Universität Bremen
Fachbereich 3
Sommersemester 2005

AG Rechnerarchitektur
Seminar „Methoden der Verifikation“
Arman Allahyari-Abhari

Ausarbeitung zum Thema

QUAFFLE: Ein Beweiser für quantifizierte Boolesche Ausdrücke

Einleitung

In den letzten Jahren wurden weitreichende Forschungen im Bereich des Erfüllbarkeitsproblems (kurz: *SAT*) betrieben. Ein reales Problem (z.B. ein Schaltkreis) wird in eine Boolesche Formel transferiert und in ihre konjunktive Normalform *(CNF)* überführt. Der resultierende Boolesche Ausdruck wird dann daraufhin analysiert, ob eine Variablenbelegung existiert, so dass die Formel zum Funktionswert 1 evaluiert. Aktuelle SAT-Solver basieren auf dem Algorithmus von *Davis, Logemann und Loveland (DLL)* [3] und beherrschen Techniken wie das konfliktbasierte Lernen [5], verbesserte Implememtierung der *Boolean Constraint Propagation* [6] oder ausgefeilte Heuristiken zur Variablenentscheidung [4, 6].
Quantifizierte Boolesche Formeln *(QBF)* sind Boolesche Ausdrücke, die um Quantoren erweitert sind. Auch hier stellt sich die Frage, ob diese quantifizierten Ausdrücke zum Funktionswert 1 evaluieren können oder nicht. Aufgrund der engen Verwandschaft zum klassischen Erfüllbarkeitsproblem werden in dieser Ausarbeitung ersteinmal algorithmische Ansätze zur Lösung von *SAT* erläutert. Diese Ansätze werden dann auf das *QBF*-Problem adaptiert, so dass sie auch hier Anwendung finden können. Da konfliktbasiertes Lernen im Bereich *SAT* auch heute noch als wichtigste Erweiterung des *DLL*-Algorithmus angesehen wird, liegt der Schwerpunkt dieser Ausarbeitung in den Möglichkeiten des Lernens in quantifizierten Booleschen Formeln. Dieses *Learning* wird anhand der Vorgehensweise des *QBF*-Solvers *Quaffle* dargestellt, welcher momentan der effizienteste *QBF*-Solver ist.
Die Erfüllbarkeit quantifizierter Boolescher Formeln erfordert neben einer erfüllenden Variablenbelegung auch die Erfüllung beider Zweige von allquantifizierten Variablen. Aufgrund dessen wird neben konfliktbasiertem Lernen auch eine Methode vorgestellt, die sich erfüllende Variablenbelegungen mit nichterfüllten allquantifizierten Variablen merkt und so auch erfüllende Zweige des Suchbaums abschneidet.

Im Folgenden wird vorerst auf die Definitionen und Begriffe hinsichtlich *SAT* eingegangen. Auf diese Grundlagen aufbauend wird dann in der Algorithmus von *Davis, Logemann* und *Loveland* erklärt und als letzter einführender Abschnitt in das SAT-Problem werden kurz die wichtigsten Verbesserungen des Basisalgorithmus erläutert. Nachdem die grundlegenden Konzepte für *SAT* behandelt wurden, werden die für *QBF* nötigen zusätzlichen Terminologien und der auf *QBF* adaptierte Basisalgorithmus sowie die zugehörigen erweiterten Regeln für Konflikte, Implikation und Erfüllung dargelegt. Der Hauptteil dieser Ausarbeitung konzentriert sich auf das Lernen durch Konflikte im Laufe des Suchprozesses und das in dem *QBF*-Solver *Quaffle* neu hinzugekommene Lernen durch erfüllende Variablenbelegungen. Schließlich wird versucht diese Konzepte anhand von experimentellen Benchmarks zu bewerten um so sinnvolle Einsatzgebiete für die einzelnen Ansätze erfassen zu können.

Terminologien für SAT

Eine Boolesche Formel *f* liegt in ihrer konjunktiven Normalform (*CNF*) vor, wenn sie die Form $C_1 C_2 ... C_{m-1} C_m$ hat, also eine Konjunktion von mehreren C_i ist. Jedes C_i wird als Klausel bezeichnet und ist eine Disjunktion von Literalen x_j , sie liegt also in der Form $x_{j_1} + x_{j_2} + ... + x_{j_k}$ vor. Ein Literal ist eine Variable in positiver oder negativer Phase, also x_j oder $\bar{x}_j$.

Beispiel für eine CNF: $(x_1 + x_3 + \bar{x}_4)(\bar{x}_1 + x_2 + \bar{x}_3)(\bar{x}_1 + \bar{x}_2)$

Gegeben sei nun eine Boolesche Formel *f* in konjunkiver Normalform. Wie oben bereits erwähnt stellt das *SAT*-Problem die Frage, ob es eine erfüllende Belegung für diese Boolesche Formel gibt. Im Jahre 1971 bewies *Stephen A. Cook* die *NP-Vollständigkeit* des Erfüllbarkeitsproblems [7]. Es ist somit bisher kein Algorithmus bekannt, der die Lösung des Problems in polynomieller Zeit liefert, so dass die Laufzeit für größere Probleme sehr stark ansteigt und ohne effiziente Algorithmen für viele SAT-Instanzen keine Lösung in akzeptabler Zeit gefunden werden kann.

Der erste Ansatz zur Lösung von *SAT*-Problemen war der *Davis-Putnam*-Algorithmus von 1960. Dieser Algorithmus trägt das Problem mit sich, eine äußerst hohe Speicherkomplexität aufzuweisen, so dass er nicht praktikabel ist. Der *Davis-Putnam*-Algorithmus löst Instanzen durch Anwendung von *Resolution*, welche weiter unten zur Generierung von Konfliktklauseln in *SAT*- und *QBF*-Instanzen noch immer eingesetzt wird.

