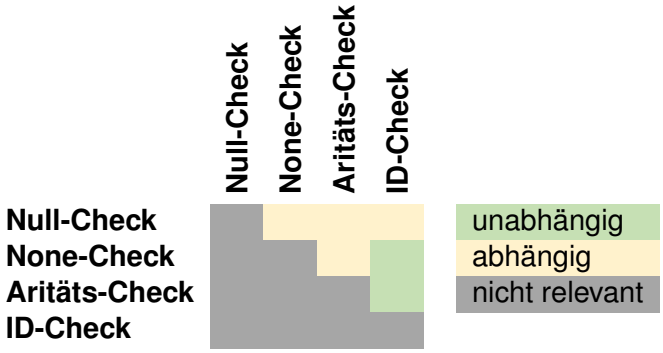


Abbildung 4.4: `ExpressionWrapper`-Baum, links nach der Erstellung, rechts nach der Typanalyse

4.1.2 Verifikationskomponenten

Wie in Abschnitt 3.1 beschrieben, müssen die Verifikationskomponenten in zwei Gruppen aufgeteilt werden: strukturelle und semantische Komponenten. Da die Funktionalität gleich ist, bildet eine gemeinsame Schnittstelle die Basis für die Komponenten.

Tabelle 4.1: Abhängigkeiten der Verifikationskomponenten



Die Schnittstelle `IExpressionVerificationComponent` definiert die Methode `Verify`, die als Parameter ein `ExpressionWrapper`-Objekt erwartet und ein `VerificationResult`-Objekt zurückgibt. Von ihr leiten zwei Marker-Schnittstellen[1] ab, um so die Komponenten nach struktureller und semantischer Verifikation zu unterteilen. Die Verifikationskomponenten implementieren jeweils eine der beiden Marker-Schnittstellen (siehe Tabelle B.1 und Abbildung C.3). Die Verifikationskomponenten sind dabei so entworfen, dass sie jeweils nur die aktuelle Expression (und je nach Art der Verifikation Eltern oder Kindelemente) untersuchen und das Traversieren der Baumstruktur dem übergeordneten Verifikationsmodul überlassen. Eine weitere Eigenschaft der Komponenten ist, dass sie jeweils ausschließlich ihren eigenen Verantwortungsbereich überprüfen sollen. Das heißt, eine Komponente, welche beispielsweise die Gültigkeit der ID einer Expression überprüft, ignoriert, ob eine Expression **`null`** ist. Grundsätzlich ist zwar die Reihenfolge, in welcher die Komponenten aufgerufen werden, vorgegeben, dies kann aber leicht geändert werden und daher müssen die Komponenten mit solchen Situationen umgehen können, ohne False-Positive-Resultate[2] zu erzeugen. Bei Änderungen der Reihenfolge der Verifikationskomponenten müssen die ermittelten Abhängigkeiten zwischen den Komponenten (siehe Tabelle 4.1) berücksichtigt werden.

[1] Marker-Schnittstellen enthalten keine eigenen Definitionen, sondern werden dazu genutzt, Meta-Informationen bereitzustellen [9]. Ein Beispiel dafür ist das `Serializable`-Interface in Java [12].

[2] False-Positive wird in diesem Zusammenhang ein erkannter Fehler sein, der aber eigentlich auf einen anderen Fehler zurückzuführen ist.

4.1.3 Das Resultat der Verifikation

Die Verifikation soll stoppen, wenn der erste Fehler aufgetreten ist. Dies und die ausgewählte Reihenfolge der Verifikation soll sicherstellen, dass in jedem Fall die richtige Ursache des Problems und nicht nur ein Symptom erkannt wird.

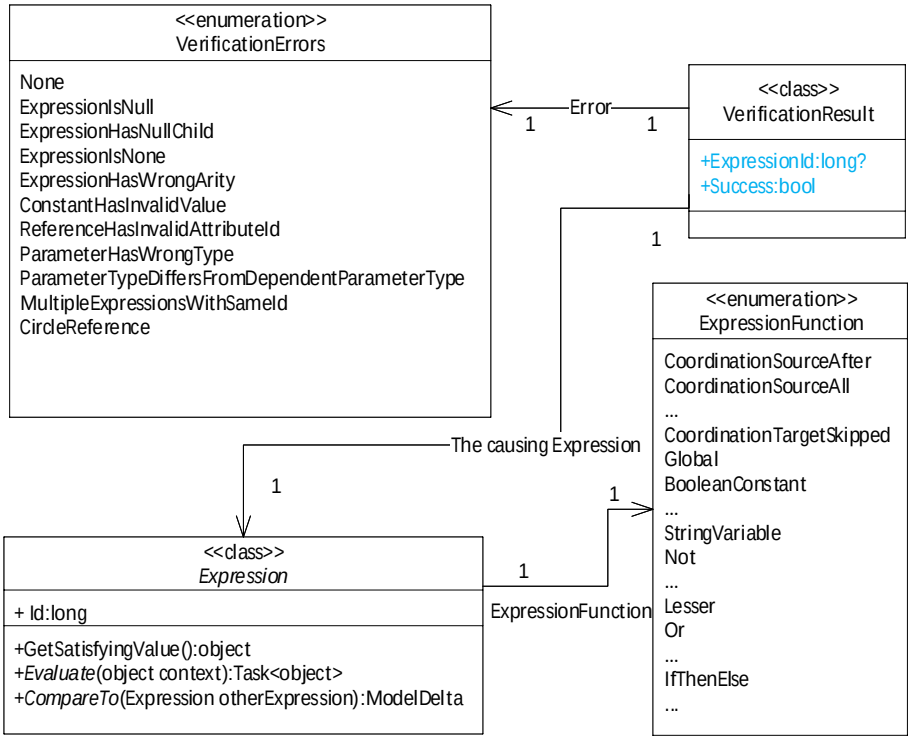


Abbildung 4.5: Klassendiagramm `VerificationResult`

Beispielsweise könnte man vermuten, dass wenn die `ExpressionFunction` einer `NullaryExpression` gleich -1 ist, ein Problem mit der Arität vorliegt, wenn in Wahrheit die ungültige `ExpressionFunction`: `None` die eigentliche Ursache für das Problem ist.

Für die aufrufende Klasse der Expression-Verifikation sind die folgenden Informationen relevant:

- Die `Id` der fehlerhaften Expression. Falls das Problem eine **`null`**-Referenz für eine Kind-Expression ist, wird die `Id` der Eltern-Expression angegeben. Sollte die Wurzel-Expression eine **`null`**-Referenz sein, wird keine `Id` zurückgegeben.
- Die Art des Fehlers, übergeben durch die Enumeration `VerificationErrors`.

Diese Informationen zusammengefasst ergeben die Klasse `VerificationResult` (siehe Abbildung: 4.5). Im Erfolgsfall, der sich über das `Success`-Flag leicht abprüfen lässt, wird keine Expression referenziert.

4.1.4 Pre-Checks

Die `ExpressionWrapper`-Klasse ist so konzipiert, dass Informationen über die Struktur und die Datentypen der Expressions durch Traversieren der Baumstruktur ermittelt werden. Jedes Element in diesem Baum kennt sein Elternelement und seine Kindelemente. Während diese Struktur für die meisten Verifizierungen gut geeignet ist, hat sie sich zum Ermitteln von Zirkelreferenzen, beschrieben in Abschnitt 3.1.1, als ungeeignet erwiesen. Da das Expression-Verifikationsframework darauf ausgelegt ist, mit der Erweiterung des PHILharmonicFlows-Frameworks mitzuwachsen, kann nicht ausgeschlossen werden, dass in Zukunft noch weitere Aspekte der Verifikation hinzukommen, die ebenfalls nicht für die `ExpressionWrapper`-Klasse geeignet sind.

Daher wird eine allgemeine Möglichkeit benötigt, Anforderungen an die Struktur zu überprüfen, bevor die Expression zu einem `ExpressionWrapper`-Baum verarbeitet wird. Die Schnittstelle `IExpressionPreVerificationCheck` definiert die `Check`-Methode, die eine Expression als Parameter erhält und ein `VerificationResult` zurückgibt (siehe Abbildung 4.6). Die einzige tatsächliche Implementierung dieser Schnittstelle ist die `NoCircleReferenceCheck`-Klasse, welche die Expression auf die vorher erwähnten Zirkelreferenzen untersucht. Die einzige andere Implementierung ist die Klasse `ExpressionPreVerificationCheckComposite`, welche allerdings keine eigene Prüfung durchführt sondern dazu dient, alle tatsächlichen Implementierungen von `IExpressionPreVerificationCheck` zu vereinen, anzuordnen und dem Verifikationsframework einen unkomplizierten Zugriff zu bieten.

