Informatik-Praktikum für Fortgeschrittene

Fachbereich Mathematik und Informatik

Philipps-Universität Marburg

Im Wintersemester 2003/2004

Implementierung der Weiterentwicklung eines Programmverifizierers in Java

Version 2.0

von

Uli Schäfer

Christian Hohmann

Silvia Ockenfels

Manuela Viehmeyer

betreut von:

Professor Dr. H. Peter Gumm

&

Jörn Abels

Inhaltsverzeichnis

1 Problem- und Aufgabenstellung

Das Ziel des Praktikums ist die Weiterentwicklung eines Programmverifiziers für annotierte Programme einer an die Programmiersprache angelehnten, imperative Sprache.

Der Verifizierer soll Programme dieser Sprache, welche mit logischen Bedingungen und Invarianten versehen sind, beweisen. Er ist für die Verwendung in der Lehre der *Floyd/Hoare Programmverifikation* gedacht.

Der Beweis kann durch zwei Arten erfolgen: Entweder entscheidet sich der Benutzer für den automatischen Beweismodus, in dem der Verifizierer selbständig arbeitet, oder für den manuellen Modus, in dem der Benutzer über jeden einzelnen Schritt selbst entscheidet, während der Verifizierer nur Umformungsmöglichkeiten angibt.

Als Vorlage dient dabei der von Prof. Dr. H. Peter Gumm entwickelte Verifizierer *NPPV* („*New Paltz Program Verifier*“), welcher in der Programmiersprache Prolog implementiert ist und Pascal-ähnliche Programme verifiziert, und der von Markus Hampel und Manuel Werner entwickelte *JPV* („*Java Program Verifier*“) *Version 1.0*, welcher in der Programmiersprache Java implementiert ist, für gegebene Programme Verifikationsbedingungen ermittelt, diese in einer Baumstruktur repräsentiert und bereits einen automatischen Beweismodus zum Verifizieren der Bedingungen zur Verfügung stellt. An dieser Stelle sei insbesondere auf die Dokumentation des *JPV Version 1.0* hinweisen.

Die Grundaufgabe besteht somit in der Programmierung des manuellen Beweismodus mit den dazugehörigen Fensterstrukturen und Funktionen.

Die Aufgabe gliedert sich wie folgt:

- Entwicklung eines rein manuellen Beweismodus, in dem Verifikationsbedingungen Schritt für Schritt per Hand vereinfacht werden

- Entwicklung einer manuellen Ergänzung zum automatischen Beweismodus, die es ermöglicht, Verifikationsbedingungen manuell zu verifizieren, die im automatischen Modus nicht bewiesen werden konnten

- Standardisierung und Anpassung der internen Repräsentation der Verifikations-bedingungen als Baumstruktur

- Optimierung und Neu-Implementierung von Vereinfachungsregeln, um den Verifizierer sowohl im automatischen als auch im manuellen Beweismodus zu verbessern

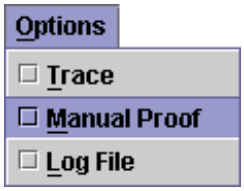
2 Technische Realisierung

2.1 Manueller Beweis

Wie bereits dargestellt, war es Teil der Aufgabe, einen Modus zum manuellen Beweis der Verifikationsbedingungen zu implementieren. Dieser Modus sollte in zweifacher Hinsicht zum Einsatz kommen. Zum einen sollte es möglich sein, die Verifikationsbedingungen von vorneherein manuell zu bearbeiten, ohne auf das automatische Beweisverfahren zurückzugreifen, das in *JPV Version 1.0* bereits integriert und weitgehend unangetastet in *Version 2.0* übernommen wurde.

Nebst diesem rein manuellen Modus sollte der Anwender immer dann die Möglichkeit haben, Bedingungen manuell zu vereinfachen, wenn das automatische Beweisverfahren nicht in der Lage ist, diese mit den gegebenen Regeln vollständig zu verifizieren.

2.1.1 Rein manueller Beweis



Durch Aktivieren des Menüpunktes *Manual Proof* unter *Options* wird der rein manuelle Modus aktiviert. Wird im Folgenden für ein geöffnetes Programm die Aktion *Prove* unter *Verifier* ausgeführt, so werden wie in *Version 1.0* die Verifikationsbedingungen erzeugt. Statt jedoch sofort die automatische Verifikationsroutine für die erste Bedingung zu starten und das Ergebnis in einem neuen Fenster zu präsentieren, wird die Bedingung nun unverändert in einem neuen Fenster ausgegeben. Der Anwender kann nun selbständig die Bedingung bearbeiten (siehe hierzu 2.1.3). Im rein manuellen Modus stehen ihm außerdem drei weitere Optionen zu Verfügung:

- Durch Aktivieren des *Next*-Buttons wird die derzeitig bearbeitete Verifikationsbedingung verworfen und im Fenster die nächste Bedingung angezeigt. Im Fenstertitel ist jederzeit abzulesen, die wievielte Bedingung gerade dargestellt wird. Der *Next*-Button wird automatisch deaktiviert, falls keine weitere Verifikationsbedingung mehr vorhanden ist.

- Der *Reset*-Button macht die bisher vorgenommenen Vereinfachungsschritte rückgängig und setzt die Anzeige auf die ursprünglich zu beweisende Verifikationsbedingung zurück.

- Der *Accept and close*-Button beendet den rein manuellen Beweismodus und schließt das Fenster.

2.1.2 Manuelle Ergänzungen zum automatischen Beweismodus

Wird das Verifikationsverfahren im automatischen Modus gestartet, so arbeitet das Programm die Verifikationsbedingungen der Reihe nach ab und meldet dem Benutzer jeweils, ob die Bedingung bewiesen werden konnte. Wurde eine Bedingung nicht vollständig verifiziert, so

gibt das Programm die vereinfachte Bedingung aus, die automatisch nicht weiter reduziert werden konnte.

In diesem Fall ist der neu eingefügte Button *Manual* am unteren Rand des Fensters aktiv. Wird er betätigt, so öffnet sich ein weiteres Fenster, das diese automatisch nicht vereinfachbare Bedingung darstellt. Gemäß der in 2.1.3 beschriebenen Vorgehensweise kann die Formel manuell bearbeitet werden. Es stehen außerdem die aus dem rein manuellen Beweismodus bekannten *Reset*- und *Accept and close*-Buttons zur Verfügung.

2.1.3 Vereinfachung der Bedingungen im manuellen Modus

Im manuellen Beweismodus kann der Anwender bestimmte Teilausdrücke der zu verifizierenden Bedingung markieren. Das Programm liefert anschließend eine Liste derjenigen Ausdrücke, zu denen der markierte Ausdruck gemäß der implementierten Regeln umgewandelt werden kann. Hierbei ist zu beachten, dass speziell für den manuellen Modus Vereinfachungsregeln hinzugefügt wurden (siehe 2.2.1), die auch dann greifen, wenn im automatischen Modus eine vollständige Verifizierung nicht möglich war.

Markiert man den zu vereinfachen Ausdruck anhand des entsprechenden Operators und drückt anschließend die rechte Maustaste, so öffnet sich das PopUp-Menü *Anwendbare Formeln*.

Man beachte, dass sich nach Drücken der rechten Maustaste die Markierung automatisch an einen sinnvoll bearbeitbaren Ausdruck anpasst. Markiert der Anwender in der Formel

$$5 + k * 1 > 0 ==> k >= -5$$

das *, so dehnt sich die Markierung nach einem Drücken der rechten Maustaste auf den Ausdruck *k * 1* aus:

$$5 + k * 1 > 0 ==> k >= -5$$

Sollte die angepasste Markierung nicht mit dem beabsichtigten Ausdruck übereinstimmen, dann sollte der Anwender den Vorgang wiederholen und darauf achten, nur die Operation des gewünschten Ausdruck mit der niedrigsten Präzedenz auszuwählen.

Das PopUp-Menü enthält immer den Menüpunkt *Evaluate*. Wird dieser ausgewählt, so wird das automatische Beweisverfahren auf den markierten Ausdruck angewendet, d.h. das Programm vereinfacht den Ausdruck selbständig mit den zur Verfügung stehenden Regeln (findet keine Veränderung statt, so konnte keine Regeln angewendet werden).

Abhängig vom markierten Ausdruck stehen im PopUp-Menü weitere Punkte zur Auswahl. Jeder Menüpunkt stellt den markierten Ausdruck nach Anwendung einer passenden Vereinfachungsregel dar. Wird ein solcher Menüpunkt ausgewählt, so wird in der Bedingung der markierte Ausdruck durch diesen ersetzt. Iterativ kann man so die Bedingung weiter vereinfachen und verifizieren. Eventuell kann man in Zukunft neben den bereits eingefügten Vereinfachungsregeln (siehe 2.2) weitere Regeln implementieren, die die manuelle Verifizierung erleichtern.

2.1.4 Markierung anpassen und entsprechenden Knoten extrahieren

Markiert der Anwender während der manuellen Bearbeitung einen Teil der Verifikationsbedingung und drückt die rechte Maustaste, so muss aus der gegebenen Anfangs- und Endposition der Markierung der kleinste sinnvolle Teilbaum der Bedingung ermittelt werden, der den markierten Ausdruck enthält. Wie bereits in 2.1.3 beschrieben sollte ein Operator ausgewählt werden, um Mehrdeutigkeiten bei der Auswahl zu vermeiden. Es müssen aus der reinen `String`-Repräsentation Rückschlüsse auf die zugrundeliegende `SimpleNode` gezogen werden. Um dies zu bewerkstelligen wurden der Klasse `SimpleNode` weitere Methoden hinzugefügt:

- `public IntList dumpExprList()`
- `public IntList dumpExprWithPrecedencesList(int prec, boolean parenthesis, boolean isRoot, IntList liste, int lang)`

Die Methode `public IntList dumpExprList()` liefert unter Verwendung der Methode `public IntList dumpExprWithPrecedencesList(int prec, boolean parenthesis, boolean isRoot, IntList liste, int lang)` eine Liste von Typ `Integer`, die an i-ter Position die Länge des i-ten Zeichens in der String-Repräsentation einer `SimpleNode` enthält. Dabei wird die `SimpleNode` entsprechend der Methode zur Ermittlung der geklammerten `String`-Repräsentation durchlaufen. Es werden Leerzeichen berücksichtigt und immer der benachbarten Operation zugerechnet.

Anschließend wird das k-te Zeichen im `String`-Ausdruck ermittelt, das der Startposition der Markierung entspricht. Um den entsprechenden Knoten zu erhalten, wird die `SimpleNode`-Represäntation der Verifikationsbedingung in einer Liste mit Elementen vom Typ `SimpleNode` dargestellt, in der die einzelnen Knoten in der Reihenfolge der entsprechenden `String`-Repräsentation enthalten sind. Der k-te Knoten in dieser Liste stellt nun den ausgewählten Ausdruck dar. Sollte es sich hierbei nicht um eine Operation handeln (nur diese sind sinnvoll zu vereinfachen), so wird der entsprechende Elternknoten ausgewählt.

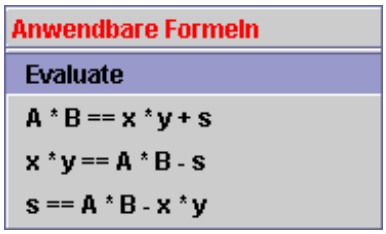
Es ist abschließend der Teil der `String`-Repräsentation zu markieren, der dem derart ermitteln Knoten entspricht. Hierzu wird die `String`-Repräsentation dieses Knotens mittels `public String dumpExpr()` und `public String dumpExprOneLine()` generiert und in der `String`-Repräsentation lokalisiert. Sollte der Teilstring mehrmals in der Gesamtrepräsentation auftreten, wird anhand der Position der ursprünglichen Markierung sichergestellt, dass nicht der falsche Teilstring markiert wird.

Eine genauere Erläuterung der neuen Methoden befindet sich im Abschnitt 2.4.

2.1.5 Das PopUp-Menü

Wie bereits angesprochen erfolgt im manuellen Beweismodus die Vereinfachung der Verifikationsbedingung über eine Liste von Vereinfachungsmustern, die in einem PopUp-Menü aus der Swing-Toolbox dargestellt werden.

Das PopUp-Menü enthält immer den Menüpunkt *Evaluate*. Wird dieser ausgewählt, so wird das automatische Beweisverfahren auf den markierten Ausdruck angewendet.