Der DLL-Algorithmus für SAT

Heutige *SAT*-Solver basieren auf dem 1962 publizierten *Davis-Logemann-Loveland*-Algorithmus (*DLL*). Der Algorithmus geht folgendermaßen vor. Es wird eine Variable entschieden, d.h. es wird entweder das positive oder das negative Literal der ausgesuchten Variable mit dem Wert 1 belegt. Welche Variable bzw. welches ihrer Literale entschieden wird, ist Aufgabe von Heuristiken. Nach der Entscheidung werden aus der Entscheidung folgende Implikationen anhand der *Unit Clause Rule* (oft auch als *Implication Rule* bezeichnet) errechnet und durchgeführt. Die *Unit Clause Rule* besagt folgendes: Sind in einer Klausel allen Literalen außer einem der Wert 0 zugewiesen, so muss das eine freie Literal den Wert 1 zugewiesen bekommen, da sonst die Klausel unerfüllt wäre und somit die gesamte Formel *f* unerfüllbar wäre. Diese Regel wird solange iterativ auf die Klauseln angewendet (iterierte Anwendung der *Unit Clause Rule* wird als *Boolean Constraint Propagation* bezeichnet), bis entweder keine Implikationen mehr durchgeführt werden können oder ein Konflikt auftritt. Im ersten Fall würde dann die nächste Variable entschieden werden und dann erneut impliziert werden. Im zweiten Fall, also beim Auftreten eines Konflikts, ist mindestens eine Klausel unerfüllt, so dass der aktuelle Zweig im Suchbaum ebenfalls unerfüllbar ist. Da nur auf eine Art und Weise anhand der *Unit Clause Rule* impliziert wird, kann der Fehler nur anhand einer falschen Variablenentscheidung entstanden sein, so dass alle Implikationen seit der letzten Entscheidung rückgängig gemacht werden müssen und die Entscheidung der letzten Variable auf das Komplement des letzten entschiedenen Literals fallen muss. Dieser Vorgang wird als *chronologisches Backtracking* bezeichnet. Wird auf die erste Entscheidung zurückgesprungen und das Komplementliteral gewählt und auch dieser Zweig ist unerfüllbar, dann gibt es keine erfüllende Variablenbelegung für diese Formel *f*. Sind alle Klauseln erfüllt, so ist die *SAT*-Instanz erfüllbar. In Abbildung 1 ist ein vereinfachter Pseudocode des *DLL*-Algorithmus dargestellt.

```
while(1) {
  decide_next_branch();                    //Variablenentscheidung
  while (true) {
    status = deduce();                     //Implikationen
    if (status == CONFLICT) {
      blevel = analyze_conflict();         //Konfliktbehandlung
      if (blevel == 0)
        return UNSAT;
      else backtrack(blevel);
    }
    else break;
  }                                        //nächste Variable entscheiden
}
```

Abbildung 1: Pseudocode des *Davis-Logemann-Loveland*-Algorithmus

Dieser Algorithmus bedarf deutlich weniger Speicher als der Davis-Putnam-Algorithmus und bringt als neuen Flaschenhals die CPU eines Rechners mit sich. Der reine DLL-Algorithmus ist trotzdem für aktuelle SAT-Probleme noch deutlich zu langsam, so dass Verbesserungen an dieser Stelle zwingend erforderlich sind.

Erweiterungen des DLL-Algorithmus

Ein großer Vorteil des *DLL*-Algorithmus ist, dass er sehr modular aufgebaut ist. Die Modularität ermöglicht es, die einzelnen Teile des Algorithmus getrennt zu betrachten und so mögliche Verbesserung in den einzelnen Modulen sehr leicht einzupflegen.
Die erste wesentliche Verbesserung des *DLL*-Algorithmus liegt in dem Modul der Konfliktdiagnose bzw. Konfliktbehandlung. Die Neuerungen umfassen *nicht-chronologisches Backtracking* und natürlich das *Learning* durch Konflikte. Die Implementierung dieser Neuerungen wurde im *SAT*-Solver *GRASP* [5] realisiert.
Durch *nichtchronologisches Backtracking* wird es möglich, bei einem Konflikt nicht nur zur letzten Entscheidung im Suchbaum zu springen, sondern anhand des Implikationsgraphen herauszufinden, welche Entscheidung den Konflikt verursacht hat und dann den Rücksprung direkt zu dieser Stelle im Suchbaum zu machen, unabhängig davon, wie weit die Entscheidung vom aktuellen Zweig entfernt ist.
Das *Learning* wird als die wichtigste Erneuerung der letzten zehn Jahre in der Forschung hinsichtlich des Erfüllbarkeitsproblems angesehen. Die Klauseln, welche an einem Konflikt beteiligt sind, bieten die Möglichkeit, dass aus Ihnen mithilfe von *Resolution* eine neue Klausel generiert wird. Diese neue Klausel wird in die Klauselmenge hinzugefügt und soll verhindern, dass im weiteren Suchprozess durch Konflikte bekannte unerfüllbare Zweige des

Suchbaums durchsucht werden. Dieser Prozess des Lernens wird in dieser Ausarbeitung für *QBF* adaptiert und erweitert.
Des Weiteren existieren noch andere Verbesserungen, wie eine optimierte *Boolean Constraint Propagation* anhand des *Two Literal Watching Schemes* , ausgefeilte Heuristiken zur Variablenentscheidung und umfassendes Management der Klauseldatenbank wie sie in den *SAT*-Solvern *Chaff* [4] und *BerkMin* [6] implementiert wurden. Diese sind jedoch nicht Gegenstand dieser Ausarbeitung, so dass für weitere Informationen auf die Originalarbeiten dieser verwiesen wird.