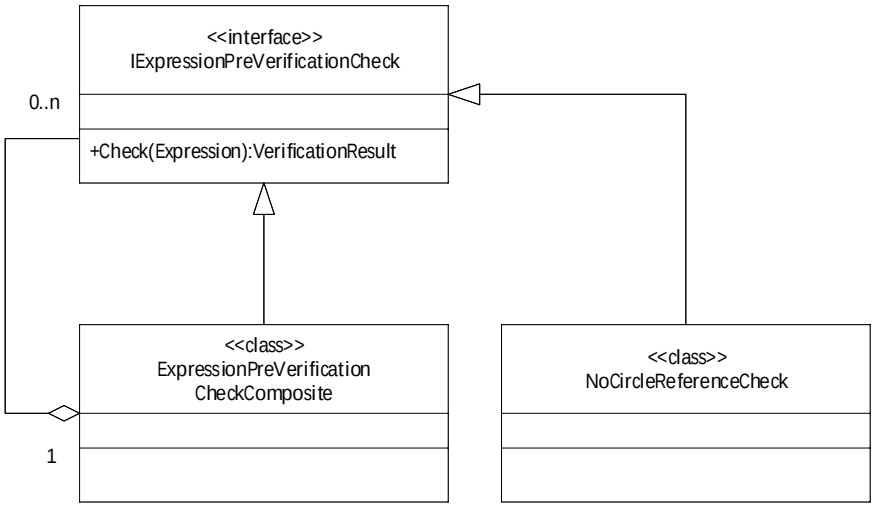


Abbildung 4.6: Klassendiagramm `ExpressionPreVerificationCheck`

4.1.5 Das Verifier-Modul

Vor Benutzern des Expression-Verifikationsframeworks soll die gesamte in den vorherigen Abschnitten beschriebene Komplexität versteckt werden. Das Verifizierungsmodul wird von außen über die Klasse `ExpressionVerifier` angesprochen. Über die Methode `Verify` wird eine `Expression` als Parameter übergeben und ein `VerificationResult` zurückgegeben. Das Resultat wird als `Task` übergeben, damit die Verifizierung nebenläufig erfolgen kann.

Die Klasse `ExpressionVerificationComponentComposite` implementiert die Schnittstelle `IExpressionVerificationComponent`, um, angelehnt an das Composite-Pattern [4], mehrere Verifikationskomponenten als eine Komponente darzustellen. Aufgabe dieser Klasse ist die Bereitstellung und Anordnung der jeweils benötigten Verifikationskomponenten.

Die Schnittstelle `IVerificationFactory` bietet Methoden für die Komposition der einzelnen Komponenten zu einer strukturellen, einer semantischen und der `PreCheck`-Komponente. Die `VerificationFactory` ist die derzeit einzige Implementierung die-

ser Schnittstelle, für zukünftige Anwendungsfälle könnten unterschiedlich konfigurierte Factory-Klassen eine Option sein.

Die Klasse `ExpressionVerifier` (siehe Abbildung 4.7) verwendet die `IVerification-Factory`-Schnittstelle, um die Verifikationskomponenten zu instanziieren. Die öffentliche `Verify`-Methode soll dabei die folgenden Schritte in der vorgegebenen Reihenfolge durchführen:

- Die `PreVerificationCheck`-Prüfungen durchführen.
- Die Expressions in den `ExpressionWrapper` verpacken.
- Die strukturellen Tests durchführen.
- Den `ExpressionWrapper`-Baum mit Typinformationen befüllen.
- Die semantischen Tests durchführen.

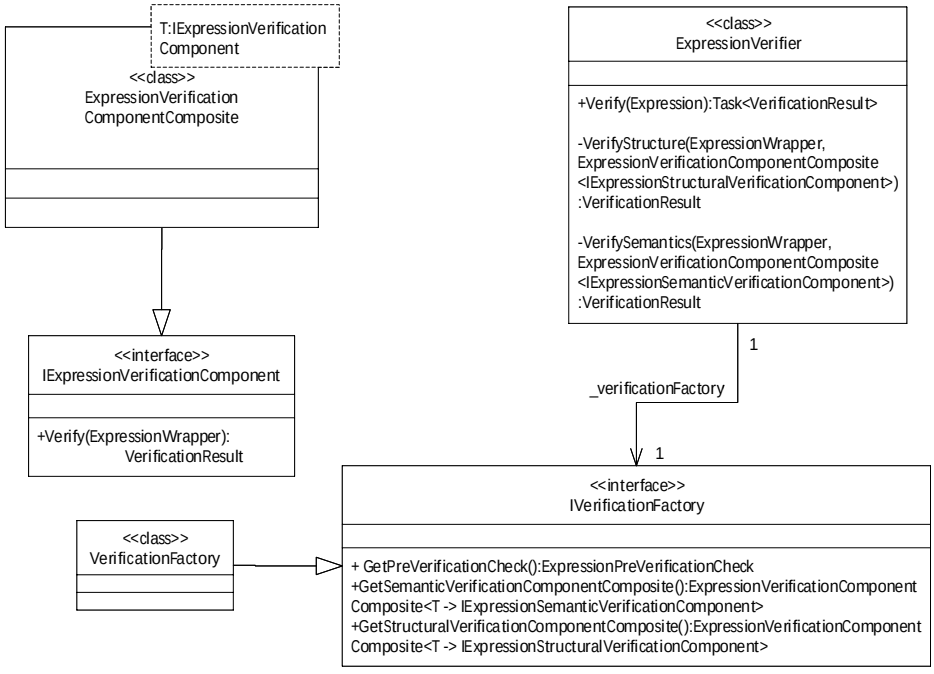


Abbildung 4.7: Klassendiagramm `ExpressionVerifier`

4.2 Meta-Informationen

In den vorangegangenen Abschnitten wurde ein Konzept beschrieben, um Expressions zu analysieren und zu verifizieren. Ein wichtiger Punkt wurde dabei bisher als gegeben angenommen: Das Wissen darum, welche `ExpressionFunction` welche Eigenschaften bezüglich der Datentypen besitzt. Während insbesondere die Struktur-Verifikationen keine zusätzlichen Informationen benötigen, sind die für die semantische Verifikation benötigten Typeigenschaften der einzelnen Funktionen aus `ExpressionFunction` bisher nur implizit bekannt.

Ein naiver Ansatz wäre hier, in den einzelnen Verifikationsklassen große **`switch`**-Statements zu bilden, die je nach `ExpressionFunction` eine gesonderte Prüfung abhängig von den Anforderungen an die entsprechende Funktion durchführen können. Dieser Ansatz hat jedoch mehrere Nachteile. Zum einen würden so Informationen bezüglich der Typ-Einschränkungen auf mehrere Klassen aufgeteilt werden, so dass sie für zukünftig an dem System arbeitende Entwickler schwerer zugänglich wären. Noch relevanter ist allerdings, dass eine solche Lösung die Wartbarkeit des Verifikationsframeworks stark einschränkt. Jedes Einführen einer neuen Funktion würde neben ihrer Definition in der `ExpressionFunction`-Enumeration auch das Erweitern der betreffenden Verifikationskomponenten erfordern.

Ein zielführenderer Ansatz wäre, diese Informationen unabhängig von der Verifikation als Eigenschaften der jeweiligen `ExpressionFunction` zu betrachten. Und in der Tat sind an dieser Stelle bereits einige Eigenschaften wie die Arität oder der Anzeigename für jede Funktion definiert. Mit einem Set an zusätzlichen Attributen könnten alle Eigenschaften der jeweiligen Funktion in der `ExpressionFunction`-Enumeration deklarativ definiert werden und dann zur Laufzeit vom Verifikationsframework analysiert werden, um so zu ermitteln, welche Typabhängigkeiten überprüft werden müssen.

Ein erster Entwurf sah vor, dass alle Typinformationen einer Funktion frei kombinierbar in einem einzigen Attribut zusammengefasst werden würden. Dies hätte den Vorteil, dass pro Funktion nur ein Attribut hätte hinzugefügt werden müssen, in welchem dann verschachtelt die Eigenschaften für den Rückgabetyp und alle Parametereigenschaften

deklariert worden wären. Allerdings hat eine Testimplementierung in einem Prototypen gezeigt, dass eine solche Lösung nicht umsetzbar ist, da die `Attribute`-Klassen so definiert sind, dass als Argumente für den Konstruktor dynamische Strukturen wie beispielsweise eine variable Anzahl an Parametern mit dem **`param`**-Keyword[3] nicht vorgesehen sind. Lediglich Konstrukte, die bereits zur Kompilierungszeit verfügbar sind, wie zum Beispiel primitive Datentypen oder Enumerationen, dürfen verwendet werden. Da die variable Anzahl von Argumenten eine zwingende Voraussetzung für diesen Ansatz war, wurde der Entwurf verworfen.