Abhängig vom markierten Ausdruck stehen im PopUp-Menü weitere Punkte zur Auswahl. Jeder Menüpunkt stellt den markierten Ausdruck nach Anwendung einer passenden Vereinfachungsregel dar. Das Programm erzeugt einen Array von passenden Vereinfachungsmustern, indem der ausgewählte Ausdruck an die Methode `public SimpleNode[] getResults(SimpleNode ptree)` der Klasse `Simplifier` übergeben wird, welche mittels Aufruf von `public boolean reduceOne(SimpleNode ccondition, Rules rules, int rule, boolean output)` jede Regel auf eine Kopie des Ausdrucks anwendet und im Erfolgsfall das Ergebnis in den Array aufnimmt. (Beachte: Bei dieser Vorgehensweise wird der Ausdruck, im Unterschied zum Verfahren beim automatischen Beweismodus, selbst nicht verändert.)

2.2 Regeloptimierung

2.2.1 Vereinfachungsregeln für den manuellen Modus

Die Vereinfachungsregeln sind allgemein bekannte Regeln aus der Mathematik und insbesondere der Logik.

Die Regeln sind technisch folgendermaßen umgesetzt und implementiert. Der allgemeine Aufbau einer Vereinfachungsregel besteht aus drei Teilen: einem Muster *pattern*, einem Vereinfachungsmuster *simpPattern* und in manchen Fällen einer Bedingung *condition*.

Zusätzlich zu den bereits in *Version 1.0* vorhandenen Regeln zum automatischen Beweismodus wurden neue algebraische, boolesche und relationale Regeln hinzugefügt (siehe Anhang A). Diese Regeln können im automatischen Beweismodus nicht verwendet werden, da sie entweder symmetrisch sind (z.B. X == Y vereinfacht zu Y == X) oder Umkehrungen zu bereits vorhandene Regeln darstellen, und daher der automatische Beweisalgorithmus bei Verwendung dieser Regeln zwangsläufig nicht terminieren würde.

In der Klasse `ListOfSimpRules`, in der die Vereinfachungsregeln durch die Methode `fill(int type, boolean manual)` generiert werden, ist durch die Überprüfung der Variablen `manual` vor den für den manuellen Gebrauch bestimmten Regeln gesichert, dass diese Regeln ausschließlich im manuellen Modus auf Anwendbarkeit getestet werden.

2.2.2 Modifikation des Regelsystems

Um ein Entfernen bzw. Hinzufügen von Vereinfachungsregeln zu erleichtern, wurde die in *Version 1.0* vorgenommene Implementierung des Regelsystems überarbeitet. Die Regeln werden nicht wie vorher durch das Aufrufen durchnummerierten Methoden der Gestalt `createRuleXY()` in den Klassen `AlgSimp` (algebraische Vereinfachungsregeln), `AlgSimp` (relationale Vereinfachungsregeln) und `BoolSimp` (boolesche Vereinfachungsregeln) generiert. Stattdessen wird in diesen Klassen jeweils ein Objekt `listrules` vom Typ

`ListOfSimpRules` erstellt. Dabei handelt es sich, wie der Name schon sagt, um eine Liste, deren Elemente Vereinfachungsregeln vom Typ `SimpRule` sind.

Durch die Methode `fill(int type, boolean manual)` wird anschließend die Liste mit den Vereinfachungsregeln vom jeweiligen Typ (0 für algebraisch, 1 für relational, 2 für boolesch) gefüllt, wobei im Fall von `manual = true` auch Regeln für den manuellen Gebrauch zugelassen werden. Anschließend wird die so erstellte Liste in einen Array `rules[]` vom Typ `SimpRule` umkopiert, dessen Länge mit der der Liste übereinstimmt, um die vorhandene Programmstruktur nicht vollständig abändern zu müssen.

Dieses Vorgehen hat gegenüber der *Version 1.0* den Vorteil, dass die Anzahl der Verifikationsbedingungen, also die Länge des Arrays `rules[]`, und die Methodenköpfe bei Veränderungen der Regeln nicht angepasst werden müssen. Zum Entfernen einer Regel genügt es, die entsprechenden Zeilen in der Methode `fill(int type, boolean manual)` der Klasse `ListOfSimpRules` zu entfernen. Analog können neue Regeln eines entsprechenden Typs einfach hinzugefügt werden, ohne den Quellcode an anderer Stelle verändern zu müssen. Man beachte allerdings, dass im automatischen Modus die Regeln in der Reihenfolge abgearbeitet werden, in der sie in der Methode `fill(int type, boolean manual)` aufgelistet sind.

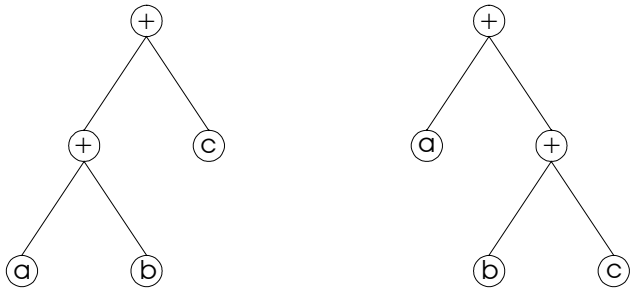
2.3 Strukturelle Modifikationen

2.3.1 Problematik der internen Repräsentation

Bei der manuellen Bearbeitung der Verifikationsbedingungen stellte sich das Problem, dass bestimmte Teilausdrücke aufgrund der internen Darstellung als Binärbaum nicht oder nur eingeschränkt markiert werden konnten. Speziell bei der Aneinanderreihung von Operationen gleicher Präzedenz führt die `SimpleNode`-Repräsentation dazu, dass nicht beliebige benachbarte Terme isoliert ausgewählt werden können. Zur Verdeutlichung des Problems betrachte das folgende Beispiel bzgl. der Addition. Tritt ein Term der Gestalt $a + b + c$ auf, so gilt natürlich wegen der Assoziativität der Addition

$a + b + c = (a + b) + c = a + (b + c)$.

Intern gibt es für die `SimpleNode`-Repräsentation zwei Möglichkeiten:



Der linke Baum entspricht dem Ausdruck *(a + b) + c*. Es ist demzufolge nicht möglich den Term *b + c* isoliert zu markieren. Analoges gilt im rechten Baum für den Term *a + b*. Da arithmetisch nicht notwendige Klammern nicht ausgegeben werden, ist es für den Anwender nicht ersichtlich, welche Struktur vorliegt. Des Weiteren sollte es unabhängig von der internen Repräsentation möglich sein, beliebige benachbarte Terme, sofern mathematisch korrekt, auszuwählen und zu bearbeiten. Identische Probleme treten im Zusammenhang mit *, & und | auf.

Um diese Probleme zu lösen, war es notwendig, die Gestalt der Bäume in gewisser Hinsicht zu standardisieren (siehe 2.3.2).

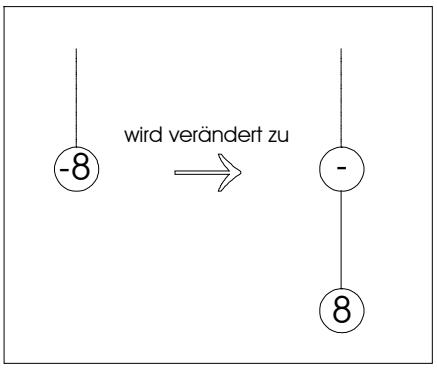
Aufgrund der Standardisierung lässt sich dann vergleichsweise einfach überprüfen, ob einer der dargestellten Sonderfälle vorliegt und dementsprechend eine konkrete Lösung implementieren, ohne dass die generelle Struktur der Binärbäume verändert werden muss. Ansatzpunkte hierbei sind zum einen die korrekte Auswahl eines Terms in der Verifikationsbedingung, zum andern das korrekte Einfügen des bearbeiteten Ausdrucks (siehe `public void mod(SimpleNode simpCond)` in der Klasse `SimpleNode`).

2.3.2 Standardisierung der Baumrepräsentation

Um eine gewisse Standardisierung in der Baumstruktur zu erreichen, wurden verschiedene Methoden in der Klasse `SimpleNode` erstellt:

- `public void standard()`
- `public void negSub()`
- `public void leftFirst(int type, String name)`
- `public void leftAddFirst()`

Die Methode `public void standard()` startet den Standardisierungsvorgang. Von ihr aus werden die drei anderen genannten Methoden aufgerufen.



Die Methode `public void negSub()` ersetzt numerische Knoten mit negativem Wert durch einen Knoten vom Typ `VerifierTreeConstants.JJTNEG` und einen Knoten, der den Betrag des ursprünglichen Knotens darstellt.

Die Methode `public void leftFirst(int type, String name)` führt zur Linksklammerung bei mehreren benachbarten Operationen vom Typ `type` und Namen `name`, das heißt:

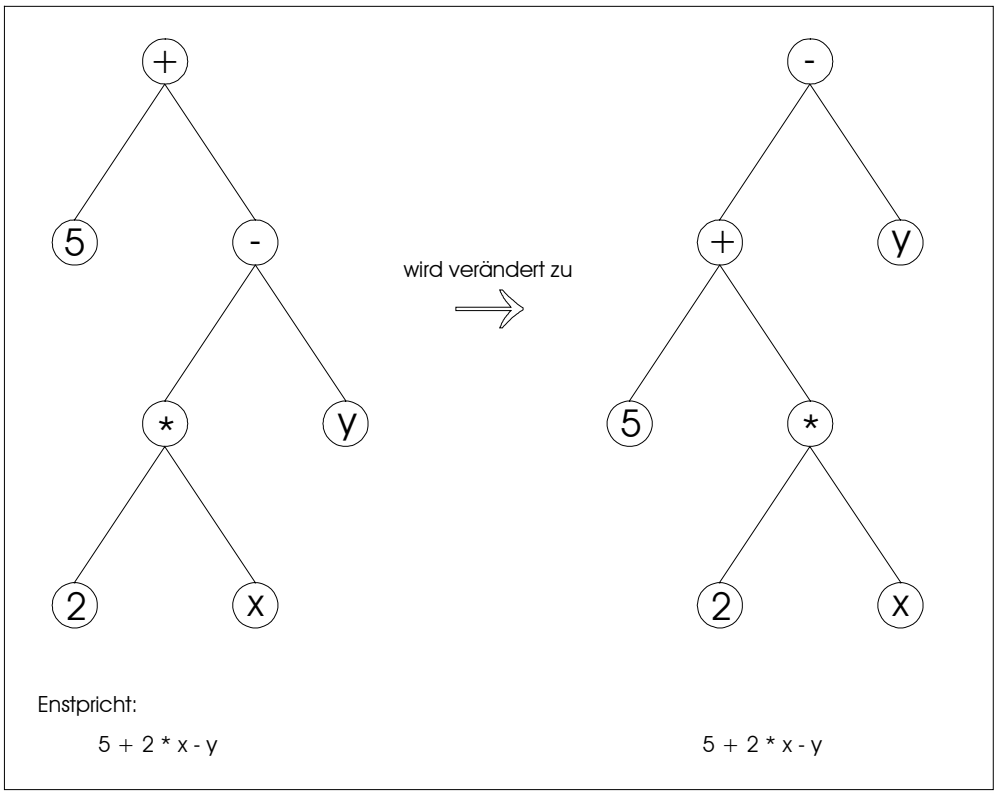
*a * ((b * c) * d)* wird verändert zu *((a * b) * c) * d*.