Quaffle – Ein QBF-Solver nach Stand der Technik

Der im Jahre 2002 veröffentlichte *QBF*-Solver *Quaffle* bedient sich einiger bekannter Techniken aus dem bereits sehr weit erforschten Gebiet des Erfüllbarkeitsproblems. Der Schwerpunkt dieses *QBF*-Solvers liegt in effizienten Techniken, Sachverhalte aus gemachten Fehlern im Suchprozess aufzuzeichnen und so zu „lernen“ denselben Fehler nicht noch einmal zu machen. Um dies zu ermöglichen werden einerseits auftretende Konflikte während des Suchprozesses in Form von neuen Klauseln in die Datenstruktur eingefügt und andererseits auch durch ein neues Konzept erfüllende Belegungen gelernt, die für die gegebene *QBF* jedoch keine Lösung darstellen. Um diese Sachverhalte verstehen zu können werden nun die Grundlagen von *QBF*s diskutiert.

Terminologien für QBF

Eine quantifizierte Boolesche Formel (*QBF*) ist eine Boolesche Formel, die um Quantoren erweitert ist und die Form $Q_1 x_1 ..... Q_n x_n \ \varphi$ aufweist, wobei φ die aussagenlogische Formel repräsentiert und jedes Q_i die zugehörige Variable x_i quantifiziert. Jedes Q_i ist entweder der Existenzquantor $\exists$ oder der Allquantor $\forall$. Für existenzquantifizierte Variablen ($\exists x_i$) muss die Formel φ mit nur einer der beiden Phasen dieser Variablen erfüllbar sein, wohingegen die Formel φ bei allquantifizierte Variablen mit beiden Phasen der Variablen erfüllbar sein muss. Die Formel φ muss also bei einer allquantifizierten Variablen ($\forall x_j$) für das positive Literal x_j wie auch für das negative Literal $\bar{x}_j$ erfüllbar sein.
Die Formel φ liegt im Normalfall in konjunktiver Normalform (*CNF*) vor, so dass wir die Darstellung $Q_1 x_1 ..... Q_n x_n \ C_1 .... C_m$ erhalten.

Die Reihenfolge der Quantifizierungen spielt eine wichtige Rolle und muss strikt eingehalten werden. Da $\exists x \exists y = \exists y \exists x$ und $\forall x \forall y = \forall y \forall x$ gelten, können gleichquantifizierte Variablen mit derselben Priorität in einer Menge X_i zusammengefasst werden können. Da jede Variable nur genau einmal quantifiziert wird, sind diese Mengen disjunkt zueinander. Die Darstellung hat nun folgende Form: $Q_1 X_1 ..... Q_k X_k \; C_1 .... C_m$, k ≤ n . Das j aus jedem Q_j bzw. X_j gibt an, welches Quantifizierungslevel die in X_j enthaltenen Variablen aufweisen. Die Variabeln aus X_1 haben das Quantifizierunglevel 1, usw. Je niedriger das Quantifizierungslevel, desto höher ist die Priorität der Variablen.

Erwähnenswert ist an dieser Stelle, dass *SAT*-Instanzen implizit alle enthaltenen Variablen existenzquantifizieren, so dass *SAT* ein Spezialfall von *QBF* ist.

Für die später folgenden Regeln der Implizierbarkeit und Erfüllbarkeit ist es nötig die *CNF* für QBFs um so genannte *Cubes* zu erweitern. Ein solcher *Cube* S_i ist eine Konjunktion von Literalen und hat daher die Form $S_i = x_{j_1} x_{j_2} .... x_{j_n}$. Diese *Cubes* werden disjunktiv miteinander verknüpft und an die „normale“ *CNF* für *QBF*s angefügt. Die *Cubes* werden so gewählt, dass sie redundant zu den Lösungen der *CNF* für *QBF*s sind und die quantifizierte Boolesche Formel daher nicht in ihrer Funktionalität beeinflusst wird. Jeder *Cube* repräsentiert mindestens eine Lösung der *CNF*. Es ist nicht gefordert, dass die Disjunktion der *Cubes* alle Lösungen repräsentiert, sondern nur einen Teil dessen. Somit erhalten wir eine Darstellung der Form $Q_1 X_1 ..... Q_k X_k \; C_1 .... C_m + (S_1 + S_2 + ... + S_k)$, welche im folgenden als *ACNF* (Augmented CNF) bezeichnet wird.

Eine Klausel heißt *Tautology Clause*, wenn beide Literale einer Variablen in ihr vorkommen. Ebenso heißt ein *Cube*, der beide Phasen einer Variablen enthält, ein *Empty-Cube*. Im Folgenden bezieht sich der Begriff Klausel auf *Non-Tautology Clauses* und der Begriff *Cube* auf *Non-Empty Cubes*, da später definierte Regeln nicht für *Tautology Clauses* bzw. *Empty-Cubes* gelten.

Des Weiteren werden im folgenden *E(C)* und *E(S)* verwendet um die Menge der existenziell quantifizierten Literale in der Klausel *C* bzw. dem *Cube S* zu beschreiben. Ebenso werden *U(C)* und *U(S)* verwendet um die Menge der universellen Variablen in einer Klausel *C* bzw. einem *Cube S* darzustellen. Eine im Folgenden geltende Konvention ist, dass existenziell quantifizierte Variablen und ihre Literale durch Buchstaben repräsentiert werden, die am Anfang des Alphabets stehen, also *a*, *b*, *c* usw. Ebenso werden allquantifizierte Variablen und ihre Literale durch Buchstaben dargestellt, die sich am Ende des Alphabets befinden, also *x*, *y*, *z* etc.