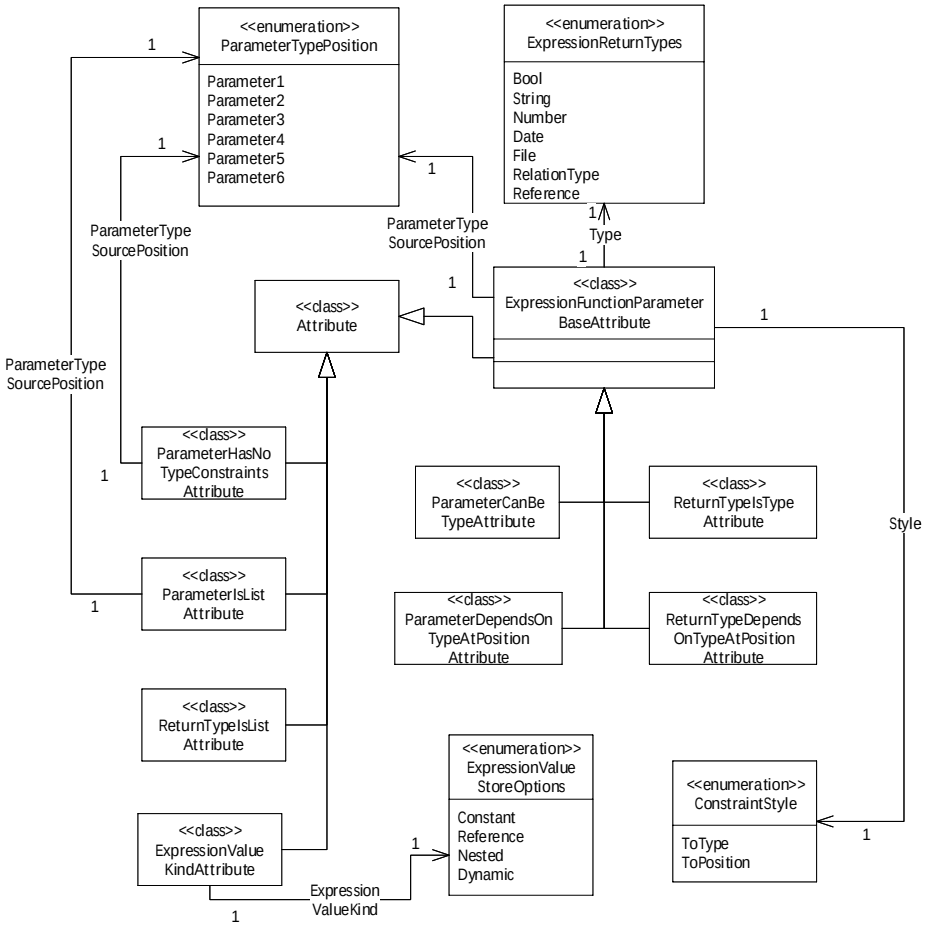


Abbildung 4.8: Klassendiagramm Attribute für `ExpressionFunction`

[3] Das **`param`**-Keyword gestattet es, eine Methode mit einer variablen Anzahl von Argumenten aufzurufen. Diese können dann in der Methode über ein `Array` angesprochen werden.

Der finale Entwurf (siehe Abbildung 4.8) sieht pro Eigenschaft ein eigenes Attribut vor. Dies hat den Nachteil, dass die `ExpressionFunction`-Enumeration aufgrund der Mehrzahl an Attributen Gefahr läuft, unübersichtlicher zu werden, stellt aber mit Blick auf die Grundanforderung, alle Metainformationen an einer Stelle zu definieren, die beste Lösung dar.

Die Enumeration `ParameterTypePosition` gibt die Position des Parameters an, für welchen eine Eigenschaft definiert wird. In Tabelle 4.2 werden zunächst die verwendeten Aufzählungstypen erläutert.

Tabelle 4.2: Übersicht über die von den Expression-Attributen verwendeten Enumerationen

Enumeration	**Beschreibung**
`ParameterTypePosition`	Gibt die Position des Parameters an, für welchen die Eigenschaft definiert ist.
`ExpressionReturnTypes`	Diese Aufzählung wird benutzt, um die Datentypen der Blatt-Elemente im Expressionbaum zu definieren.
`ConstraintStyle`	Gibt an, ob eine Abhängigkeit für einen Datentyp oder auf einen anderen Parameter definiert ist.
`ExpressionValueKindAttribute`	Gibt an, wie Expressions ihre Daten verwalten.

Die Attribute teilen sich in zwei Gruppen auf. Die Attribute der ersten Gruppe leiten sich direkt von der `Attribute`-Klasse aus dem .net-Framework ab (siehe Tabelle B.4).

Die zweite Gruppe von Attributen leitet sich zunächst von der abstrakten Basisklasse `ExpressionFunctionParameterBaseAttribute` von `Attribute` ab. Diese Klasse enthält Definitionen, um einen bestimmten Datentyp und/oder eine Position festzulegen, auf welche sich das Attribut bezieht. Die von der Basisklasse abgeleiteten Attribute werden in Tabelle B.3 erklärt.

5

Implementierung

Die Entwicklung des Verifikationsframeworks fand außerhalb des PHILharmonicFlows-Frameworks statt und wird nachträglich in dieses importiert. Dadurch konnten Änderungen und Erweiterungen der bestehenden Expression-Struktur vorgenommen werden, ohne davon abhängige Anwendungen zu stören. Die Tabelle B.2 beschreibt den Aufbau der Namensräume, ihre Anordnung ist dargestellt in Abbildung C.2.

Das Projekt wurde in der Entwicklungsumgebung Visual Studio 2017 in der Enterprise Edition entwickelt.

Im vorliegenden Kapitel wird die Umsetzung des Entwurfs mit einigen Code-Beispielen beschrieben. Die umfangreicheren Code-Ausschnitte (über eine halbe Seite lang) sind im Anhang zu finden. Da die Struktur des Codes umgesetzt wurde wie durch den Entwurf im Kapitel 4 beschrieben, liegt der Schwerpunkt in diesem Kapitel auf der Beschreibung einer Auswahl an Implementierungsdetails.

5.1 Beschreibung der Expression-Eigenschaften

Im vorangegangenen Kapitel wurde im Abschnitt 4.2 ein Konzept zur Deklaration von Metadaten zu den einzelnen Funktionen der Expressions beschrieben. Dazu sollten Attributklassen erstellt und den einzelnen Funktionen zugewiesen werden. Listing 5.1 zeigt die Metainformationen für die `Boolean`-Konstante. In Zeile 2 wird der Rückgabetyp dieser Funktion auf Bool festgelegt. Mit dieser Information kann während der Verifikation für eine übergeordnete Expression geprüft werden, ob sie an dieser Stelle eine Expression mit dem Boolean-Datentyp zulässt. In Zeile 3 wird definiert, wie die Expres-

sion ihren Wert speichert. Für Variablen und Konstanten wäre es grundsätzlich auch möglich gewesen, diese Information aus dem Bezeichner oder den bereits vorhandenen Metainformationen zu generieren. Ein solche, vermeintlich simple Lösung hätte jedoch Limitierungen bei der Benennung zukünftiger Funktionen nach sich gezogen und die Wartbarkeit in Bezug auf die Anpassbarkeit an neue Anforderungen eingeschränkt. Mit der vorliegenden Lösung kann diese Information nicht nur eindeutig deklariert werden, darüber hinaus kann so auch der tatsächliche Datentyp angegeben werden, so dass dieses Attribut alle erforderlichen Informationen enthält, um die Daten in den Blättern des Expressionbaums verifizieren zu können.

```
1 [EnumMember] [Display(Name = "Constant<Bool>")] [Arity(0)]
2 [ReturnTypeIsType(ExpressionReturnTypes.Bool)]
3 [ExpressionValueKind(ExpressionValueStoreOptions.Constant,
     typeof(bool))]
4 BooleanConstant,
```