Diese Methode wird verwendet für

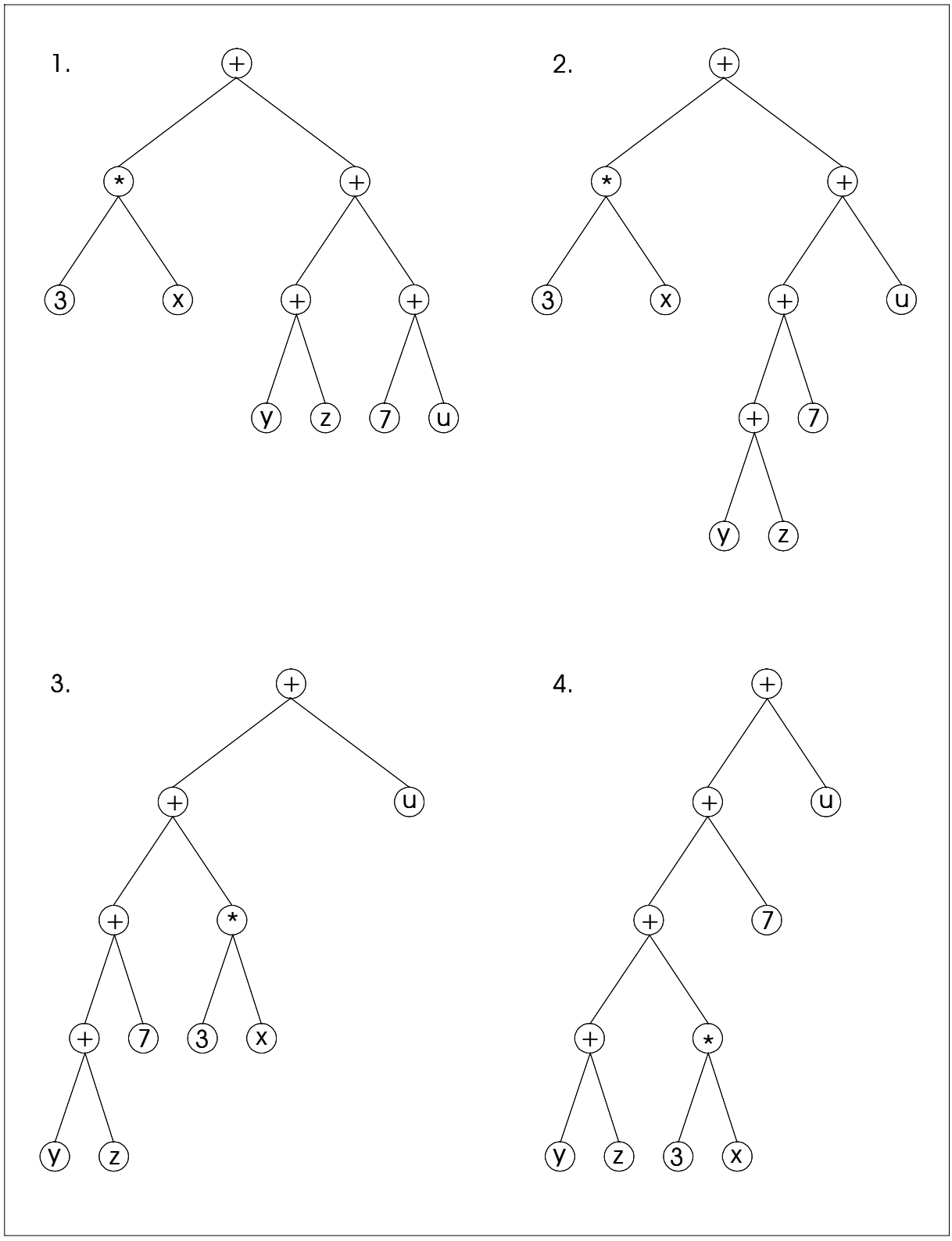
- `type = VerifierTreeConstants.JJTMULT, name = "*"`
- `type = VerifierTreeConstants.JJTAND, name = "&"`
- `type = VerifierTreeConstants.JJTOR, name = "|"`

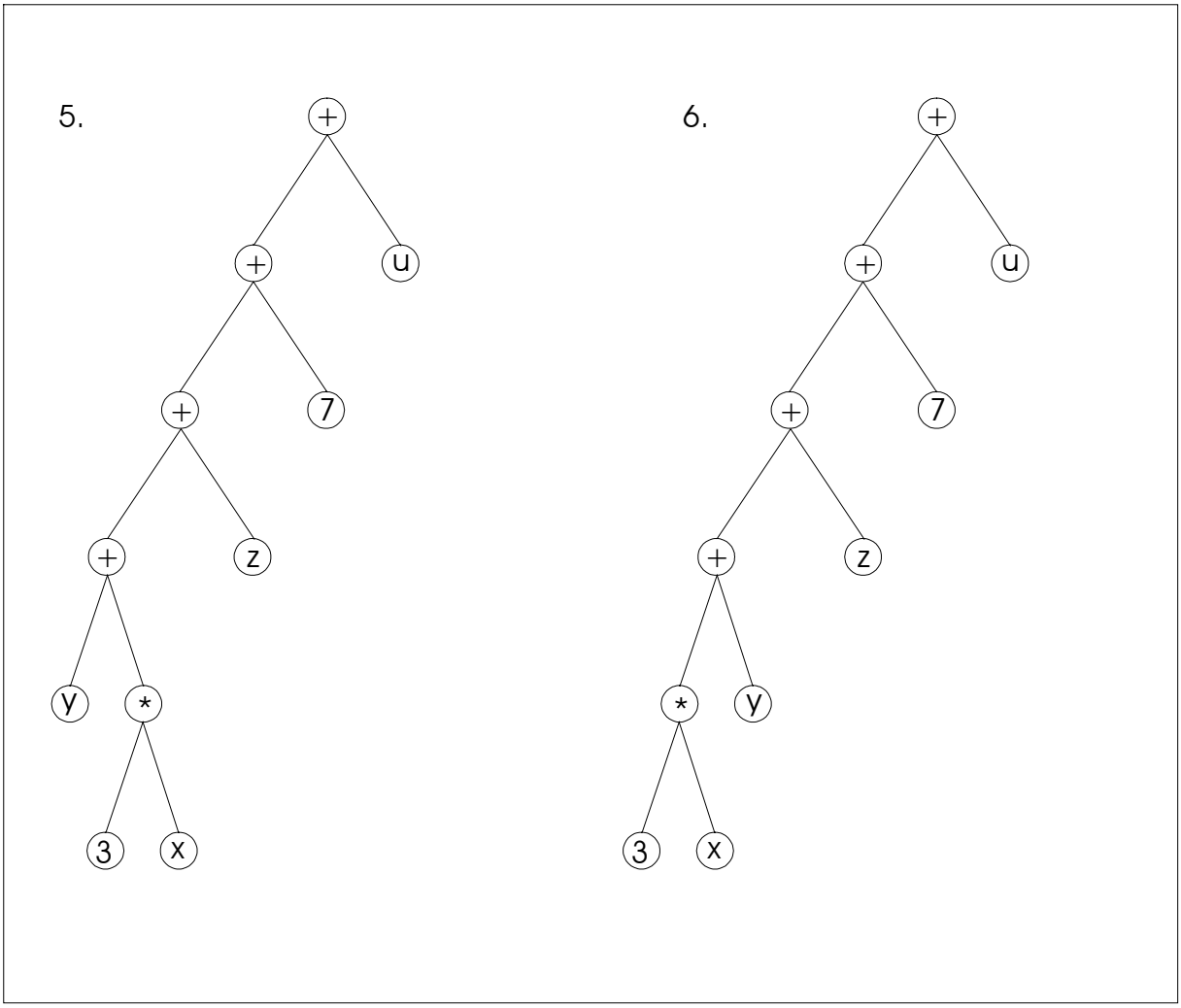
Die Methode `public void leftAddFirst()` führt zur Linksklammerung bei mehreren benachbarten Additionen analog zu `public void leftFirst(int type, String name)`. Weiter wird soweit möglich die Subtraktion in die Standardstruktur integriert, wie das nachfolgende Beispiel veranschaulicht:

Beispiel 1: Integration von Subtraktion



Beispiel 2: Schrittweise Standardisierung einer gegebenen Baumstruktur





Man beachte beim zweiten Beispiel:
Durch die Vertauschung der beiden Teilbäume im zweiten Schritt, die von der Methode `public void inverse()` in der Klasse `SimpleNode` vorgenommen wird, ist eine weitere Veränderung des Baums notwendig, damit sich der durch den Baum dargestellte Ausdruck nicht verändert. Dies geschieht in den Schritten 3 bis 6 durch Anwendung der rekursiven Methode `public void downAdd()`, die ebenfalls in der Klasse `SimpleNode` zu finden ist.

Die Standardisierung von Verifikationsbedingungen findet grundsätzlich immer dann statt, wenn eine neue Verifikationsbedingung dargestellt wird oder wenn eine Verifikations-bedingung mittels der vorhandenen Vereinfachungsregeln verändert wurde.

2.4 Verzeichnis der neuen / veränderten Klassen und Methoden

`AlgSimp, BoolSimp, OrdSimp`

Wie bereits in Abschnitt 2.2.2 dargelegt, wurde die Struktur des Regelwerks gegenüber Version 1.0 verändert. Die Regeln des jeweiligen Typs (algebraisch, boolesch und relational) finden sich nicht mehr in den Klassen `AlgSimp, BoolSimp, OrdSimp`, sondern sind nun in der Klasse `ListOfSimpRules` untergebracht. Dementsprechend entfallen sämtliche Methoden der Gestalt `createRuleXY()`. Es wird lediglich ein Objekt vom Typ `ListOfSimpRules` erstellt, welches durch Aufruf der Methode `public void fill(int type, boolean manual)` aus der Klasse `ListOfSimpRules` mit den Vereinfachungsregeln des jeweiligen Typs gefüllt wird. Dabei steht `type = 0` für algebraische, `type = 1` für relationale und `type = 2` für boolesche Regeln. Im Gegensatz zur Version 1.0 muss bei der Konstruktion eines Elements vom Typ `AlgSimp, BoolSimp` oder `OrdSimp` nun mitangegeben werden, ob die Liste der Regeln für das automatische Beweisverfahren oder für den manuellen Gebrauch bestimmt sind. Dies geschieht durch die Übergabe der booleschen Variablen `manual` im Konstruktor der drei Klassen. Ihr Wert muss ebenfalls beim Aufruf von `public void fill(int type, boolean manual)` übergeben werden. Es kann anschließend ein Array von Elementen des Typs `SimpRule` mit der Länge der so erhaltenen `ListOfSimpRules` erstellt werden. Die Liste wird dann komplett in den Array umkopiert.

```
private int NUMBEROFRULES;

private SimpRule[] rules;

public AlgSimp(boolean manual) {
    ListOfSimpRules listrules = new ListOfSimpRules();
    listrules.fill(0, manual);
    NUMBEROFRULES = listrules.getNumber();
    rules = new SimpRule[NUMBEROFRULES];
    int i = 0;
    while(listrules.hasNext()){
        rules[i++] = listrules.getNext();
    }
}
```

Die Methoden der Klassen wurden vollständig übernommen.

`InternalFrame`

Durch diese Klasse werden interne Fenster innerhalb des Hauptfensters erzeugt. Neben den bereits in Version 1.0 vorhandenen Typen (Programmeingabefenster, diverse Ausgabefenster) wurden zusätzlich zwei weitere Fenstertypen hinzugefügt, die zur manuellen Bearbeitung der Verifikationsbedingungen eingesetzt werden. Wird das Fenster zur manuellen Bearbeitung aus dem automatischen Beweismodus heraus angefordert, so wird der Konstruktor mit dem Parameter `type = BYHAND` aufgerufen.

```
public InternalFrame(int type, int posX, int posY, int width, int height,
                     String title, boolean resizable, boolean closable,
                     boolean maximizable, boolean iconifiable) {
    super(title, resizable, closable, maximizable, iconifiable);
    this.type = type;

    ...

    //  Fenster fuer die manuelle Bearbeitung im automatischen Modus

    else if (type == BYHAND) {
        output = new JTextArea();
        output.setFont(new Font("Courier", Font.PLAIN, 12));
        output.setVisible(true);
        output.setEditable(false);
        JPanel buttonPanel = new JPanel();
        resetButton = new JButton("Reset");
        acceptButton = new JButton("Accept and close");
        buttonPanel.add(resetButton);
        buttonPanel.add(acceptButton);
        getContentPane().add(new JScrollPane(output), BorderLayout.CENTER);
        getContentPane().add(buttonPanel, BorderLayout.SOUTH);
    }

    ...

    setSize(width, height);
    setLocation(posX, posY);
    setVisible(true);
}
```

Im rein manuellen Modus hingegen wird als Parameter `type = NBYHAND` übergeben. Das erzeugte interne Fenster unterscheidet sich von obigem lediglich durch einen zusätzlichen Button *Next*, mit dem die nächste Bedingung aufgerufen werden kann.

```
    ...

    //Fenster für die rein manuelle Bearbeitung der Verifikationsbedingungen

    else if (type == NBYHAND) {
            output = new JTextArea();
            output.setFont(new Font("Courier", Font.PLAIN, 12));
            output.setVisible(true);
            output.setEditable(false);
            //Buttons zum Weitermachen, Zurücksetzen und Schließen des Fensters
            JPanel buttonPanel = new JPanel();
            nextButton = new JButton("Next");
            nextButton.setEnabled(false);
            resetButton = new JButton("Reset");
            acceptButton = new JButton("Accept and close");
            buttonPanel.add(nextButton);
            buttonPanel.add(resetButton);
            buttonPanel.add(acceptButton);
            getContentPane().add(new JScrollPane(output), BorderLayout.CENTER);
            getContentPane().add(buttonPanel, BorderLayout.SOUTH);
        }

    ...
```

Außerdem wurde im Ausgabefenster des automatischen Verifizierers (*VerifierOutput*-Fenster) ein zusätzlicher Button *Manual* hinzugefügt, der in der Klasse `JPV` immer dann aktiviert wird, wenn eine Bedingung nicht vollständig bewiesen werden konnte.

```
else if (type == VERIFOUT) {
    ...
    manualButton = new JButton("Manual");
    manualButton.setEnabled(false);
    ...
    buttonPanel.add(manualButton);
    ...
}
```

Die dabei definierten Buttons können mit Hilfe der Methoden

```
public JButton getNextButton(),
public JButton getNewButton(),
public JButton getManualButton(),
public JButton getResetButton(),
public JButton getAcceptButton()
```

abgefragt werden.