Der DLL-Algorithmus für QBF

Ein entscheidendes Kriterium für das Lösen von *QBF*s ist die Einhaltung der Reihenfolge der Quantifizierungen. Es wird von links beginnend eine der Variablen mit dem kleinsten Quantifizierungslevel, also aus der Menge X_1, zuerst entschieden. Im nächsten Durchgang wird die nächste Variable aus X_1 entschieden, bis in dieser Menge alle Variablen entschieden worden sind und dann eine Variable aus der Menge mit dem nächsthöheren Quantifizierungslevel gewählt wird, usw. Eine neue Variable kann also nur dann entschieden werden, wenn alle Variablen mit kleinerem Quantifizierungslevel bereits einen Wert zugewiesen bekommen haben.
Zusätzlich zur Einhaltung der Quantifizierungslevels müssen die Regeln für Konflikte und Implikationen angepasst werden, da die Regeln für den *DLL*-Algorithmus für *SAT* allquantifizierte Variablen außer Betracht lassen.
Wie oben erwähnt, ist es bei *QBF* Formeln zwingend erforderlich das allquantifizierte Variablen in beiden Zweigen erfüllbar sind. Wird nun der *DLL*-Algorithmus für *QBF* angepasst, so muss eine Routine hinzugefügt werden, die bei erfüllenden Belegungen alle bisher entschiedenen allquantifizierten Variablen *flippt* und deren anderen Zweig ebenfalls auf Erfüllbarkeit prüft, diese Aufgabe übernimmt die Funktion *analyze_SAT*(). In Abbildung 2 ist ein schematischer Pseusocode zum *DLL*-Algorithmus für *QBF* zu sehen.

```
while(1) {
  decide_next_branch();                          //Variablenentscheidung
  while (true) {
    status = deduce();                           //Implikationen
    if (status == CONFLICT) {
      blevel = analyze_conflict();               //Konfliktdiagnose und
      if (blevel == 0)                           //Konfliktbehandlung
        return UNSAT;
      else backtrack(blevel);
    }
    else if (status == SATISFIABLE) {            //Behandlung für erfüllende
      blevel = analyze_SAT()                     //Belegungen
      if (blevel == 0)
        return SAT;
      else backtrack(blevel);
    }
    else break;                                  //nächste Variablenentscheidung
  }
}                                                //Terminierung
```

Abbildung 2: Der Pseudocode des DLL-Algorithmus für QBF

Die zugehörigen Regeln der *deduce()*-Funktion für Konflikte, Implikationen und Erfüllung werden im Folgenden für *QBF*s erweitert.

Konfliktregel für Non-Tautology-QBF-Klauseln

Damit ein Konflikt in einer Klausel *C* entsteht müssen ebenso wie bei SAT alle existenziellen Literale den Wert 0 aufweisen.

$\forall a \in E(C),\ V(a) = 0$

Darüber hinaus ist es für einen Konflikt erforderlich, dass im aktuellen Zweig keinem der universellen Literale der Wert 1 zugewiesen worden, d.h. alle bis dahin aufgetauchten universellen Literale müssen entweder den Wert 0 aufweisen oder in nur einem Zweig den Wert 1 haben und im anderen Zweig einen noch unbestimmten Wert.

$\forall x \in U(C),\ V(x) \neq 1$

Treffen beide Bedingungen zu, so ist ein Konflikt vorhanden; der aktuelle Zweig ist somit unerfüllbar.

Implikationsregel für Non-Tautology-QBF-Klauseln

Wenn eine Klausel C ein existenzielles Literal a hat, welches als einziges in der Klausel *C* keinen zugewiesenen Wert hat und darüber hinaus alle anderen enthaltenen Literale den Wert 0 aufweisen, also

$a \in E(C),\ V(a) = X.\ Für\ \text{alle}\ b \in E(C),\ b \neq a;\ \text{V(b)} = 0$

gilt und außerdem keines der in *C* enthaltenen universellen Literale den Wert 1 aufweist, also

$\forall x \in U(C),\ V(x) \neq 1. Wenn V(x) = X, dann\ L(x) > L(a)$

gilt (d.h. alle unzugewiesenen universellen Literale müssen ein höheres Quantifizierungslevel als das von *a* aufweisen und alle mit kleinerem Quantifizierungslevel den Wert 0), dann muss *a* den Wert 1 zugewiesen bekommen, da sonst die Klausel im aktuellen Zweig unerfüllt wäre und somit auch die Formel *f* unerfüllt wäre. Gilt diese Regel, so wird *C* als *Unit Clause* und *a* als *Unit Literal* bezeichnet. Wichtig ist an dieser Stelle das diese Implikationsregel ausschließlich mit existenziellen Literalen *a* funktioniert.

Erfüllungsregel für Non-Empty-QBF-Cubes

Ein *Cube S* in einer *QBF* ist erfüllt, wenn alle enthaltenen universellen Literale den Wert 1 aufweisen, also

$\forall x \in U(S),\ V(x) = 1$

gilt und außerdem in aktuellen Zweig keinem existenziellen Literal der Wert 0 zugewiesen worden ist, also

$\forall a \in E(S),\ V(a) \neq 0$

gilt.