Listing 5.1: Auszug aus der `ExpressionFunction`-Enumeration

Listing 5.2 zeigt ein Beispiel für eine ternäre Expression, die `IfThenElse`-Expression. Während der Datentyp für den ersten Operanden auf den Boolean-Datentyp festgelegt ist (siehe Zeile 3), sind die anderen zwei Operanden nicht auf einen bestimmten Datentyp limitiert, müssen aber den gleichen Datentyp haben. In Zeile 4 wird definiert, dass der zweite Operand keine Beschränkungen hat und in Zeile 5 wird für den dritten Operanden festgelegt, dass dieser vom zweiten Parameter abhängt, also dessen Datentyp besitzen muss. Darüber hinaus hängt für diese Funktion auch der Rückgabetyp vom zweiten Parameter ab.

```
1 [EnumMember] [Display(Name = "?:")] [Arity(3)]
2 [ReturnTypeDependsOnTypeAtPosition(ParameterTypePosition.Parameter2)]
3 [ParameterCanBeType(ParameterTypePosition.Parameter1,
     ExpressionReturnTypes.Bool)]
4 [ParameterHasNoTypeConstraints(ParameterTypePosition.Parameter2)]
5 [ParameterDependsOnTypeAtPosition(ParameterTypePosition.Parameter3,
     ParameterTypePosition.Parameter2)]
6 [ExpressionValueKind(ExpressionValueStoreOptions.Nested)]
7 IfThenElse,
```

Listing 5.2: Auszug aus der `ExpressionFunction`-Enumeration

Der Schwerpunkt bei der Implementierung der Attribute zur Beschreibung der Eigenschaften der einzelnen Expression-Funktionen lag auf deren einfacher Kombinierbarkeit. Listing A.1 zeigt ein Beispiel für die Implementierung der Attribute. In Listing A.2 wird die Auswertung eines Attributes am Beispiel einer Verifikation dargestellt.

5.2 Verifikation

Um den Verifikationsreport als Ergebnis der Verifikation möglichst aussagekräftig zu machen, wurde die Basisklasse `Expression` um die Eigenschaft `Id` vom Typ **`long?`**[1] erweitert. Dies ermöglicht es, für negative Resultate die verursachende Expression beziehungsweise deren ID mitzuliefern. Diese Information kann dann wiederum beispielsweise in der Modellierungsumgebung genutzt werden, um dem Modellierer visuelle Unterstützung bei der Eingrenzung des Problems zu geben.

5.2.1 Expressionanalyse

Die `ExpressionWrapper`-Klasse ist dafür verantwortlich, die an das Verifikationsframework übergebene Expression für die Verifikationskomponenten aufzubereiten. Dafür

[1] Das Fragezeichen hinter dem primitiven Datentyp zeigt an, dass der Wert auch **`null`** sein darf.

wird der Expression-Baum in Pre-Order[2] traversiert und ein entsprechender Baum aus `ExpressionWrapper`-Objekten gebildet. Die in diesem Baum verfügbaren Informationen (zum Beispiel der direkte Zugriff auf die Kindelemente) ermöglicht die Durchführung aller strukturellen Verifikationen. Sobald diese erfolgreich komplettiert worden sind, muss für die semantische Prüfung der `ExpressionWrapper`-Baum abermals traversiert werden, diesmal jedoch in Post-Order[3]-Reihenfolge (siehe Listing A.3). Dabei werden zunächst für die Blattelemente und dann jeweils für die Elternelemente die entsprechenden Rückgabetypen ermittelt. In der Regel ist der erwartete Rückgabetyp als `ReturnTypeIsTypeAttribute` direkt definiert. Sollte der Rückgabetyp jedoch von einem der Parameter abhängen, muss stattdessen der bereits ausgewertete Rückgabetyp der entsprechenden Kindexpression verwendet werden.

5.2.2 Verifikationskomponenten

Listing A.2 zeigt als Beispiel die Implementierung der `ParameterHasExpectedDependentTypeVerificationComponent`. Zu Beachten ist, dass intern einige erforderliche Bedingungen an die Struktur geprüft werden. Sollten diese Prüfungen fehlschlagen, wird ein positives Resultat zurückgegeben, da diese Probleme jeweils von anderen Komponenten geprüft werden und die aktuelle Verifikation nicht durchgeführt werden kann, ohne dass diese Bedingungen erfüllt sind.

Die Komponenten werden zu Beginn einer Verifikation von der Fabrik erstellt und dann für die Verifikation jedes einzelnen Expressionobjekts wiederverwendet.

Für die Pre-Checks (siehe Abschnitt 4.1.4) wird im Listing A.4 die derzeit einzige Implementierung der `Check`-Methode gezeigt. Um eine Zirkelreferenz zu vermeiden, muss für jede Verbindung zwischen einem Blattelement und dem Wurzelelement im Expressionbaum gelten, dass jede Expression nur einmal vorkommt. Um dies zu prüfen, wird die Expression in Post-Order-Reihenfolge durchlaufen und der aktuelle Zweig in einem Stack verwaltet. Wird eine Expression erreicht, die sich bereits im Stack befindet, wird

[2] Bei der Pre-Order-Traversierung wird zuerst das Elternelement durchlaufen und dann in der vorgegebenen Reihenfolge von links nach rechts die Kindelemente.

[3] Bei der Post-Order-Traversierung werden zunächst die Kindelemente in der definierten Reihenfolge und dann das Elternelement durchlaufen.

die Prüfung abgebrochen und ein Fehlerbericht erstellt. Andernfalls wird der Stack vom Blattelement bis zur nächsten Abzweigung verworfen und der Prozess mit dem nächsten Zweig wiederholt.

Struktur-Verifikation

Die strukturellen Verifikationskomponenten prüfen den Aufbau des Expressionbaumes. Hervorgehoben werden sollen hier zwei dieser Komponenten.

Die `ExpressionIsNotNullVerificationComponent`-Klasse prüft die aktuelle Teilexpression darauf, ob eine der Kindexpressions nicht initialisiert ist. Da die semantischen Verifikation auf dem Expressionbaum von der Wurzel hin zu den Blättern (in Post-Order) ausgeführt wird, besteht nur beim Wurzelelement die Gefahr, dass die Expression selbst uninitialisiert ist. Da die Verifikation aber so entworfen ist, dass die Komponente selbst kein Wissen darüber hat, wo im Baum sich die aktuell zu prüfende Expression befindet[4], muss ausgeschlossen werden, dass versucht wird, die Kindelemente eines nicht initialierten Wurzelelements zu prüfen. Daher wird das Elternelement bei dieser Verifikation immer mit geprüft.

Im vorherigen Absatz wurde hervorgehoben, dass Verifikationskomponenten immer nur auf der aktuellen Expression agieren und den sonstigen Baum außer Acht lassen sollen. Die einzige Ausnahme dieser Regel ist die `ExpressionHasUniqueIdVerificationComponent`-Klasse. Um die Einzigartigkeit aller ID's im Expressionbaum überprüfen zu können, muss diese Klasse die ID's aller bereits besuchten Expressions kennen, was intern über ein Hashset realisiert wird.

Semantische Verifikation

Die semantische Verifikation überprüft die formale Korrektheit unter der Annahme der strukturellen Korrektheit. Dies betrifft sowohl die korrekte Vorhaltung von Daten in den

[4] Die Verifikationskomponenten können Blattelemente erkennen, dies spielt in diesem Fall aber keine Rolle.

Blattelementen als auch die Einhaltung der Datentypeinschränkungen der jeweiligen `ExpressionFunction`. Für beide Bereiche wird jeweils eine Komponente vorgestellt.

Die `ConstantHasValidValueVerificationComponent`-Klasse prüft jede Expression darauf, ob diese eine Konstante ist. Wenn dies zutrifft, wird das `ExpressionValueKindAttribute` für die entsprechende `ExpressionFunction` untersucht. Dieses Attribut gibt an, wie die jeweilige Funktion ihren Wert speichert und darüber hinaus kann für Konstanten auch der (C#-)Datentyp definiert werden. Dieser definierte Datentyp wird dann mit dem Datentyp der `Value`-Property der Expression verglichen.

Für jeden Parameter einer Expression kann eine beliebige Anzahl von direkten Typeinschränkungen definiert werden. Dabei wird nach einem Whitelist-Prinzip festgelegt, welche Typen für einen Parameter erlaubt sind. Die `ParameterHasExpectedTypeVerificationComponent`-Klasse iteriert über jeden Parameter einer Expression. Dabei überprüft sie, ob für den Parameter direkte Typeinschränkungen definiert sind. Sollte dies der Fall sein, wird geprüft, ob der Rückgabetyp der entsprechenden Kindexpression mit einem der in den Typeinschränkungen definierten Typen übereinstimmt.