Die Klasse `InternalFrame` wurden neben dieser Ergänzung um vier Felder vom Typ `SimpleNode` erweitert:

```
private SimpleNode mantree
private SimpleNode savetree
private SimpleNode ptree
private SimpleNode unmodtree
```

Diese vier Felder sind standardmäßig mit dem Wert `null` initialisiert. `mantree` enthält im Falle eines *VerifierOutput*-Fensters und eines Fensters zur manuellen Bearbeitung die Verifikationsbedingung, die im Fenster dargestellt wird. Im manuellen Fall wird `mantree` nach jeder Veränderung der Bedingung durch den Anwender angepasst. `savetree` enthält bei einem Fenster zur manuellen Bearbeitung die unveränderte Verifikationsbedingung, mit der das Fenster ursprünglich geöffnet wurde und auf die die Bedingung zurückgesetzt wird, sollte der *Reset*-Button betätigt werden. `ptree` hingegen enthält den vom Anwender zur Vereinfachung ausgewählten Teil der Verifikationsbedingung; dieser wurde aber so abgeändert, dass bei mehreren benachbarten Operationen gleicher Präzedenz nur die zwei tatsächlich ausgewählten enthalten sind (vergleiche Abschnitt 2.3.1). `unmodtree` hingegen enthält den ausgewählten Ausdruck in unveränderter Form.

Alle vier Felder werden durch die Klasse `JPV` über die neu hinzugefügten Methoden

```
public void setNode(SimpleNode tree)
public void setOldNode(SimpleNode tree)
public void setPtree(SimpleNode tree)
public void setUnModTree(SimpleNode tree)
public SimpleNode getNode()
```

```
public SimpleNode getOldNode()
public SimpleNode getPtree()
public SimpleNode getUnModTree()
```

gesetzt bzw. ausgelesen.

IntList

Die neu hinzugekommenes Klasse `IntList` stellt Listen von Elementen des Typs `Integer` zur Verfügung. Sie weist die für Listen typischen Methoden auf und wird u.a. eingesetzt in der Klasse `SimpleNode` in der Methode

```
public IntList dumpExprWithPrecedencesList(int prec, boolean
            parenthesis, boolean isRoot, IntList liste, int lang).
```

JPV

In der Klasse `JPV` wurde die Methode `private void createMenuBar()` grundlegend erweitert. Zunächst wurde im *Options*-Menü der Menüpunkt *Manual Proof* eingefügt.

```
optionsManual = new JCheckBoxMenuItem("Manual Proof");
optionsManual.setMnemonic(KeyEvent.VK_M);
menuOptions.add(optionsManual);
```

Wählt der Anwender den Menüpunkt *Prove* aus dem *Verifier*-Menü, so wird zunächst überprüft, ob der Menüpunkt *Manual Proof* aktiviert ist. Ist dies der Fall, so wird der rein manuelle Beweismodus, andernfalls der automatische Beweismodus gestartet.

```
verifierProve.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ...
        try {
            showMessage("Now Parsing... ");
            try {
                if (optionsManual.isSelected()) {
                    //  rein manueller Beweismodus
                }
                else {
                    //  automatischer Beweismodus
                }
            }
            catch (ParseException err) {
                ...
            }
            catch (TokenMgrError err) {
                ...
            }
        }
        catch (Exception err) {
            ...
        }
    }
});
```

Im rein manuellen Beweismodus wird zunächst ein internes Fenster vom Typ `InternalFrame.NBYHAND` zur manuellen Bearbeitung geöffnet. Es werden die Verifikationsbedingungen erzeugt. Wenn eine erste Verifikationsbedingung existiert, wird diese ausgegeben und intern mittels der in der Klasse `SimpleNode` integrierten Methode `standard()` standardisiert. Weiter wird eine Kopie von ihr unter Verwendung der Methoden

`public void setNode(SimpleNode tree)` und
`public void setNode(SimpleNode tree)`

der Klasse `InternalFrame` dem internen Fenster zugrunde gelegt. Sind noch weitere Verifikationsbedingungen vorhanden, so wird außerdem der *Next*-Button des Fensters aktiv gesetzt.

```
final InternalFrame manual = new InternalFrame(InternalFrame.NBYHAND,
                X_OFFSET * openFrameCount, Y_OFFSET * openFrameCount,
                800, 200, "", true, true, true, true);
final JTextArea manta = manual.getJTextArea();
manual.resetConditionCounter();
...
VerifCondCreater vcc = new VerifCondCreater((SimpleNode) Verifier.jjtree.rootNode());
vcc.createConditions();

...

if (vcc.hasNextCondition()) {
    manual.setCursor(new Cursor(Cursor.WAIT_CURSOR));
    SimpleNode condition = vcc.getNextCondition();
    condition.standard();
    manual.setTitle("Manual proof of " + manual.getConditionCounter()
                + ". condition");
    manual.incConditionCounter();
    SimpleNode simpCond = condition;
    manual.setNode(simpCond.copy());
    manual.setOldNode(simpCond.copy());
    if (vcc.hasNextCondition())
        manual.getNextButton().setEnabled(true);
    manual.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
    ...
}
```

Es werden weiter vier verschiedene *Listener* erzeugt – drei für die vorhandenen Buttons sowie einen `MouseListener`.

```
manta.addMouseListener(new MouseAdapter() {
    ...
});
manual.getNextButton().addActionListener(new ActionListener() {
    ...
});
manual.getAcceptButton().addActionListener(new ActionListener() {
    ...
});
manual.getResetButton().addActionListener(new ActionListener() {
    ...
});
```

Wird der *Reset*-Button betätigt, so wird mit Hilfe der Methoden `getOldNode()` und `setNode(SimpleNode tree)` der Klasse `InternalFrame` die aktuelle auf die ursprüngliche Bedingung zurückgesetzt und die Ausgabe aktualisiert.

Betätigt der Anwender hingegen den *Accept and close*-Button, wird das Fenster geschlossen und `openFrameCount` um den Wert 1 verringert.

Der *Next*-Button liest die nächste Verifikationsbedingung aus, standardisiert sie, setzt sie als Grundlage für das Fenster und gibt sie aus. Weiter wird der *Next*-Button deaktiviert, falls es keine weitere Verifikationsbedingung gibt.

Der `MouseListener` überprüft, ob die rechte Maustaste gedrückt wurde. Ist dies der Fall, so wird derjenige Knoten ermittelt, welcher vom Benutzer markiert wurde. Hierzu wird eine Liste vom Typ `IntList` erstellt. Dabei enthält das *i*-te Element der Liste die Länge des *i*-ten Knotens in der `String`-Repräsentation der Verifikationsbedingung. Diese Liste entsteht durch Aufruf der Methode `dumpExprList()` der Klasse `SimpleNode`.

Anschließend wird durch die Methode `setParents()` der Klasse `SimpleNode` jedem Kindknoten der aktuellen `SimpleNode` und wiederum deren Kindknoten usw. ihr jeweiliger Elternknoten zugeordnet.

Durch Abzählen und dem Vergleich mit der Startposition der Markierung wird ermittelt, das wievielte Zeichen in der `String`-Repräsentation markiert wurde. Das Ergebnis wird unter **`int`** `pos` gespeichert.

```
SimpleNode mantree = manual.getNode();
IntList liste = mantree.dumpExprList();
mantree.setParents();
int pos = -1, tmp, sum = 0, zaehler = 1;
while (liste.hasNext()) {
    tmp = liste.getNext();
    if (sum <= manta.getSelectionStart() - 2
            && sum + tmp > manta.getSelectionStart() - 2)
        pos = zaehler;
    sum += tmp;
    zaehler++;
}
```

Hat `pos` nach der Überprüfung den Wert –1, so wurde vom Anwender keine Markierung vorgenommen. Besitzt `pos` hingegen einen anderen Wert, so ist nun diejenige `SimpleNode` zu ermitteln, die dem ausgewählten Zeichen in der `String`-Repräsentation entspricht.

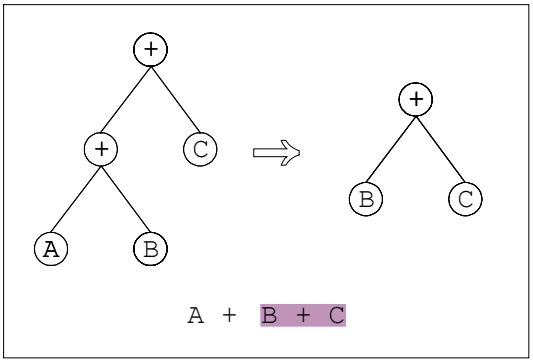
Hierzu wird eine Liste vom Typ `ListOfSimpleNodes` erstellt, in welcher mittels der Methode **`public void`** `createTree(SimpleNode node)` die Knoten der Verifikationsbedingung in der Reihenfolge gespeichert werden, in der sie in der `String`-Repräsentation dargestellt werden. Es genügt nun, das Element an Position `pos` der Liste zu ermitteln. Dieses entspricht dem markierten Ausdruck.

Sollte der so ermittelte Teilbaum `ptree` keine Kindknoten aufweisen, so handelt es sich um einen atomaren Ausdruck, der nicht sinnvoll vereinfacht werden kann. In diesem Fall geht man zu dessen Elternknoten über, was aufgrund des vorherigen Aufrufs von `setParents()` kein Problem darstellt.

```
ListOfSimpleNodes nodelist = new ListOfSimpleNodes();
nodelist.createTree(mantree);
nodelist.resetIterator();
SimpleNode ptree = null;
for (int k = 1; k <= pos; k++)
    if (nodelist.hasNext())
        ptree = nodelist.getNext();
if (ptree.jjtGetNumChildren() == 0)
    ptree = ptree.getParent();
```

Der extrahierte Knoten wird mittels **`public void`** `setUnModTree(SimpleNode tree)` der Klasse `InternalFrame` gespeichert, da er in dieser Form später noch benötigt wird. Er ist aber an dieser Stelle eventuell weiter zu modifizieren. Dies ist dann notwendig, wenn einer der weiter oben beschriebenen Sonderfälle auftritt, bei denen mehrere Operationen gleicher Präferenz nebeneinander stehen und lediglich die beiden direkt benachbarten selektiert werden sollen.

Aufgrund der vorgenommenen Standardisierung der Verifikationsbedingungen (siehe **`public void`** `standard()` in der Klasse `SimpleNode`) kann nun vergleichsweise einfach überprüft werden, ob ein solcher Fall vorliegt. Falls zwei Kindknoten vorhanden sind, müssen dazu nur die Wurzel des ausgewählten Teilbaums sowie der erste der beiden Kindknoten überprüft werden.



Beispielhaft folgt der Teil des Quellcodes, der beim in der Grafik dargestellten Fall greifen würde:

```
if (ptree.getNodeType() == VerifierTreeConstants.JJTADD
        && ptree.getName() == "+"){
    if (ptree.getChild(0).getNodeType() == VerifierTreeConstants.JJTADD
            && ptree.getChild(0).getName() == "+"){
        SimpleNode[] children = {ptree.getChild(0).getChild(1).copy()
                                 , ptree.getChild(1).copy()};
        ptree = new SimpleNode(VerifierTreeConstants.JJTADD, "+", children);
    }
    ...
}
```

Völlig analoge Modifikationen ergeben sich im Fall der Multiplikation sowie bei „`&`“- und „`|`“-Operator.