Implikationsregel für Non-Empty-QBF-Cubes

Gilt für einen *Cube S* mit einem universellen Literal *x* das erstens alle universellen Literale aus *S* außer *x* den Wert 1 aufweisen und x noch unzugewiesen ist, also

$x \in U(S),\ V(x) = X.\ Für$ alle $y \in U(S),\ y \neq x;\ \mathrm{V(y)} = 1$

gilt und außerdem kein bisher entschiedenes existenzielles Literal *a* aus *S* den Wert 0 hat (wenn *a* unzugewiesen ist, so muss es ein höheres Quantifizierungslevel haben als *x*), also

$\forall a \in E(S),\ V(a) \neq 0.\ Wenn V(a) = X,\ dann\ L(a) > L(x)$

gilt, dann muss *x* den Wert 0 zugewiesen bekommen, damit unnötiger Suchraum abgeschnitten wird und die Suche fortgesetzt werden kann. Gilt diese Regel für einen *Cube S*, so wird *S* als *Unit Cube* und *x* als *Unit Literal* bezeichnet.
Diese Implikationsregel funktioniert nur für universelle Literale.

Conflict Driven Learning

Viele der im Suchprozess durchgeführten Entscheidungen enden in einem Konflikt, so dass der aktuelle Zweig unerfüllbar ist und die Suche in einem anderen Teil des Suchbaums fortgesetzt werden muss. Das Konzept des *Learnings*, erstmal für *SAT* vorgestellt in [5], ist eine sinnvolle Methode aus solchen Konflikten zu lernen und so zu vermeiden, dass unerfüllbare Zweige und alle zugehörigen Permutationen im Suchbaum (nochmals) durchlaufen werden. Konfliktbasiertes *Learning* ist als eine Verbesserung der *analyze_conflict()*-Funktion aus dem *DLL*-Algorithmus anzusehen. Die am Konflikt beteiligten Klauseln werden zu diesem Zweck durch die Resolutionsregel zu einer neuen

Klausel zusammengefügt (ohne die alten Klauseln zu löschen) und die somit neu erhaltene, gelernte Klausel wird in die Klauselmenge hinzugefügt.

Resolution

Gegeben seien zwei Klauseln C_1 und C_2. Die Klausel C_1 enthält die Literale $\{l_1, ..., l_m, a\}$ und C_2 die Literale $\{l_{m+1}, l_{m+2}, ..., l_n, a'\}$. Die Literale a und a' sind komplementäre Literale derselben Variable. Durch Resolution dieser beiden Klauseln wird eine neue Klausel mit den Literalen $\{l_1, ..., l_m, l_{m+1}, ..., l_n\}$.

Die Methode *resolution_gen_clause()* wird in der *analyze_conflict()*-Methode aufgerufen bekommt als Parameter die Klausel übergeben, in der der Konflikt aufkam. In *resolution_gen_clause()* sucht nun *choose_literal()* das zuletzt implizierte existenzielle Literal aus der konfliktverursachenden Klausel heraus. Nun wird durch *variable_of_literal()* die zugehörige Variable des Literals ermittelt. Die Methode *antecedent()* gibt im nächsten Schritt die Klausel *ante* zurück, in der das Literal impliziert wurde. Die Resolution wird dann auf der konfliktverursachenden Klausel und der antecedent-Klausel mit der durch *variable_of_literal()* erhaltenen Variable durchgeführt. Die Resolution wird solange mit der erhaltenen, neuen Klausel iterativ durchgeführt bis folgende Bedingungen für die generierte Klausel gelten:

1. Von allen enthaltenen existenziellen Variablen hat genau eine Variable *v* das höchste Entscheidungslevel.
2. Alle universellen Literale mit kleinerem Quantifizierungslevel als das der Variablen v wurden mit dem Wert 0 belegt

Durch Einhaltung dieser Bedingungen ist gewährleistet, dass die oben aufgeführten Regeln auch auf die gelernte, neue Klausel anwendbar sind. Außerdem kann aus dieser entstandenen Klausel unmittelbar impliziert werden, da die obigen Regeln gewährleisten, dass eben genau die Implikationsregel für *Non-Tautology*-QBF-Klauseln hier greifen kann. Die Abbildung 3 stellt die *analyze_conflict()*-Prozedur dar, welche dann die *resolution_gen_clause()*-Funktion mit ihren weiter oben erläuterten Einzelschritten aufruft.

```
resolution_gen_clause( cl ) {
  lit = choose_literal (cl);                    //Finden der
  var = variable_of_literal( lit );             //konfliktverursachenden
  ante = antecedent( var );                     //Klausel
  new_cl = resolve(cl, ante, var);
  if (stop_criterion_met(new_cl))               //Durchführung der
    return new_cl;                              //Resolution
  else
    return resolution_gen_clause(new_cl);
}
analyze_conflict(){                             //Konfliktdiagnose
  conf_cl = find_conflicting_clause();          //Resolution aufrufen
  new_cl = resolution_gen_clause(conf_cl);      //Klausel zur
  add_clause_to_database(new_cl);               //Datenbank hinzufügen
  back_dl = clause_asserting_level(new_cl);
  return back_dl;
}
```

Abbildung 3: Generieren und hinzufügen neuer Klauseln durch *Resolution*

Satisfiability Directed Learning

Wie weiter oben bereits erwähnt wurde, ist im Falle einer erfüllenden Belegung für *QBF*s die Suche nicht beendet, wie es bei *SAT* der Fall ist. Zusätzlich zur Erfüllung der *CNF* ist es bei *QBF*s notwendig, dass beide Zweige der universellen Variablen erfüllt werden um letztendlich die gesamte *QBF* zu erfüllen. Wird eine erfüllende Belegung gefunden, so muss ein *Backtrack* vom aktuellen Zweig zur zuletzt entschiedenen universellen Variable durchgeführt werden und der andere Zweig dieser Variable ebenfalls auf Erfüllbarkeit geprüft werden. Ist auch dieser Zweig erfüllt, so wird die selbe Prozedur auf die universelle Variable mit dem nächstkleineren Entscheidungslevel angewendet bis entweder ein Konflikt auftritt oder alle universellen Variablen die bisher entschieden worden sind in beiden Zweigen erfüllt werden können.