5.2.3 Verifikationsmodul

Die `ExpressionVerifier`-Klasse ist das Element, welches von Konsumenten der Expressionverifikation aufgerufen wird. Die dort definierte `Verify`-Methode erwartet eine Expression als Argument und liefert ein `VerificationResult` als `Task` zurück.

Innerhalb der Methode werden die folgenden Aktionen durchgeführt:

1. Der Pre-Check wird ausgeführt.
2. Der `ExpressionWrapper`-Baum wird erstellt.
3. Der `ExpressionWrapper`-Baum wird in Pre-Order-Reihenfolge durchlaufen. Auf jedem Element des Baums werden die strukturellen Verifikationen durchgeführt.
4. Der `ExpressionWrapper`-Baum wird einer Typanalyse unterzogen.
5. Der `ExpressionWrapper`-Baum wird in Post-Order-Reihenfolge durchlaufen. Auf jedem Element des Baums werden die semantischen Verifikationen durchgeführt.

Sollte einer dieser Schritte auf Probleme treffen, wird der Verifikationsvorgang abgebrochen. In jedem Fall wird am Ende ein Verifiktionsreport erstellt und zurückgegeben.

5.3 Codequalität

Großer Wert wurde während der Implementierungsphase dem Erreichen einer hohen Codequalität beigemessen, um so zur Erreichung der Anforderung A8 (siehe Abschnitt 3.2.2) beizutragen. Dies umfasste unter anderem:

1. Angemessene und einheitliche Code-Dokumentation
2. Vermeidung von Redundanzen
3. Sprechende Methoden- und Variablenbezeichnungen
4. Eine hohe Testabdeckung (> 90%)

Des Weiteren gehörte auch die sinnvolle Strukturierung des Codes zu den Maßnahmen. So zeigt Abbildung 5.1 beispielsweise, dass die Abhängigkeiten bezüglich der Klassen, die andere Klassen aufrufen oder instanziieren, alle in dieselbe Richtung verlaufen.

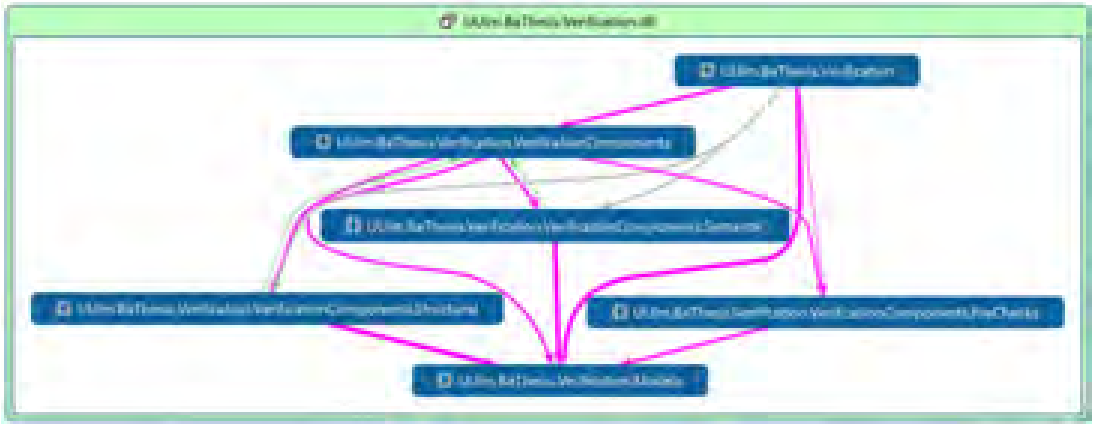


Abbildung 5.1: Paketstruktur der Verifikationsassembly

Dabei beschreiben die violetten Pfeile die Aufrufe, die grauen Pfeile repräsentieren Verweise und die grünen Pfeile die Implementierung von Schnittstellen.

6

Verwandte Arbeiten

Das PHILharmonicFlows-Framework verwendet Expressions auf eine spezielle Weise und benötigt volle Kontrolle über sie. Daher greift PHILharmonicFlows nicht auf bestehende Expression-Frameworks zurück, sondern setzt auf eine Eigenentwicklung. Der Bereich der objektzentrierten Prozessmanagementsysteme befindet sich zum Zeitpunkt der Erstellung dieser Arbeit noch in der Entstehung und vergleichbare Systeme stehen derzeit nicht zur Verfügung. In diesem Kapitel werden daher zwei Frameworks vorgestellt, die einen anderen fachlichen Schwerpunkt haben, aber ebenfalls Expressions einsetzen.

6.1 Drools

Drools ist ein von Red Hat entwickeltes Business-Rule-Management-System und Bestandteil von JBoss Enterprise BRMS[1] [15, 14]. Die Regeln werden in der Drools-Sprache als Expressions in Aussagenlogik-Form definiert. In Listing 6.1 wird in einem einfachen Beispiel, entnommen aus den Testdaten von Drools, die Definition einer Regel gezeigt.

Diese Regeln werden intern in Entscheidungstabellen übersetzt und anschließend verifiziert. Das Ergebnis der Verifikation wird in einem Verifikationsreport bereit gestellt.

```
17 package org.drools.verifier.visitor;
18 rule "Test 1"
19 when
20 Person(
```

[1] JBoss Enterprise BRMS ist eine Plattform für Business-Rule-Management.

```
21 age > 0  && < 100
22 )
23 then
24 System.out.println( "Test" );
25 end
```

Listing 6.1: Beispiel für eine Regel-Expression aus der Drools-Teststruktur, Dateiname: `ExprConstraintDescr2.drl`

6.2 Expressions in .net

Mit .net Version 3.5 wurde 2007 das LINQ[2]-Framework veröffentlicht. Der Zweck des Frameworks ist die Vereinheitlichung des Zugriffes auf unterschiedliche Datenquellen (wie zum Beispiel SQL, XML oder interne Datenstrukturen wie Listen) mit Support durch Entwicklungsumgebungen wie Visual Studio [18]. Als Teil des LINQ-Frameworks wurden Expressions unter dem Namensraum `System.Linq.Expressions` eingeführt. Expressions werden vom LINQ-Framework genutzt, um Abfragen an Datenquellen abzubilden oder auszuführen.

Im Rahmen dieser Arbeit wurde der Quelltext dieser Expressions daraufhin untersucht, wie Expressions dort verifiziert werden. Dazu wurde das JustDecompile-Tool[3] von Telerik® in der 2017er Version verwendet. Auch wenn der Einsatzzweck der LINQ-Expressions sich deutlich von dem der PHILharmonicFlows-Expressions unterscheidet, sind Ähnlichkeiten unverkennbar. So enthält der Expressions-Namensraum von LINQ Definitionen für die Klassen `UnaryExpression` und `BinaryExpression`, und die Enumeration `ExpressionType` definiert die verfügbaren Funktionen.

Die Untersuchung hat ergeben, dass Expressions im LINQ-Framework nicht verifiziert werden. Stattdessen findet zur Laufzeit eine Validierung statt. Sollten dabei Probleme auftreten, wirft das Framework eine Exception.

[2] LINQ steht für Language Integrated Query
[3] https://www.telerik.com/products/decompiler.aspx

7

Abschluss

In diesem Kapitel wird zunächst die vorliegende Arbeit zusammengefasst. Abschließend wird erläutert, welche Weiterentwicklungen für das Verifikations-Framework als möglich erachtet werden.

7.1 Zusammenfassung

Das Ziel dieser Arbeit war die Planung und Entwicklung eines Expression-Verification-Frameworks für das PHILharmonicFlows-Framework. Zunächst wurde dafür in Kapitel 2 PHILharmonicFlows vorgestellt, anschließend die dort verwendeten Expressions erläutert und die Verifikation in Zusammenhang mit Expressions beschrieben. Kapitel 3 analysierte die Expressions in Hinblick auf Problemquellen und beschrieb anschließend die daraus abgeleiteten Anforderungen an ein Verifikations-Framework. Im Kapitel 4 wurde auf der Grundlage der ermittelten Anforderungen ein Entwurf erstellt, welcher die Verifikation selbst und die Erweiterung der Expressiondefinition um Metainformationen umfasst. In Kapitel 5 wurde beschrieben, wie der zuvor erstellte Entwurf in Software umgesetzt und getestet wurde. Alle Tests des Systems sind positiv verlaufen, sodass einem Transfer in das PHILharmonicFlows-Framework nichts im Wege steht.