Im Folgenden muss die Markierung in der Bildschirmausgabe korrekt vorgenommen werden. Dazu wird die `String`-Repräsentation des Teilbaums `ptree` erzeugt und der Text in der `String`-Repräsentation der gesamten Verifikationsbedingung gesucht und markiert. Um sicherzustellen, dass die richtige Stelle markiert wird, auch wenn der Teilausdruck mehrmals in der Bedingung vorkommt, wird ein Vergleich mit der Startposition der vom Benutzer vorgenommenen Markierung durchgeführt. Sonderfall: Stellt die Verifikationsbedingung eine Implikation dar, so wird sie im Fenster in mehreren Zeilen ausgegeben. In diesem Fall wird die `String`-Repräsentation des ausgewählten Ausdrucks isoliert im linken und rechten Teil der Implikation gesucht.

Anschließend wird das PopUp-Menü *Anwendbare Formeln* erzeugt, das eine Reihe von Alternativen für den ausgewählten Term darstellt.

```
JPopupMenu formeln = new JPopupMenu("Anwendbare Formeln");
JLabel label = new JLabel(" Anwendbare Formeln");
String name = label.getFont().getFontName();
int groesse = label.getFont().getSize();
label.setFont(new Font(name , Font.BOLD , groesse));
label.setForeground(Color.RED);
formeln.add(label);
formeln.addSeparator();
```

Standardmäßig wird ein `JmenuItem` erstellt, das mit *Evaluate* beschriftet und mit einem `ActionListener` ausgestattet wird.

```
JMenuItem pcMenueItem = new JMenuItem("Evaluate");
pcMenueItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {

        ...

    }
});
formeln.add(pcMenueItem);
```

Wählt der Anwender dieses Item aus, so soll der markierte Teilausdruck mit dem automatischen Vereinfachungsverfahren so weit wie möglich vereinfacht werden. Damit dies geschehen kann werden Regelsysteme für die drei verschiedenen Regeltypen (algebraisch, relational, boolesch) erstellt, die jeweils nur die Regeln für den automatischen Modus beinhalten. Mit diesen Regelsystemen und dem ausgewählten Teilbaum wird ein Objekt vom Typ `Simplifier` erstellt. Durch Aufruf der Methode `public SimpleNode doWork(boolean output, int numSteps)` erhält man nun das Ergebnis des automatischen Vereinfachungsverfahrens, das der Variablen `SimpleNode simpCond` zugewiesen wird.

Nun muss in der Verifikationsbedingung der markierte Term durch den durch `simpCond` repräsentierten Ausdruck ersetzt werden. Hierzu wird erneut der unmodifizierte durch die Markierung festgelegte Teil der Verifikationsbedingung betrachtet, den man mittels **`public`** `SimpleNode getUnModTree()` der Klasse `InternalFrame` wieder erhält. Durch `ptree.mod(simpCond)` wird anschließend der markierte Ausdruck gemäß der Standardisierung durch den vereinfachten ersetzt. Die derart veränderte Verifikationsbedingung wird standardisiert und gespeichert, die Ausgabe des Fensters wird aktualisiert.

```
algSimp = new AlgSimp(false);
boolSimp = new BoolSimp(false);
ordSimp = new OrdSimp(false);

Simplifier simplifier = new Simplifier(manual.getPtree().copy(),
                                       boolSimp, ordSimp, algSimp, manta);
SimpleNode simpCond = simplifier.doWork(optionsTrace.isSelected(), 0);
SimpleNode mantree = manual.getNode();
mantree.setParents();
SimpleNode ptree = manual.getUnModTree();
ptree.mod(simpCond);
manual.setPtree(null);
manual.setUnModTree(null);
manual.setPos(-1);
mantree.standard();
manual.setNode(mantree);
manta.setText("");
manta.append("\n" + " " + mantree.dumpExpr() + "\n\n");
if (mantree.getNodeType() == VerifierTreeConstants.JJTTRUE)
   showMessage("Proof of " + (manual.getConditionCounter() - 1) +
               ". condition successful!\n");
else if (mantree.getNodeType() == VerifierTreeConstants.JJTFALSE)
   showMessage("Condition simplifies to false!\n");
```

Weiter ist eine Liste derjenigen Regeln zu erstellen, die anwendbar sind, und für jede dieser Regeln ein Item im Menü *Anwendbare Formeln* zu erzeugen.

Zunächst werden wiederum drei Regelsystem für die verschiedenen Regelklassen gebildet, allerdings diesmal unter Berücksichtigung der Regeln für den manuellen Einsatz. Es wird ein Objekt `simplifier` vom Typ `Simplifier` auf der Grundlage des ausgewählten Ausdrucks erstellt. Durch den Aufruf der Methode `getResults()` erhält man ein Array mit Elementen vom Typ `SimpleNode`, die den ausgewählten Teilbaum nach Anwendung sämtlicher passender Regeln darstellen

Nun wird das Array durchlaufen, wobei bei jeder `SimpleNode` überprüft wird, ob deren `String`-Repräsentation mit der einer anderen `SimpleNode` übereinstimmt, die bereits unter kleinerem Index im Array gespeichert ist. Ist dies nicht der Fall, so wird im Menü *Anwendbare Formeln* ein Item erzeugt, das als Titel die `String`-Repräsentation der jeweiligen `SimpleNode` trägt. Es wird weiter ein `ActionListener` gesetzt. Wird ein solches Item vom Anwender ausgewählt, so soll der markierte Teilausdruck durch die zugehörige `SimpleNode` ersetzt werden.

Den vom Anwender ausgewählten Teilbaum erhält man auch hier mittels `public SimpleNode getUnModTree()` der Klasse `InternalFrame`. Durch den Befehl `ptree.mod(results[z])` findet nun letztlich der Austausch des markierten Teilbaums durch den vereinfachten Ausdruck statt.

Die so erhaltene vereinfachte Bedingung muss noch standardisiert und gespeichert werden. Anschließend wird die Ausgabe aktualisiert.

```
algSimp = new AlgSimp(true);
boolSimp = new BoolSimp(true);
ordSimp = new OrdSimp(true);
Simplifier simplifier = new Simplifier(manual.getPtree().copy(),
                                       boolSimp, ordSimp, algSimp, manta);
final SimpleNode[] results = simplifier.getResults();
for (int i = 0; i < results.length; i++) {
    boolean repeated = false;
    for (int j = 0; j < i; j++)
        if (results[i].dumpExpr().equals(results[j].dumpExpr()))
            repeated = true;
    if (!repeated) {
        JMenuItem regelcounter = new JMenuItem(results[i].dumpExpr());
        final int z = i;
        regelcounter.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                SimpleNode mantree = manual.getNode();
                SimpleNode ptree = manual.getUnModTree();
                ptree.mod(results[z]);
                manual.setPtree(null);
                manual.setUnModTree(null);
                mantree.standard();
                manual.setNode(mantree);
                manta.setText("");
                manta.append("\n" + " " + mantree.dumpExpr() + "\n\n");
                if (mantree.getNodeType() == VerifierTreeConstants.JJTTRUE)
                    showMessage("Proof of " + (manual.getConditionCounter() - 1)
                                + ". condition successful!\n");
                else if (mantree.getNodeType() == VerifierTreeConstants.JJTFALSE)
                    showMessage("Condition simplifies to false!\n");
            }
        });
        formeln.add(regelcounter);
    }
}
```

Wählt der Anwender den Menüpunkt *Prove* aus dem *Verifier*-Menü, während der Menüpunkt *Manual Proof* im *Options*-Menü deaktiviert ist, so wird der automatische Beweismodus gestartet. Wie bereits erwähnt wurde im *VerifierOutput*-Fenster, das nach jeder Verifikationsbedingung angibt, ob die Verifizierung geglückt ist, ein zusätzlicher Button mit Aufschrift *Manual* eingefügt. Dieser Button wird immer dann aktiv gesetzt, wenn eine Bedingung nicht vollständig bewiesen werden konnte. Wird dieser Button betätigt, so wird ein Fenster vom Typ `InternalFrame.BYHAND` geöffnet, das den nicht weiter vereinfachbaren Ausdruck darstellt. Bis auf den *Next*-Button, der in diesem Modus nicht integriert wurde, da es ihn bereits im *VerifierOutput*-Fenster gibt, stimmt die Funktionsweise dieses Fensters hundertprozentig mit der des rein manuellen Modus überein. Aus diesem Grund wird an dieser Stelle nicht weiter darauf eingegangen.

ListOfSimpleNodes

In der bestehenden Klasse `ListOfSimpleNodes` wurden zwei weitere Methoden eingefügt.

`public void stepBack()` setzt den Iterator einer Liste um ein Element zurück, falls er nicht schon auf den Anfang der Liste verweist.

`public void createTree(SimpleNode node)` hängt ans Ende einer vorhandene Liste die im Parameter übergebene `SimpleNode` und alle ihre Kindknoten an - und zwar in der Reihenfolge, in der sie in der `String`-Repräsentation der `SimpleNode` auftreten. Dabei wird die `SimpleNode` auf ganz analoge Weise durchwandert wie in:

```
public String dumpExprWithPrecedences(int prec, boolean parenthesis,
                                      boolean isRoot, boolean oneline)
```

```
public void createTree(SimpleNode node) {
   if (node.getNodeType() == VerifierTreeConstants.JJTINTFUNC
         || node.getNodeType() == VerifierTreeConstants.JJTBOOLFUNC) {
      this.addLast(node);
      for (int i = 0; i < node.jjtGetNumChildren(); i++) {
         this.createTree(node.getChild(i));
      }
   }
   else if (node.jjtGetNumChildren() > 0) {
      if (node.jjtGetNumChildren() == 1) {
         this.addLast(node);
         this.createTree(node.getChild(0));
      }
      else if (node.jjtGetNumChildren() == 2) {
         this.createTree(node.getChild(0));
         this.addLast(node);
         this.createTree(node.getChild(1));
      }
   }
   else {
      this.addLast(node);
   }
}
```

ListOfSimpRules

Die Klasse `ListOfSimpRules` stellt eine gewöhnliche Liste von Elementen des Typs `SimpRule` zur Verfügung, die mit den aus `ListOfSimpleNodes` bekannten Methoden zur Handhabung von Listen bearbeitet werden kann.

Außerdem enthält die Klasse die Methode `public void fill(int type, boolean manual)`, welche an das Ende einer Liste sämtliche zur Verfügung stehende Regeln eines bestimmten Typs einfügt. Dabei steht `type = 0` für algebraische, `type = 1` für relationale

und `type = 2` für boolesche Regeln. Hat der boolesche Parameter `manual` den Wert `true`, so werden auch die Regeln für den manuellen Vereinfachungsmodus zugelassen.

```
public void fill(int type, boolean manual){

    //  Deklaration von Hilfsvariablen

    if (type == 0){

    //  algebraische Regeln

    }
    else if (type == 1){

    //  relationale Regeln

    }
    else if (type == 2){

    //  boolesche Regeln

    }
}
```

Die Vereinfachungsregeln werden in der genannten Methode sukzessive erzeugt und an die bestehende Liste angehängt. Durch den veränderten Aufbau des Regelsystems ist es einfacher, neue Regeln hinzuzufügen bzw. bereits enthaltene Regeln zu entfernen. Will man z.B. die Regel für die Kommutativität der Addition entfernen, so genügt es, die entsprechenden Zeilen aus dem Quelltext zu löschen.

```
    /*
     * _X + _Y vereinfacht zu _Y + _X   (MANUAL)
     */
    if (manual){
        //  pattern erzeugen
        children = new SimpleNode[2];
        children[0] = new SimpleNode(VerifierTreeConstants.JJTID, "_X");
        children[1] = new SimpleNode(VerifierTreeConstants.JJTID, "_Y");
        pattern = new SimpleNode(VerifierTreeConstants.JJTADD, "+", children);
        //  simpPattern erzeugen
        children = new SimpleNode[2];
        children[0] = new SimpleNode(VerifierTreeConstants.JJTID, "_Y");
        children[1] = new SimpleNode(VerifierTreeConstants.JJTID, "_X");
        simpPattern = new SimpleNode(VerifierTreeConstants.JJTADD, "+", children);
        this.addLast(new SimpRule(pattern, simpPattern));
    }
```

SimpleNode

An der Klasse `SimpleNode` wurden diverse Veränderungen vorgenommen. Der Methode

```
public String dumpExprWithPrecedences(int prec, boolean parenthesis,
                                      boolean isRoot, boolean oneline)
```

wird nun der zusätzliche Parameter `boolean oneline` übergeben. Hat dieser den Wert `true`, so wird die `String`-Repräsentation in einer Zeile erzwungen. Dies ist notwendig, wenn Teilausdrücke einer Bedingung betrachtet werden und wenn die Einträge für das PopUp-Menü generiert werden.