Bisher gab es keinerlei Möglichkeiten erfüllende Belegungen, die jedoch nicht alle universellen Variablen in beiden Zweigen erfüllen in einer Datenstruktur festzuhalten und so, ähnlich wie beim konfliktbasierten Lernen, auch von erfüllenden Belegungen zu lernen. Im *QBF*-Solver *Quaffle* wird erstmals eine solche Idee implementiert. Das verwendete Verfahren hierfür ist stark an das obige konfliktbasierte Lernen angeknüpft und verhält sich auch diesbezüglich symmetrisch. Die Idee ist, sich erfüllende Belegungen in Form von redundanten *Cubes* zu merken, ähnlich wie das Lernen von Klauseln durch Konflikte. Der Pseudocode der Prozedur zur Generierung der *Cubes* ist in Abbildung 4 aufzufinden. Die Konsensus-Regel ist das exakte Pendant zur weiter oben vorgestellten Resolution, nur dass er nicht auf disjunktiv verknüpften Termen (Klauseln), sondern auf konjunktiv verknüpften Termen (*Cubes*) arbeitet.

Beispiel-*ACNF*:

```
(a + b + x)(c + y')(a + b' + y')(a + x' + y') + xy'
```

$\{a, y'\}$ ist eine erfüllende Belegung und wird als *Cube* ay' an die *ACNF* angefügt:

```
(a + b + x)(c + y')(a + b' + y')(a + x' + y')+ ay'+ xy'
```

Allerdings wird hier das Anwenden der Regel solange wiederholt, bis der generierte *Cube* folgende Bedingungen erfüllt:

1. Von allen enthaltenen universellen Variablen hat genau eine Variable *v* das höchste Entscheidungslevel.
2. Alle existenziellen Literale mit kleinerem Quantifizierungslevel als das der Variablen v wurden mit dem Wert 1 belegt.

Durch diese Bedingungen ist gewährleistet, dass erzeugte Cube den Suchraum so beschneidet, dass zugehörige erfüllende Belegungen nicht im weiteren Verlauf des Lösungsprozesses durchsucht werden und die Implikationsregel für *Non-Empty-QBF-Cubes* unmittelbar auf diesen *Cube* anwendbar ist. Die Abbildung 4 stellt die *analyze_SAT()*-Prozedur dar, welche dann die *consensus_gen_clause()*-Funktion aufruft, welche ebenso wie die *resolution_gen_clause()*-Routine funktioniert, mit dem Unterschied, dass sie auf Würfeln arbeitet und die Terminierungskriterien anders sind als in einem Konfliktfall.

```
consensus gen cube( s ) {                    //Durchführen der
  lit = choose_literal (s);                  //Konsensus-Regel auf
  var = variable_of_literal( lit );          //zwei cubes zum
  ante = antecedent( var );                  //Generieren neuer
  new_cube = resolve(s, ante, var);          //cubes
  if (stop_criterion_met(s))
    return new_cube;
  else
    return consensus_gen_cube(new_cube);
}

analyze_SAT(){                               //Erfüllungsdiagnose.
  cube = find_sat_cube();                    //Wenn kein solcher
  if (cube == NULL)                          //cube vorhanden,
    cube = construct_sat_induced_cube();     //neuen cube
  if (!stop_criterion_met(cube))             //generieren.
    cube = consensus_gen_cube(cube);         //Hinzufügen in die
  add_cube_to_database(cube);                //ACNF
  back_dl = cube_asserting_level(cube);
  return back_dl;
}
```

Abbildung 4: *Cube*-Generierung anhand der *Konsensus*-Regel

Experimentelle Benchmarks

In Abbildung 5 ist eine Tabelle zu sehen, die *Quaffle-CDL*, d.h. die *Quaffle*-Version mit konfliktbasiertem Lernen aber ohne erfüllbarkeitsbasiertes Lernen, mit *Quaffle*-Versionen ohne *Learning* und den *QBF*-Solvern *QuBE* und *Decide* vergleicht. *Quaffle-BJ* (Backjump) beherrscht gegenüber der naiven Version *nichtchronologisches Backtracking*. *QuBE* ist ein