7.2 Mögliche Erweiterungen

Die an das Verifikations-Framework gestellten Anforderungen (definiert in Abschnitt 3.2) wurden erfüllt. Bedingt durch die Tatsache, dass sich das PHILharmonicFlows-

Framework noch in der Entwicklungsphase befindet, ist es möglich, dass neue Anforderungen an PHILharmonicFlows eine Erweiterung oder Änderung des Verifikations-Frameworks erfordern. Darüber hinaus ist es auch denkbar, das Verifikations-Framework selbst weiterzuentwickeln. Im Folgenden werden drei Ansätze vorgestellt.

Parallele Ausführung

Während die strukturellen Verifikationen unbedingt vor den semantischen durchgeführt werden müssen, wäre eine parallele Ausführung der einzelnen Verifikationen denkbar. Während der Verifikation findet kein Schreibzugriff statt. Zu Bedenken wäre aber, dass die Auswertung der Ergebnisse noch immer in definierter Reihenfolge ausgeführt werden muss.

Des Weiteren wäre auch die nebenläufige Verifikation von mehreren Kind-Expressions möglich. Hierbei müsste beachtet werden, dass die Auswertung in Post-Order erfolgt, um das erwartete Resultat zu erhalten.

Erweiterung des Verifikationsberichtes

Es wäre möglich, das `VerificationResult` so zu erweitern, dass es bei Problemen mit Operanden noch die Position des Operanden angeben kann. Dies würde für einige Fälle eine noch genauere Auswertung ermöglichen. Als Beispiel sei hier ein nicht initialisiertes Kindelement genannt. In einem solchen Fall wird aktuell nur die ID der Eltern-Expression angegeben.

Geschwindigkeitstest

Im Rahmen der Testdurchführung wurde ein Expressiongenerator entwickelt, mit welchem sich sehr große Expressionbäume generieren lassen. Diese könnten in einem weiteren Schritt genutzt werden, um in der Praxis zu testen, ob die Verifikation die Komplexität der Verarbeitung eines Expressionbaums wie erwartet nur linear beeinflusst.

Literaturverzeichnis

[1] Balzert, H.: Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2 (German Edition). Spektrum Akademischer Verlag (2011)

[2] Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-Based Runtime Verification. In: Verification, Model Checking, and Abstract Interpretation. pp. 44–57. Springer (2004)

[3] Chiao, C.M., Künzle, V., Andrews, K., Reichert, M.: A Tool for Supporting Object-Aware Processes. In: IEEE 18th Int'l Distributed Object Computing Conference - Workshops and Demonstrations (EDOCW). pp. 410–413. IEEE Computer Society Press (2014)

[4] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1994)

[5] ISO: International Standard ISO/IEC 9899:1999: Programming languages — C (Nov 2007)

[6] Künzle, V.: Object-Aware Process Management. Ph.D. thesis, University of Ulm (July 2013)

[7] Künzle, V., Reichert, M.: A Modeling Paradigm for Integrating Processes and Data at the Micro Level. In: Proc. 12th Int'l Working Conference on Business Process Modeling, Development and Support (BPMDS). pp. 201–215. No. 81 in LNBIP, Springer (June 2011)

[8] Kühnel, A.: Visual C# 2008. Rheinwerk Verlag (2008)

[9] Mayer, J.: On Quality Improvement of Scientific Software - Theory, Methods, and Application in the GeoStoch Development. Tenea Verlag Ltd., Berlin (2005)

[10] Microsoft: Garbage Collection. https://msdn.microsoft.com/de-de/library/0xy59wtx(v=vs.110).aspx (2017), zuletzt abgerufen am 01.12.2017

[11] Oestereich, B.: Die UML-Kurzreferenz 2.3 für die Praxis: Kurz, Bündig, Ballastfrei (German Edition). De Gruyter Oldenbourg (2009)

[12] Oracle: Interface Serializable. https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html (2017), zuletzt abgerufen am 01.12.2017

[13] Rothmaier, D.: Evaluation der Modellierungskonzepte eines objektzentrierten Prozessmanagementsystems. Master's thesis, Ulm University (2017)

[14] Shi, J., Qiao, Y., Wang, H.: Visualizing Inference Process of a Rule Engine. In: Proceedings of the 2011 Visual Information Communication-International Symposium. p. 10. ACM (2011)

[15] Sottara, D., Mello, P., Sartori, C., Fry, E.: Enhancing a Production Rule Engine with Predictive Models using PMML. In: Workshop on Predictive Markup Language Modeling. pp. 39–47. ACM (2011)

[16] Steinau, S.: Design and Implementation of a Runtime Environment of an Object-Aware Process Management System. Master's thesis, Ulm University (2015)

[17] Steinau, S., Künzle, V., Andrews, K., Reichert, M.: Coordinating Business Processes Using Semantic Relationships. In: 19th IEEE Conference on Business Informatics (CBI). pp. 33–42. IEEE Computer Society Press (July 2017)

[18] Wagner, B., Wenzel, M., B., M., Latham, L., Hoag, S.: Language Integrated Query (LINQ). https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/index (2017), zuletzt abgerufen am 01.12.2017

[19] Wenger, R.: Handbuch der .NET 4.0-Programmierung Band 1 C# und .NET-Grundlagen. Microsoft GmbH (2010)

A

Quelltexte

In diesem Anhang sind einige wichtige Quelltexte aufgeführt.

```
1  namespace UUlm.BaThesis.ExpressionSim.ExpressionAttributes
2  {
3          /// <summary>
4          /// Marks a parameter as list.
5          /// </summary>
6          /// <seealso cref="System.Attribute" />
7          [AttributeUsage(AttributeTargets.Field, AllowMultiple = true)]
8          public class ParameterIsListAttribute : Attribute
9          {
10                 /// <summary>
11                 /// Initializes a new instance of the <see
                       cref="ParameterIsListAttribute"/> class.
12                 /// </summary>
13                 /// <param name="parameterTypeTargetPosition">The
                       parameter type target position.</param>
14                 public ParameterIsListAttribute(ParameterTypePosition
                       parameterTypeTargetPosition)
15                 {
16                       ParameterTypeTargetPosition =
                             parameterTypeTargetPosition;
17                 }
18
```

```
19                  /// <summary>
20                  /// Gets the parameter type target position.
21                  /// </summary>
22                  public ParameterTypePosition
                        ParameterTypeTargetPosition { get; }
23          }
24  }
```

Listing A.1: Beispiel-Attribut: ParameterIsListAttribute

```
1   namespace UUlm.BaThesis.Verification.VerificationComponents.Semantic
2   {
3   /// <summary>
4   ///     Checks if all parameters that have indirect type constraints
5   ///     (which means, that their type has to be the same type as the
6   ///     related parameter type) meet their constraints.
7   /// </summary>
8   /// <seealso cref="IExpressionSemanticVerificationComponent" />
9   public class ParameterHasExpectedDependentTypeVerificationComponent :
        IExpressionSemanticVerificationComponent
10  {
11  /// <summary>
12  ///     Verifies the specified expression wrapper.
13  /// </summary>
14  /// <param name="expr">The expression wrapper.</param>
15  /// <returns>The result of the verification.</returns>
16  public VerificationResult Verify(ExpressionWrapper expr)
17  {
18          //this gets checked by another verification component
19          if (expr?.Expr == null)
20                  return new VerificationResult();
21
```

```
22          //an expression without children can't harm the condition
                checked by this component.
23          if (!expr.HasChildExpressions)
24                  return new VerificationResult();
25
26          //find all parameters which types depends on other parameters.
27          var type = typeof(ExpressionFunction);
28          var memberInfo =
                type.GetMember(expr.Expr.ExpressionFunction.ToString());
29          var directTypeAttributes = memberInfo
30                  .SelectMany(x => x.GetCustomAttributes(typeof(
31                          ParameterDependsOnTypeAtPositionAttribute),
                            false))
32                  .OfType<ParameterDependsOnTypeAtPositionAttribute>()
33                  .ToList();
34
35          //foreach of the found parameters above, check if they have
                the type as the related parameter
36          if (directTypeAttributes.All(
37                  parameterDependsOnTypeAtPositionAttribute =>
38                  expr.Children.ElementAt((int)
                        parameterDependsOnTypeAtPositionAttribute
39                          .ParameterTypeSourcePosition)
40                          .ReturnType == expr.Children.ElementAt((int)
                            parameterDependsOnTypeAtPositionAttribute
41                                  .ParameterTypeTargetPosition)
42                          .ReturnType))
43                  return new VerificationResult();
44
45          return new VerificationResult(expr.Expr, VerificationErrors
46                  .ParameterTypeDiffersFromDependentParameterType);
```

```
47 }
48 }
49 }
```