Weiter gibt es neben der bereits in Version 1.0 vorhandenen Methode `public String dumpExpr()`, die

```
public String dumpExprWithPrecedences(int prec, boolean parenthesis,
                                      boolean isRoot, boolean oneline)
```

mit Parameter `oneline = false` aufruft, die neu hinzugefügte Methode `public String dumpExprOneLine()`, die die `String`-Repräsentation mit Parameter `oneline = true` erstellt.

`public IntList dumpExprList()` liefert eine `Integer`-Liste, wobei der Wert des *i*-ten Elements der Liste der Länge des *i*-ten Elements in der `String`-Repräsentation der `SimpleNode` unter Berücksichtigung von Klammerung und Leerzeichen entspricht. Dazu wird die ebenfalls neue Methode

```
public IntList dumpExprWithPrecedencesList(int prec, boolean
                parenthesis, boolean isRoot, IntList liste, int lang)
```

verwendet. Das Erstellen einer solchen `Integer`-Liste ist notwendig, um von der Anfangs- bzw. Endposition der Markierung in der Ausgabe der `String`-Repräsentation Rückschlüsse auf den vom Anwender ausgewählten Teilausdruck der Verifikationsbedingung ziehen zu können.

Weiter gibt es in `SimpleNode` eine Reihe von Methoden, die der in Abschnitt 2.3.2 beschriebenen Standardisierung der Baumstruktur dienen. Dazu gehören:

```
public void standard()
public void negSub()
public void leftAddFirst()
public void inverse()
public void downAdd()
public void leftFirst(int type, String name)
public void down(int cons, String name)
```

Die Standardisierung einer `SimpleNode` wird durch den Aufruf von `public void standard()` gestartet. Diese Methode dient lediglich dazu, die anderen genannten Methoden auszuführen. Zunächst wird von ihr aus `negSub()` aufgerufen.

```
public void standard(){
    negSub();
    leftAddFirst();
    leftFirst(VerifierTreeConstants.JJTMULT, "*");
    leftFirst(VerifierTreeConstants.JJTAND, "&");
    leftFirst(VerifierTreeConstants.JJTOR, "|");
}
```

Die Methode **`public void`** `negSub()` durchwandert die `SimpleNode` und sorgt dafür, dass kein Knoten vorkommen kann, der einen negativen numerischen Wert darstellt. Sollte ein solcher gefunden werden, so wird er durch einen Knoten vom Typ `VerifierTreeConstants.JJTNEG`, der einem negativen Vorzeichen entspricht, und einem Knoten, der den Betrag des numerischen Wertes darstellt, ersetzt.

```
public void negSub(){
    for (int i = 0; i < jjtGetNumChildren(); i++)
        getChild(i).negSub();
    if (getNodeType() == VerifierTreeConstants.JJTNUM){
        int x = Integer.parseInt(getName());
        if (x < 0){
            setName("" + -x);
            SimpleNode copy = this.copy();
            replaceNode(new SimpleNode(VerifierTreeConstants.JJTNEG, "-",
                        new SimpleNode[]{copy}));
        }
    }
}
```

Anschließend wird die Methode **`public void`** `leftAddFirst()` aufgerufen, welche zu einer standardisierten Linksklammerung sämtlicher benachbarten Additionen und Subtraktionen führt. Die Linksklammerung wird dadurch hergestellt, dass bei jedem Knoten vom Typ `VerifierTreeConstants.JJTADD` überprüft wird, ob der zweite der Kindknoten wiederum vom Typ `VerifierTreeConstants.JJTADD` oder vom Typ `VerifierTreeConstants.JJTSUB` ist. Ist dies der Fall, wird die Baumstruktur angepasst.

Ist z.B. der zweite Kindknoten ebenfalls vom Typ `VerifierTreeConstants.JJTADD` und der erste Kindknoten nicht von diesem Typ, so werden durch Anwendung der Methode `inverse()` die beiden Kindknoten vertauscht. Durch den nachfolgenden Aufruf von `downAdd()` wird sichergestellt, dass sich in der `String`-Repräsentation durch die Standardisierung keine Veränderung (andere Reihenfolge der Summanden) ergibt.

Um sich die konkrete Vorgehensweise zu verdeutlichen, sollte man die Methoden an den in Abschnitt 2.3.2 gegebenen Beispielen nachvollziehen.

```
public void leftFirst(int type, String name){
    for (int i = 0; i < jjtGetNumChildren(); i++)
        getChild(i).leftFirst(type, name);
    if (jjtGetNumChildren() == 2)
        if (getNodeType() == type && getName() == name){
            if (((SimpleNode)children[1]).getNodeType() == type
                    && ((SimpleNode)children[1]).getName() == name){
                if (((SimpleNode)children[0]).getNodeType() != type
                        || ((SimpleNode)children[0]).getName() != name){
                    inverse();
                    down(type, name);
                }
                else {
                    SimpleNode save = ((SimpleNode)children[1]).getChild(0).copy();
                    ((SimpleNode)children[1]).getChild(0).replaceNode(((SimpleNode)children[0]).copy());
                    ((SimpleNode)children[0]).replaceNode(((SimpleNode)children[1]).copy());
                    ((SimpleNode)children[1]).replaceNode(((SimpleNode)children[0]).getChild(1).copy());
                    ((SimpleNode)children[0]).getChild(1).replaceNode(save);
                }
            }
        }
}
```

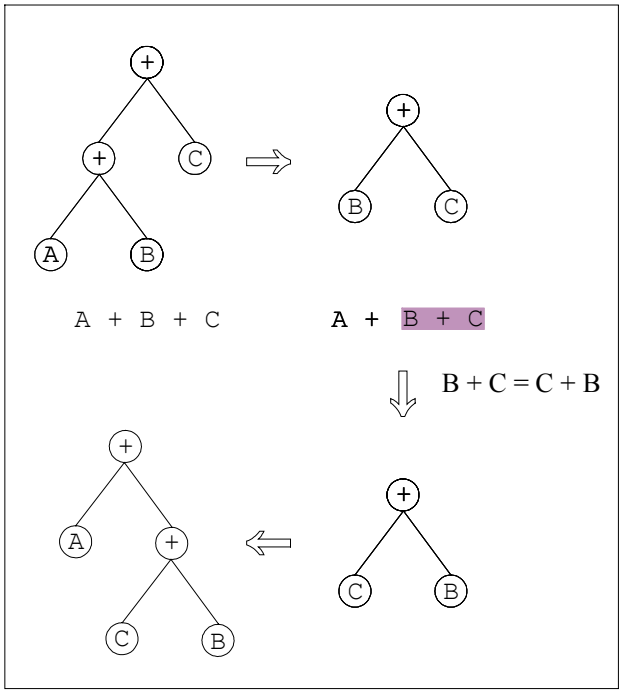
Die Methode `public void leftFirst(int type, String name)` verallgemeinert die soeben betrachtete Methode `public void leftAddFirst()` insofern, dass sie bei mehreren benachbarten Operationen vom Typ `type` und mit Namen `name` eine Linksklammerung durchführt. Dazu benutzt sie auf analoge Weise die Methoden `inverse()` und `public void down(int cons, String name)`, welche die Verallgemeinerung der weiter oben betrachteten Methode `downAdd()` darstellt.

```
public void down(int cons, String name){
    if (((SimpleNode)children[0]).getNodeType() == cons
            && ((SimpleNode)children[0]).getName() == name){
        SimpleNode change = ((SimpleNode) children[0]).getChild(1).copy();
        ((SimpleNode) children[0]).getChild(1).replaceNode(((SimpleNode) children[1]).copy());
        children[1] = change;
        getChild(0).down(cons, name);
    }
    else {
        inverse();
    }
}
```

Es wird eine Linksklammerung für die Multiplikation, den „&“- und „|“-Operator ausgeführt. Die Addition lässt sich nur wegen des Sonderfalls der Subtraktion auf gleicher Präzedenzebene nicht mit diesen allgemeinen Methoden standardisieren.

Bei der letzten neue hinzugekommene Methode der Klasse `SimpleNode` handelt es sich um `public void mod(SimpleNode simpCond)`. Diese Methode ersetzt eine `SimpleNode` durch einen vereinfachten Teilbaum `simpCond`. Sie wird benutzt, um den vom Anwender markierten Teilsausdruck in die ausgewählte Vereinfachung umzuwandeln.

Dabei muss speziell auf die Problematik der standardisierten internen Repräsentation eingegangen werden. Wie in der Grafik dargestellt, genügt es unter Umständen nicht, eine `SimpleNode` durch die neu erhaltene zu ersetzen, sondern es ist eine Anpassung vorzunehmen, die von der Struktur des ehemaligen Baums abhängt.



In `mod(SimpleNode simpCond)` werden die verschiedenen Möglichkeiten überprüft, und es wird der veränderte Teilbaum konstruiert.

Beispielhaft sei an dieser Stelle der Quellcode angegeben, der beim in der Grafik dargestellten Fall greifen würde.

```
public void mod(SimpleNode simpCond) {
    if (((getNodeType() == VerifierTreeConstants.JJTADD
                && getName() == "+")
            || (getNodeType() == VerifierTreeConstants.JJTSUB
                && getName() == "-"))
        && ((getChild(0).getNodeType() == VerifierTreeConstants.JJTADD
                && getChild(0).getName() == "+")
            || (getChild(0).getNodeType() == VerifierTreeConstants.JJTSUB
                && getChild(0).getName() == "-")))
        replaceNode(new SimpleNode(VerifierTreeConstants.JJTADD,"+",
                new SimpleNode[] {getChild(0).getChild(0), simpCond}));

    ...
}
```

Simplifier

Die Klasse `Simplifier` wurde um zwei Methoden ergänzt.

`public SimpleNode[] getResults()` wird im manuellen Beweismodus dazu genutzt, um eine Liste der auf einen markierten Teilausdruck anwendbaren Regeln zu ermitteln. Hierzu wird für jeden Regeltyp (algebraisch, boolesch und relational) ein Regelsystem erstellt, das auch die Regeln für den manuellen Gebrauch enthält:

```
AlgSimp algSimp = new AlgSimp(true);
BoolSimp boolSimp = new BoolSimp(true);
OrdSimp ordSimp = new OrdSimp(true);
```

Anschließend wird jeweils in einer *for*-Schleife für jede Regel eines Regelsystems mittels der Methode `public boolean reduceOne(SimpleNode ccondition, Rules rules, int rule, boolean output)` überprüft, ob ihr Muster auf die übergebene `SimpleNode` passt. Da bei dem Aufruf dieser Methode eine Regel nicht nur auf Anwendbarkeit überprüft, sondern sofern möglich auch tatsächlich angewendet wird, muss hierzu eine Kopie der gegebenen `SimpleNode` erstellt werden.

```
ArrayList resultList = new ArrayList();

for (int i = 0; i < algSimp.getNumberOfRules(); i++) {
    SimpleNode condition2 = condition.copy();
    if (reduceOne(condition2, algSimp, i, false))
        resultList.add(condition2);
}
```

Ist eine Regel anwendbar, so wird die durch sie vereinfachte `SimpleNode` in einer Liste gespeichert. Diese wird nach Überprüfung sämtlicher Regeln in ein Array umkopiert, welches zurückgegeben wird.