Benchmark Name		Quaffle -CDL	Quaffle -BJ	Quaffle -naïve	QuBE -BT	QuBE -BJ	Decide	Benchmark Name		Quaffle -CDL	Quaffle -BJ	Quaffle -naïve	QuBE -BT	QuBE -BJ	Decide
BLOCKS3i.4.4	F	0.07	0.07	-	-	-	0.05	impl20	T	15.82	-	-	-	-	-
BLOCKS3i.5.3	F	29.92	128.42	-	-	-	31.89	logn...A0	F	0	0	0	0	0	0.01
BLOCKS3i.5.4	T	2.91	30.75	-	-	-	6.59	logn...A1	F	2.23	67.47	-	-	7.28	0.47
BLOCKS3ii.4.3	F	0.05	0.07	-	-	5.02	0.03	logn...A2	T	127.4	-	-	-	-	28.28
BLOCKS3ii.5.2	F	0.13	0.82	-	-	82.19	0.07	logn...B0	F	0.01	0.01	0.01	0	0	0.03
BLOCKS3ii.5.3	T	0.33	2	-	-	-	1.5	logn...B1	F	8.47	342.55	-	-	37.89	0.61
BLOCKS3iii.4	F	0.03	0.05	-	-	1.62	0.01	logn...B2	F	767.94	-	-	-	-	1.88
BLOCKS3iii.5	T	0.28	0.72	-	-	-	0.26	R...3...50_0.T	T	1.23	3.15	-	0	0	0.03
BLOCKS4i.6.4	F	254.11	-	-	-	-	5.35	R...3...50_1.F	F	0.02	2.52	-	0.03	0.01	0.47
BLOCKS4ii.6.3	F	400.99	-	-	-	-	4.53	R...3...50_2.T	T	0.83	1.11	738.7	0	0	0.05
BLOCKS4ii.7.2	F	-	-	-	-	-	9.15	R...3...50_3.T	T	1.07	1.65	521.98	0.11	0	0.06
BLOCKS4ii.7.3	T	-	-	-	-	-	731.24	R...3...50_4.T	T	1.44	2.74	-	0.06	0.01	0.1
BLOCKS4iii.6	F	40.27	-	-	-	-	2.92	R...3...50_5.T	T	0.97	21.42	71.41	0.01	0	0.07
BLOCKS4iii.7	T	-	-	-	-	-	414.76	R...3...50_6.F	F	1.53	11.99	-	0.07	0.03	20.52
CHAIN12v.13	T	0.31	0.32	-	0.36	0.34	0.41	R...3...50_7.F	F	0.6	7.88	-	0.05	0.02	1.91
CHAIN13v.14	T	0.69	0.69	-	0.84	0.8	0.79	R...3...50_8.F	F	0.28	1.33	-	0.22	0.06	0.32
CHAIN14v.15	T	1.47	1.49	-	2.45	2.27	1.6	R...3...50_9.T	T	0.87	1.03	40.99	0.05	0.01	0.25
CHAIN15v.16	T	3.19	3.25	-	4.57	5.17	3.25	R...7...60_0.F	F	0.13	2.19	479.68	0.44	0.02	0.01
CHAIN16v.17	T	6.9	7.03	-	15.53	13.38	6.7	R...7...60_1.T	T	0.23	0.52	0.55	0.07	0.01	0.06
CHAIN17v.18	T	15.27	15.14	-	34.89	43.44	14.48	R...7...60_2.T	T	1.28	0.65	423.71	0	0	0.05
CHAIN18v.19	T	32.21	33.02	-	97.73	100.06	31.21	R...7...60_3.T	T	0.33	0.51	36.83	0.75	0.02	0.15
CHAIN19v.20	T	74.04	75.09	-	234.65	249.04	61.08	R...7...60_4.T	T	13.44	51.64	-	0.76	0.01	0.04
CHAIN20v.21	T	152.88	166.98	-	527.92	650.44	130.71	R...7...60_5.F	F	1.32	8.32	-	0.64	0.01	27.12
CHAIN21v.22	T	340.29	362.24	-	1271	1462.59	272.58	R...7...60_6.T	T	0.51	1.79	150.08	0.19	0.03	0.83
CHAIN22v.23	T	741	846.08	-	-	-	569.65	R...7...60_7.T	T	2.22	11.9	-	0.73	0.06	0.18
CHAIN23v.24	T	1783.73	1790.14	-	-	-	1200.91	R...7...60_8.F	F	0	0	1.85	0.02	0.01	0.08
impl02	T	0	0	0	0	0	0	R...7...60_9.T	T	0.23	3.75	-	0.02	0	0.09
impl04	T	0	0	0	0	0	0.01	TOILET02.1.iv.3	F	0	0	0	0	0	0
impl06	T	0.01	0.01	0.01	0	0.01	0.04	TOILET02.1.iv.4	T	0	0	0	0	0	0
impl08	T	0.01	0.04	0.07	0.01	0	0.29	TOILET06.1.iv.11	F	40.15	7.46	757.08	25.68	5.4	10.45
impl10	T	0.02	0.33	0.48	**	**	2.29	TOILET06.1.iv.12	T	18.17	5.57	1633.03	12.36	1.48	0.1
impl12	T	0.06	2.34	3.57	**	**	17.38	TOILET07.1.iv.13	F	-	104.47	-	599.6	92.06	141.17
impl14	T	0.25	18.19	26.41	**	**	130.64	TOILET07.1.iv.14	T	-	74.77	-	265.3	23	0.22
impl16	T	0.99	135.22	195.52	**	**	974.53	TOILET10.1.iv.20	T	-		-	-	-	1.31
impl18	T	3.98	985.57	1445.76	**	**	-	TOILET16.1.iv.32	T	-		-	-	-	15.39

Abbildung 5: Verschiedene *Quaffle*-Versionen im Vergleich mit *QuBE* und *Decide*

QBF-Solver ohne *Learning*, *Decide* hingegen beherrscht konfliktbasiertes Lernen. Aus der Tabelle wird ersichtlich das *Quaffle* ohne *Learning* meist deutlich langsamer zum Ziel kam als mit *Learning* und erst mit konfliktbasiertem Lernen gegen den Solver *Decide* bestehen kann. Die in der zweiten Spalte stehenden Buchstaben geben an, ob die *QBF*-Instanz erfüllbar ist oder nicht (**T**rue, **F**alse). Das Timeout beträgt exakt 30 Minuten bzw. 1800 Sekunden. Wurde das Timeout überschritten, so ist dies in den jeweiligen Zellen durch ein „-“ gekennzeichnet. Die Benchmarks wurden auf einer Pentium III-Maschine mit 933 MHz Taktfrequenz und 1 GByte Hauptspeicher durchgeführt.
Abbildung 6 zeigt einen Vergleich von *Quaffle-Full*, also der Version mit konfliktbasiertem Lernen und erfüllbarkeitsbasiertem Lernen, und *Quaffle-CDL*. Jeweils in der linken Spalte der Solver ist angegeben, wieviele erfüllende Belegungen für die *CNF* der *QBF*-Instanz durchlaufen wurden bis das endgültige Ergebnis feststand. Hier ist zu erkennen, dass erfüllungsbasiertes Lernen bei vielen Instanzen keine Vorteile bezüglich Laufzeit des Solvers mit sich bringt und der Overhead für Erfüllungsdiagnose und Generierung der neuen *Cubes* die Laufzeit deutlich verlängert (TOILET, CHAIN). Andererseits existieren auch Instanzen,