Listing A.2: `ParameterHasExpectedDependentTypeVerificationComponent`, Beispiel für die Auswertung einer `ExpressionFunction`-Metainformation

```
1  /// <summary>
2  ///     Fills the type informations for the given wrapper and its
       childs.
3  /// </summary>
4  /// <param name="ew">The wrapper.</param>
5  /// <exception cref="ArgumentException">
6  ///     Gets thrown, if a parameter position defined in the <see
       cref="ExpressionFunction" />
7  ///     is out of range for the arity of the expression function.
8  /// </exception>
9  private static void FillTypes(ExpressionWrapper ew)
10 {
11 if (ew?.Expr == null)
12 return;
13
14 //fill children first - postorder
15 if (ew.Children != null && ew.Children.Count > 0)
16 foreach (var expressionWrapper in ew.Children)
17 FillTypes(expressionWrapper);
18
19
20 var type = typeof(ExpressionFunction);
21 var memInfo = type.GetMember(ew.Expr.ExpressionFunction.ToString());
22
23 //standard case: return type is specific type
```

```
24
25 var attributes = memInfo.SelectMany(x => x.GetCustomAttributes(
26         typeof(ReturnTypeIsTypeAttribute), false));
27
28 if (attributes.Any())
29 {
30 var returnTypeIsTypeAttribute = (ReturnTypeIsTypeAttribute)
       attributes.Single();
31
32 if (ew.Arity == ArityEnum.ForNullaryExpression)
33 if (returnTypeIsTypeAttribute == null)
34 throw new ArgumentException();
35 if (returnTypeIsTypeAttribute != null)
36 {
37 ew.ReturnType = returnTypeIsTypeAttribute.Type;
38 return;
39 }
40 }
41
42 //alternative case: return type depends on parameter type
43
44 attributes =
45 memInfo.SelectMany(
46 x => x.GetCustomAttributes(
47         typeof(ReturnTypeDependsOnTypeAtPositionAttribute), false));
48 var returnTypeDependsOnTypeAttribute =
       (ReturnTypeDependsOnTypeAtPositionAttribute) attributes.Single();
49 if ((int)
       returnTypeDependsOnTypeAttribute.ParameterTypeSourcePosition >
       (int) ew.Arity)
```

```
50 throw new ArgumentException("Parameter position doesn't fit in the
       desired arity. " +
51 $"Parameter position: {(int)
       returnTypeDependsOnTypeAttribute.ParameterTypeSourcePosition}, " +
52 $"arity: {ew.ArityValue}");
53 if (ew.Children == null)
54 throw new ArgumentException("Return type can't depend on child if
       expression has no children.");
55
56 ew.ReturnType = ew.Children.ElementAt(
57 (int) returnTypeDependsOnTypeAttribute.ParameterTypeSourcePosition)
58 .ReturnType;
59 }
```

Listing A.3: `ExpressionWrapper`-Klasse, Auszug

```
1  /// <summary>
2  ///     Checks the specified expression for pre conditions.
3  /// </summary>
4  /// <param name="expression">The expression.</param>
5  /// <returns>
6  ///     The result of the pre check.
7  /// </returns>
8  /// <remarks>
9  ///     Traversal in preorder
10 /// </remarks>
11 public VerificationResult Check(Expression expression)
12 {
13         //check the current expression
14         if (_alreadyVisitedExpressions.Contains(expression))
15                 return new VerificationResult(expression,
                       VerificationErrors.CircleReference);
```

```
16
17          //add the current expression to the list of checked
                expressions
18          _alreadyVisitedExpressions.Push(expression);
19
20          //check all child epxressions
21          foreach (var child in GetChilds(expression))
22          {
23                  var result = Check(child);
24                  if (!result.Success)
25                          return result;
26                  }
27
28          _alreadyVisitedExpressions.Pop();
29
30          return new VerificationResult();
31  }
```

Listing A.4: `NoCircleReferenceCheck`-Klasse, Auszug

B

Tabellen

Tabelle B.1: Übersicht über die Verifikationskomponenten.

Komponente	Aufgabe
`ExpressionHasCorrectArity-VerificationComponent`	Prüft, dass die `ExpressionFunction` zur verwendeten Expression-Klasse passt.
`ExpressionHasUniqueId-VerificationComponent`	Prüft, dass die ID der Expression eindeutig ist. (Diese Komponente benötigt im Gegensatz zu allen anderen Verifikationskomponenten nicht nur Informationen zu benachbarten Elementen im Expressionbaum, sondern zu allen bereits untersuchten Elementen.)
`ExpressionIsNotNone-VerificationComponent`	Prüft, dass die `ExpressionFunction` nicht die None-Funktion ist.
`ExpressionIsNotNull-VerificationComponent`	Prüft, dass die Expression keine **`null`**-Referenzen enthält.
`ConstantHasValidValue-VerificationComponent`	Prüft, dass Konstanten einen gültigen Wert besitzen.
`ParameterHasExpected-DependentType-VerificationComponent`	Prüft, dass Parameter, deren Typ abhängig von anderen Parametertypen ist, den korrekten Typ haben.
`ParameterHasExpectedType-VerificationComponent`	Prüft, dass Parameter, die auf bestimmte Datentypen beschränkt sind, einen zulässigen Typ haben.
`ReferenceHasValidAttribute-TypeIdVerificationComponent`	Prüft, dass Variablen-Expressions eine gültige `AttributeTypeId` besitzen.

Tabelle B.2: Aufgaben der Namensräume

Namensraum	**Beschreibung**
`Uulm.BaThesis.ExpressionSim`	Enthält die originalen Expression-Klassen.
`Uulm.BaThesis.ExpressionSim.ExpressionAttributes`	Enthält die Attribute für die `ExpressionFunction`-Enumeration.
`Uulm.BaThesis.Verification`	Basis-Namensraum für die Expression-Verifikation. Enthält die `ExpressionVerifier`-Klasse, über welche das Verifikationsframework von außen angesprochen wird.
`Uulm.BaThesis.Verification.Models`	Beinhaltet unter anderem die `ExpressionWrapper`-Klasse und das `VerificationResult`.
`Uulm.BaThesis.Verification.VerificationComponents`	Basis-Namensraum für die Verifikationskomponenten. Beinhaltet die Basisschnittstelle für die Verifikation und die Verifikationsfabrik.
`Uulm.BaThesis.Verification.VerificationComponents.PreChecks`	Umfasst die Struktur für die Pre-Verifikationschecks.
`Uulm.BaThesis.Verification.VerificationComponents.Semantic`	Beinhaltet alle semantischen Verifikationskomponenten.
`Uulm.BaThesis.Verification.VerificationComponents.Structural`	Beinhaltet alle strukturellen Verifikationskomponenten.

Tabelle B.3: Von `ExpressionFunctionParameterBaseAttribute` ableitende Attribute

Attribut	**Beschreibung**
`ParameterCanBeTypeAttribute`	Um die Deklaration von Typeinschränkungen leichter lesbar zu machen, wurde ein Whitelist-Ansatz gewählt. Dabei wird definiert, welche Typen ein Parameter haben darf. Es ist vorgesehen, einen Parameter mit mehreren Typeinschränkungen belegen zu können.
`ParameterDependsOnType-AtPositionAttribute`	Dieses Attribut sagt aus, dass der Typ des Parameters von einem anderen Parameter abhängt.
`ReturnTypeIsTypeAttribute`	Im Gegensatz zu den Parametern kann der Rückgabetyp einer Expression nur auf einen einzigen Typen festgelegt werden. Eine flexiblere Definition des Rückgabetypen ist mit dem Attribut in der nachfolgenden Zeile möglich.
`ReturnTypeDependsOnType-AtPositionAttribute`	Zeigt an, dass der Rückgabetyp nicht direkt definiert ist, sondern von dem Typen eines Parameters abhängt. Ein Beispiel dafür ist die `IfThenElse`-Funktion, deren Rückgabetyp vom Typen des zweiten und dritten Parameters abhängt.