Die ebenfalls neu hinzugekommene Methode

```
public boolean reduceOne(SimpleNode ccondition, Rules rules, int
                         rule, boolean output)
```

entspricht im Aufbau völlig der bereits in Version 1.0 enthaltenen Methode

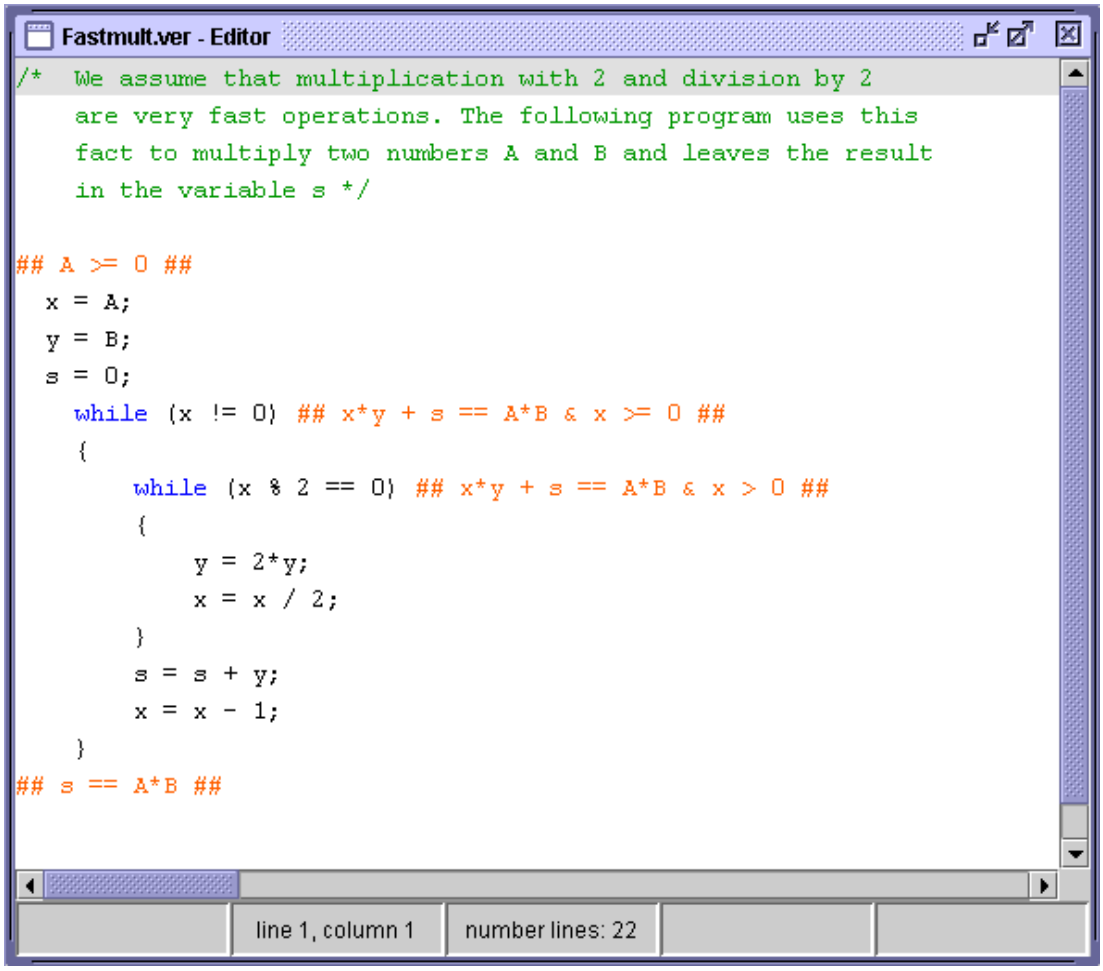
`public boolean reduce(SimpleNode ccondition, Rules rules, boolean output).`

Der einzige Unterschied besteht darin, dass in letzterer Methode sämtliche Regeln des übergebenen Regelsystems auf Anwendbarkeit überprüft werden (und gegebenenfalls auch direkt angewendet werden), während in `public boolean reduceOne` nur eine spezielle Regel des Regelsystems angewendet wird. Um die wievielte Regel es sich dabei handelt, wird als zusätzlicher Parameter `int rule` übergeben.

```
public boolean reduceOne(SimpleNode ccondition, Rules rules, int rule, boolean output) {

    ...

    int i = rule;

    ...

}


public boolean reduce(SimpleNode ccondition, Rules rules, boolean output) {

    ...

    for (int i = start; i < rules.getNumberOfRules(); i++) {

    ...

    }
}
```

3 Testbeispiel

Im Folgenden wird anhand des beiliegenden Testprogramms `fastmult.ver` die Funktionsweise des manuellen Beweismodus erläutert. Hierzu wird nach dem Öffnen des Programms zunächst der automatische Verifikationsalgorithmus im Menü *Verifier* gestartet. Kann dieser eine Bedingung mit den gegebenen Regeln nicht verifizieren, wird die Bedingung im manuellen Modus weiterbearbeitet.



Die ersten vier Verifikationsbedingungen werden vollständig automatisch verifiziert:

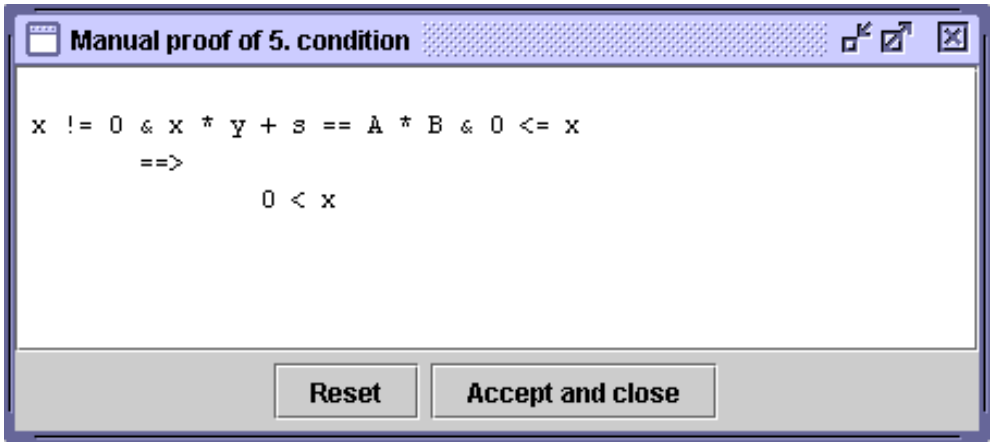
```
1. condition:
A >= 0
   ==>
         A * B + 0 == A * B & A >= 0

Proof of 1. condition successfull!


2. condition:
x * B + 0 == A * B & x >= 0
   ==>
         x * B + 0 == A * B & x >= 0

Proof of 2. condition successfull!
```

```
3. condition:
x * y + 0 == A * B & x >= 0
   ==>
         x * y + 0 == A * B & x >= 0

Proof of 3. condition successfull!


4. condition:
x * y + s == A * B & x >= 0
   ==>
         x * y + s == A * B & x >= 0

Proof of 4. condition successfull!
```

Die fünfte Verifikationsbedingung wird durch das automatischen Verifikationsverfahren zwar vereinfacht, aber nicht vollständig verifiziert werden:

```
5. condition:
x != 0 & x * y + s == A * B & x >= 0
   ==>
         x * y + s == A * B & x > 0

REMAINS TO PROVE:
x != 0 & x * y + s == A * B & 0 <= x
   ==>
         0 < x
```

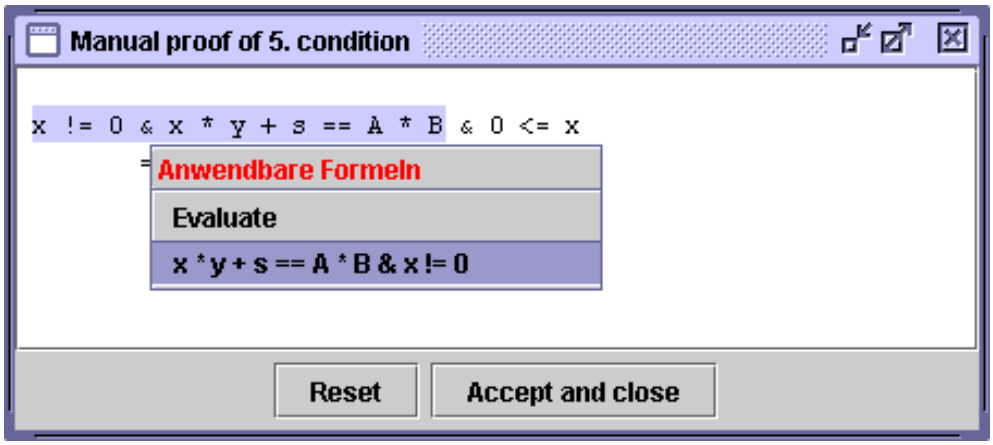
Der Button *Manual*, der bei den vorherigen Bedingungen jeweils deaktiviert war, ist nun aktiv. Wird er betätigt, so öffnet sich das Fenster *Manual proof of 5. condition*. Es enthält die vereinfachte Bedingung, die automatisch nicht weiter reduziert werden konnte.



Betrachtet man in der ersten Zeile den ersten und letzten der drei mit `&` verknüpften booleschen Ausdrücke, so folgt aus `x != 0 & 0 <= x` natürlich direkt `0 < x`. Im automatischen Modus kann die Bedingung aber nicht verifiziert werden, da die Kommutativität von `&` dort nicht möglich ist. Dieses Problem lässt sich manuell beheben. Zunächst kommutiert man die `&`, so dass `x != 0` und `0 <= x` benachbarte Bedingungen sind. Hierzu markiere das erste `&` und drücke die rechte Maustaste.

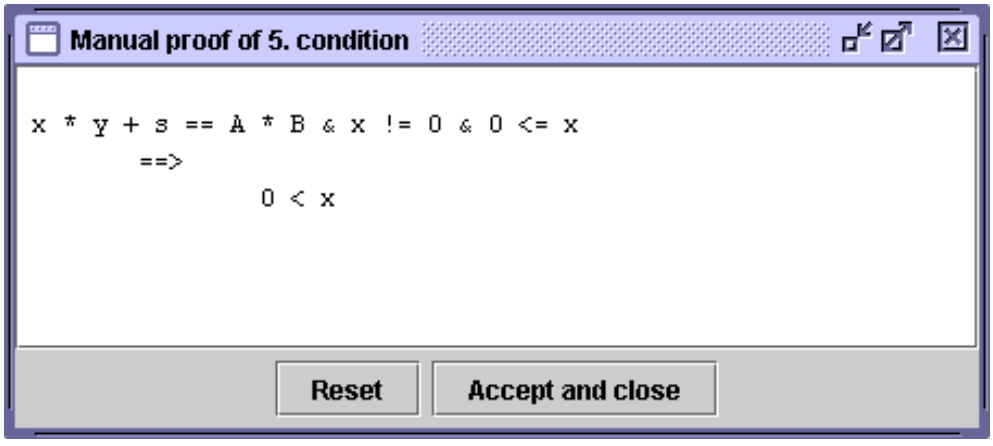
Es öffnet sich das PopUp-Menü *Anwendbare Formeln*, das in diesem Fall zwei Einträge aufweist: Zum einen den Standardeintrag *Evaluate*, der die Möglichkeit anbietet, den markierten Ausdruck so weit wie möglich durch das automatische Vereinfachungsverfahren zu reduzieren, zum anderen ein Vertauschen der beiden betroffenen Terme:

```
x * y + s == A * B & x != 0
```

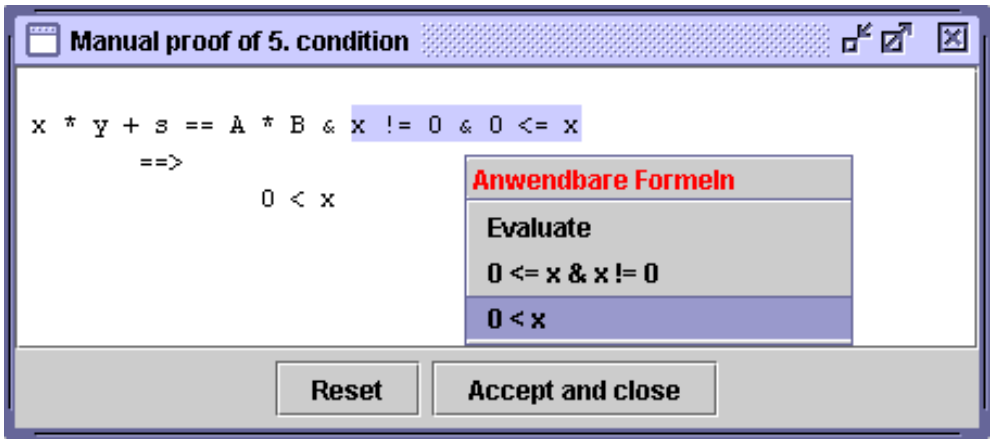


Wählt der Anwender den Eintrag *Evaluate*, so tritt keine Veränderung an der Bedingung ein, d.h. eine automatische Vereinfachung war nicht möglich. Dies ist in diesem Fall nicht weiter verwunderlich, da eine derartige Veränderung bereits bei der komplett automatischen Bearbeitung hätte auftreten müssen, bevor keine Regel mehr anwendbar war.