Tabelle B.4: Direkt von `Attribute` ableitende Attribute

Attribut	**Beschreibung**
`ParameterHasNoType-ConstraintsAttribute`	Das Attribut zeigt an, dass für den Parameter, auf den es verweist, keine Einschränkungen vorliegen. Es wird vom Framework nicht verarbeitet und soll lediglich anzeigen, dass nicht vergessen wurde, eine Einschränkung zu deklarieren.
`ParameterIsListAttribute`	Obwohl es zum Zeitpunkt dieser Arbeit noch keine Listen-Funktionen in der `Expression-Function` gibt, ist vorgesehen, diese in einer späteren Version des PHILharmonicFlows-Frameworks zu implementieren. In Kombination mit dem `ParameterCanBeTypeAttribute` kann mit diesem Attribut für einen Parameter als erwarteter Typ eine Liste mit einem bestimmten Datentyp definiert werden.
`ReturnTypeIsListAttribute`	Deklariert den Rückgabetypen einer Funktion als Liste. Definiert zusammen mit `ReturnType-IsTypeAttribute` den Rückgabetypen als Liste eines bestimmten Datentyps.
`ExpressionValueKindAttribute`	Dieses Attribut definiert, wie die jeweilige Expression ihre Daten verwaltet. Auf diese Weise können die unterschiedlichen Typen von Funktionen, die Daten direkt verwalten, leicht markiert werden.

C

Grafiken

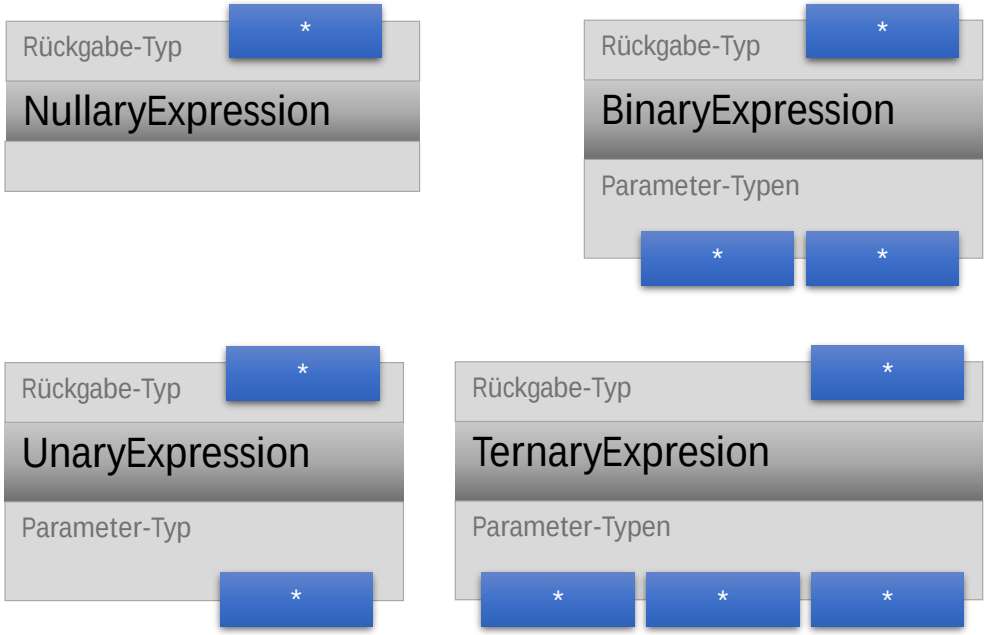


Abbildung C.1: Die unterschiedlichen Aritäten der Expressions

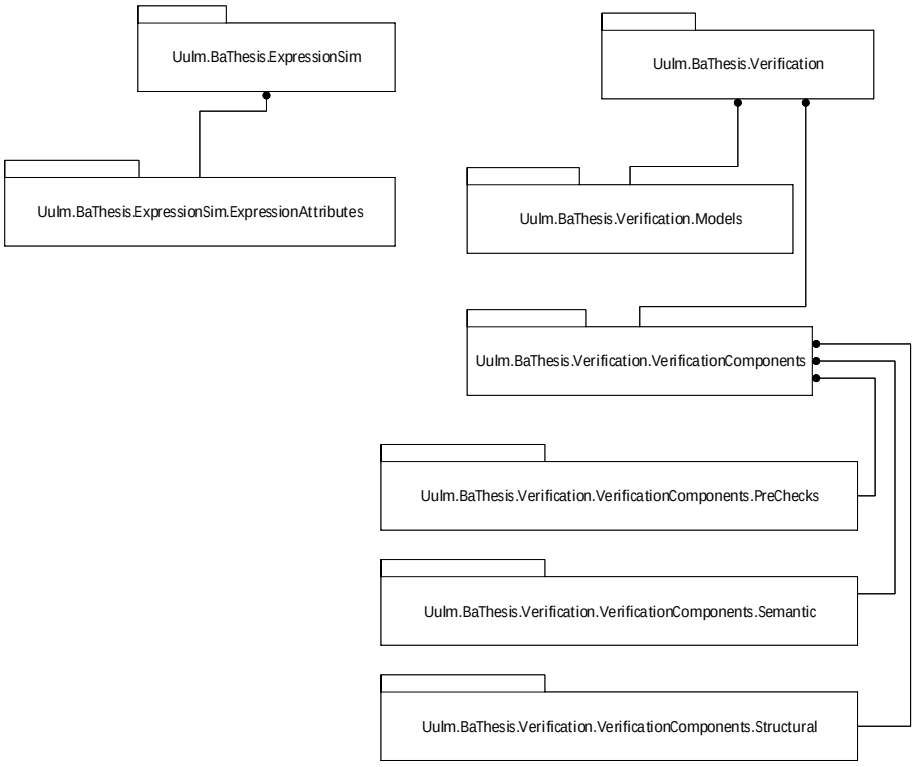


Abbildung C.2: Paketstruktur beziehungsweise Organisation der Namensräume

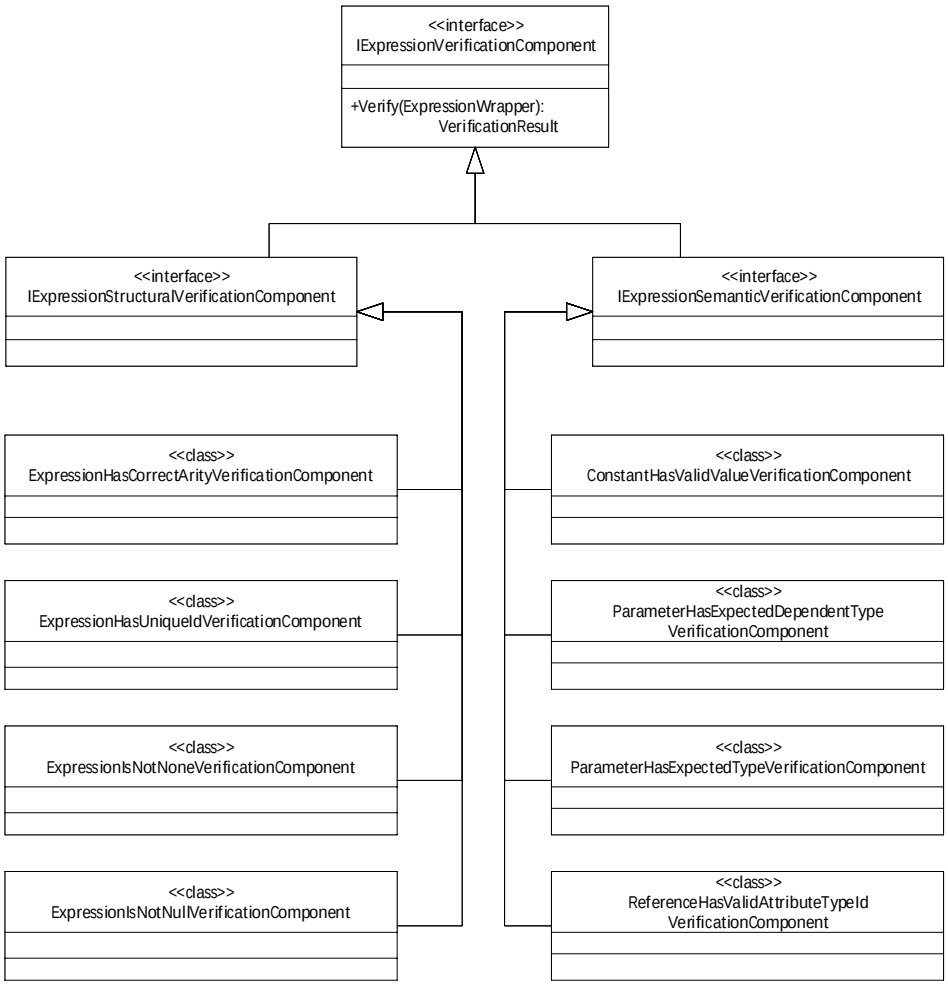


Abbildung C.3: Klassendiagramm Verifikationskomponenten

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Name: Maik Schönfeld Matrikelnummer: 859641

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Maik Schönfeld