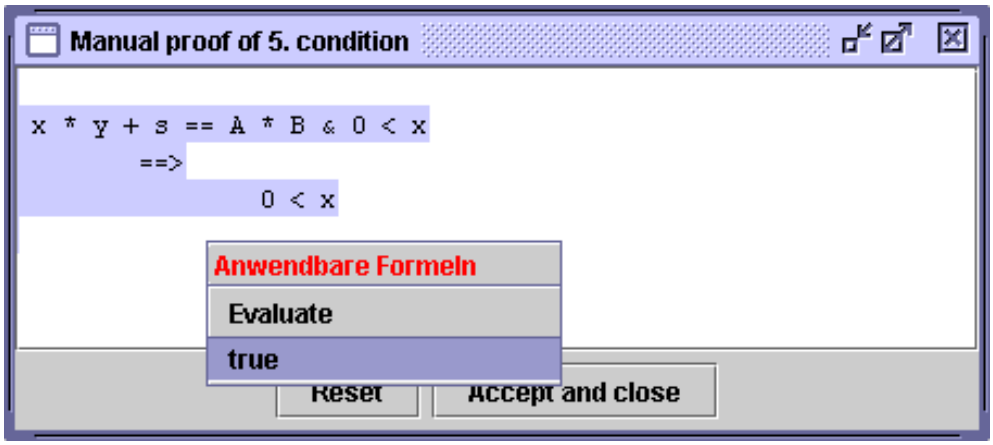
Wählt man hingegen den zweiten Eintrag, so kommutieren wie angegeben die ersten beiden booleschen Ausdrücke:



Nun lassen sich die beiden hinteren Ausdrücke `x != 0 & 0 <= x` zusammenfassen. Markiert man das verknüpfende `&` und drückt die rechte Maustaste, so öffnet sich erneut ein PopUp-Menü, das neben dem obligatorischen Eintrag *Evaluate* die Möglichkeit anbietet, entweder zu kommutieren oder `x != 0 & 0 <= x` zu `0 < x` zu vereinfachen.



Wählt man den letzten Eintrag, so ist es offensichtlich, dass die Bedingung wahr ist. Durch ein Markieren der Implikation (und damit der gesamten Bedingung) und ein erneutes Aufrufen des PopUp-Menüs mit der rechten Maustaste, lässt sich die gesamte Bedingung zu *true* vereinfach, d.h. sie wurde verifiziert.



Durch das Betätigen des Buttons *Accept and close* kann nun das Fenster zur manuellen Verifizierung geschlossen werden, und man kann mit den restlichen Verifikations-bedingungen auf analoge Weise fortfahren.

4 Zusammenfassung

Im Rahmen eines semesterbeleitenden Informatik-Praktikum für Fortgeschrittene an der Philipps-Universität Marburg wurden auf der Basis der bestehenden Version 1.0 des *Java Program Verifier*, entwickelt von Markus Hampel und Manuel Werner, einige Veränderungen und Ergänzungen vorgenommen. Der Schwerpunkt lag dabei in der Implementierung eines speziellen Modus zur manuellen Vereinfachung der erstellten Verifikationsbedingungen und den hiermit verbundenen strukturellen Veränderungen. Es wurde zum einen ein rein manueller Modus entwickelt, der es dem Anwender ermöglicht, Verifikationsbedingungen selbständig zu bearbeiten. Zum anderen wurde im bestehenden automatischen Vereinfachungsalgorithmus eine Zusatzfunktion integriert, die es ermöglicht, Bedingungen manuell zu bearbeiten, die durch das bestehende automatische Verfahren nicht vollständig verifiziert werden können.

Des Weiteren wurde das vorhandene Regelwerk an algebraischen, booleschen und relationalen Vereinfachungsregeln um einige Regeln ergänzt, wobei speziell Regeln hinzugefügt wurden, die ausschließlich für den neu entwickelten manuellen Verifikationsmodus geeignet sind, da ihre generelle Anwendung problematisch ist bzw. nicht zwangsläufig sinnvoll erscheint.

Literatur- und Quellenverzeichnis

[1] Gumm, Prof. Dr. H. P.: New Paltz Program Verifier (NPPV), http://www.mathematik.uni-marburg.de/~gumm/NPPV/nppv.html

[2] Gumm, Prof. Dr. H. P.: Generating algebraic laws from imperative Programs. In: I. Rival (ed.) Proceedings of Ordal '96. http://www.mathematik.uni-marburg.de/~gumm/Papers/nppv.ps

[3] Programmiersprache Java: http://java.sun.com

[4] Java Compiler Compiler™ (JavaCC™) – The Java Parser Generator https://javacc.dev.java.net

[5] jEdit Syntax Package: http://sourceforge.net/projects/jedit-syntax

[6] Programmdokumentation des JPV (generiert mit javadoc)

Anhang A: Liste der neuen Vereinfachungsregeln

Algebraische Vereinfachungsregeln:

43. _X + _Y vereinfacht zu _Y + _X

44. _X * _Y vereinfacht zu _Y * _X

45. _X * _Y vereinfacht zu _Y * _X

46. -(_X + _Y) vereinfacht zu -_X + (-_Y)

47. _X - _Y vereinfacht zu _X + (-_Y)

48. -_X + _Y vereinfacht zu _Y - _X

49. (_X * _Y) + (_X * _Z) vereinfacht zu _X * (_Y + _Z)

50. (_X * _Y) - (_X * _Z) vereinfacht zu _X * (_Y - _Z)

51. (_X * _Z) + (_Y * _Z) vereinfacht zu (_X + _Y) * _Z

52. _X + (_X * _Z) vereinfacht zu _X * (int(1) + _Z)

53. -_X + (-_Y) vereinfacht zu -(_X + _Y)

54. -_X - _Y vereinfacht zu -(_X + _Y)

55. _X + (_Z * _X) vereinfacht zu (int(1) + _Z) * _X

56. (_Z * _X) + _X vereinfacht zu (int(1) + _Z) * _X

57. _X + (_Z * _X) vereinfacht zu (int(1) + _Z) * _X

58. (_X * _Z) + _X vereinfacht zu (int(1) + _Z) * _X

59. _X % _Y vereinfacht zu _X - (_X / _Y) * _Y

Boolesche Vereinfachungsregeln:

60. _X >= _Y & _X > _Y vereinfacht zu _X > _Y

61. _X > _Y & _X >= _Y vereinfacht zu _X > _Y

62. _X < _Y ==> _X + int(1) <= _Y vereinfacht zu true

63. _X > _Y ==> _X - int(1) >= _Y vereinfacht zu true

64. X + int(Z) <= _Y ==> _X < _Y vereinfacht zu true,
falls _Z > int(0)

65. X - int(Z) <= _Y ==> _X < _Y vereinfacht zu true,
falls _Z < int(0)

66. X + int(Z) >= _Y ==> _X > _Y vereinfacht zu true,
falls _Z < int(0)

67. X - int(Z) >= _Y ==> _X > _Y,
falls _Z > int(0)

68. _X & _Y vereinfacht zu _Y & _X

69. _X | _Y vereinfacht zu _Y | _X

Relationale Vereinfachungsregeln:

37. _X <= _Y & _X != _Y vereinfacht zu _X < _Y

38. _X != _Y & _X <= _Y vereinfacht zu _X < _Y

39. _X <= _Y & _Y != _X vereinfacht zu _X < _Y

40. _Y != _X & _X <= _Y vereinfacht zu _X < _Y

41. _X >= _Y & _X != _Y vereinfacht zu _X > _Y

42. _X != _Y & _X >= _Y vereinfacht zu _X > _Y

43. _X >= _Y & _Y != _X vereinfacht zu _X > _Y

44. _Y != _X & _X >= _Y vereinfacht zu _X > _Y

45. _Y < _X vereinfacht zu _X > _Y

46. _Y <= _X vereinfacht zu _X >= _Y

47. int(_X) > int(_Y) vereinfacht zu true, falls _X > _Y

48. _X > _X vereinfacht zu false

49. _X >= _X vereinfacht zu true

50. int(_X) >= int(_Y) vereinfacht zu true, falls _X >= _Y

51. int(_X) >= int(_Y) vereinfacht zu false, falls _X < _Y

52. _X == _Y vereinfacht _Y == _X

53. _X >= (_U - _V) vereinfacht zu (_X + _V) >= _U

54. (_U - _V) >= _X vereinfacht zu _U >= (_X + _V)

55. (_X + _Y) >= _X vereinfacht zu _Y >= int(0)

56. _X >= _X + _Y vereinfacht zu int(0) >= _Y

57. _X > (_U - _V) vereinfacht zu (_X + _V) > _U

58. (_U - _V) > _X vereinfacht zu _U > (_X + _V)

59. (_X + _Y) > _X vereinfacht zu _Y > int(0)

60. _X > _X + _Y vereinfacht zu int(0) > _Y

61. _X + _Y == _Z vereinfacht zu _X == _Z -_Y

62. _X - _Y == _Z vereinfacht zu _X == _Z +_Y

63. _X + _Y == _Z vereinfacht zu _Y == _Z -_X

64. _X - _Y == _Z vereinfacht zu -_Y == _Z -_X

65. _Z == _X + _Y vereinfacht zu _Z -_Y == _X

66. _Z == _X - _Y vereinfacht zu _Z +_Y == _X

67. _Z == _X + _Y vereinfacht zu _Z -_X == _Y

68. _Z == _X - _Y vereinfacht zu _Z -_X == -_Y

69. _X + _Y <= _Z vereinfacht zu _X <= _Z -_Y

70. _X - _Y <= _Z vereinfacht zu _X <= _Z +_Y

71. _X + _Y <= _Z vereinfacht zu _Y <= _Z -_X

72. _X - _Y <= _Z vereinfacht zu -_Y <= _Z -_X

73. _Z <= _X + _Y vereinfacht zu _Z -_Y <= _X

74. _Z <= _X + _Y vereinfacht zu _Z -_X <= _Y

75. _Z <= _X - _Y vereinfacht zu _Z -_X <= -_Y

76. _X + _Y => _Z vereinfacht zu _X => _Z -_Y

77. _X - _Y => _Z vereinfacht zu _X => _Z +_Y

78. _X + _Y => _Z vereinfacht zu _Y => _Z -_X

79. _X - _Y => _Z vereinfacht zu -_Y => _Z -_X

80. _Z => _X + _Y vereinfacht zu _Z -_Y => _X

81. _Z => _X - _Y vereinfacht zu _Z +_Y => _X

82. _Z => _X + _Y vereinfacht zu _Z -_X => _Y

83. _Z => _X - _Y vereinfacht zu _Z -_X => -_Y

84. _X + _Y < _Z vereinfacht zu _X < _Z -_Y

85. _X - _Y < _Z vereinfacht zu _X < _Z +_Y

86. _X + _Y < _Z vereinfacht zu _Y < _Z -_X

87. _X - _Y < _Z vereinfacht zu -_Y < _Z -_X

88. _Z < _X + _Y vereinfacht zu _Z -_Y < _X

89. _Z < _X - _Y vereinfacht zu _Z +_Y < _X

90. _Z < _X + _Y vereinfacht zu _Z -_X < _Y

91. _Z < _X - _Y vereinfacht zu _Z -_X < -_Y

92. _X + _Y > _Z vereinfacht zu _X > _Z -_Y

93. _X - _Y > _Z vereinfacht zu _X > _Z +_Y

94. _X + _Y > _Z vereinfacht zu _Y > _Z -_X

95. _X - _Y > _Z vereinfacht zu -_Y > _Z -_X

96. _Z > _X + _Y vereinfacht zu _Z -_Y > _X

97. _Z > _X - _Y vereinfacht zu _Z +_Y > _X

98. _Z > _X + _Y vereinfacht zu _Z -_X > _Y

99. _Z > _X - _Y vereinfacht zu _Z -_X > -_Y