Testcase	T/F	Quaffle-CDL			Quaffle-Full		
		No. Sat. Leaves	No. Conf. Leaves	Runtime	No. Sat. Leaves	No. Conf. Leaves	Runtime
TOILET06.1.iv.12	F	24119	7212	18.23	17757	8414	74.16
TOILET06.1.iv.11	T	30553	11000	39.51	30419	13918	221.45
CHAIN15v.16	T	32768	43	3.15	32768	43	142.21
CHAIN16v.17	T	65536	46	6.82	65536	46	472.38
CHAIN17v.18	T	131072	49	14.85	131072	49	1794.35
impl16	T	160187	17	0.97	106	17	0.02
impl18	T	640783	19	3.88	124	19	0.02
impl20	T	2563171	21	15.51	142	21	0.02
R3...3...50_8.F	F	11845	374	0.29	59	460	0.05
R3...3...50_9.T	T	33224	87	0.87	35	50	0.02
logn...A2	T	3119	11559	125.85	1937	14428	193.88
logn...B1	F	2	601	8.26	2	609	8.18
BLOCKS4ii.6.3	F	5723	52757	367.54	98	45788	591.95

Abbildung 6: Gegenüberstellung von Quaffle ohne und mit SAT-basiertem Lernen

bei denen der Vorteil durch SAT-basiertes Lernen sehr groß ausfällt. Es stellt sich also die Frage, wann die Generierung der zusätzlichen *Cubes* einen Vorteil verschafft. Die Autoren von *Quaffle* schreiben in [2], dass ihre neue Technik dann besonders gut greift, wenn die Instanz sehr viele allquantifizierte Variablen enthält und auch viele erfüllende Belegungen hat. Andernfalls empfiehle es sich *SAT*-basiertes Lernen zu deaktivieren.

Zusammenfassung

In dieser Ausarbeitung wurden die Unterschiede von *SAT* und *QBF* aufgezeigt. Außerdem wurde die Grundidee des für *QBF*s angepassten *DLL*-Algorithmus mit den erweiterten Regeln für Implikationen, Konflikte und Erfüllbarkeit dargelegt und diskutiert. Schließlich wurden die Konzepte des *QBF*-Solvers *Quaffle* erläutert, der seinen Schwerpunkt eindeutig auf das übliche, aber für *QBF*s erweiterte Lernen durch Konflikte setzt und als neue Technik auch Lernen durch erfüllende Belegungen zulässt. Um dies zu ermöglichen, wurde die einfache *CNF* für quantifizierte Boolesche Formeln um *Cubes* erweitert, wodurch eine symmetrische Form des Lernens von Konflikten und erfüllenden Belegungen erzielt wurde.

Literaturverzeichnis

Primärliteratur:

[1] Zhang, L.; Malik, S. (2002): Conflict Driven Learning in a Quantified Boolean Satisfiability. In: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design (ICCAD 2002)
Link: http://www.princeton.edu/~chaff/publication/iccad_2002.pdf

[2] Zhang, L.; Malik, S. (2002): Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In: Proceedings of 8th International Conference on Principles and Practice of Constraint Programming (CP 2002).
Link: http://www.princeton.edu/~chaff/publication/cp_2002.pdf

Sekundärliteratur:

[3] Davis, M.; Logemann, G.; Loveland, D. (1962): A Machine program for theorem proving. In: Communications of the ACM, vol. 5, S.394-397. Link:
http://portal.acm.org/ft_gateway.cfm?id=368557&type=pdf&coll=GUIDE&dl=GUIDE&CFID=40004896&CFTOKEN=67399083

[4] Goldberg, E.; Novikov, Y. (2002): BerkMin: a Fast and Robust SAT-Solver. In: Design Automation and Test in Europe (DATE). S.142-149.
Link: http://www.ece.cmu.edu/~mvelev/goldberg_novikov_date02.pdf

[5] Marques-Silva, J. P.; Sakallah, K. A. (1999): GRASP: A Search Algorithm for Propositional Satisfiability. In: IEEE Transactions on Computers, vol. 48, S.506-521. Link: http://sdg.lcs.mit.edu/~ilya_shl/area/grasp.pdf

[6] Moskewicz, M.W.; Madigan, C.; Zhao, Y.; Zhang, L.; Malik, S. (2001): Chaff: Engineering an Efficient SAT Solver. In: Proceeding of the 38th Design Automation Conference. Link: http://research.microsoft.com/users/lintaoz/papers/dac_2001.pdf

[7] Cook, S.A. (1971): The complexity of theorem proving procedures. In: 3rd Symposium on Theory of Computing (ACM), S. 151-158. Link:
http://portal.acm.org/ft_gateway.cfm?id=805047&type=pdf&coll=GUIDE&dl=GUIDE&CFID=54705237&CFTOKEN=88544221

[8] Yu, Y.; Malik, S. (2005): Validating the Result of a Quantified Boolean Formula (QBF) Solver: Theory and Practice. Link: http://www.princeton.edu/~yyu/mypapers/verify_qbf.pdf

Web-Links:

[9] Yu, Y. (2005): yQuaffle - Solver Version 093004.
Link: http://www.princeton.edu/~chaff/Quaffle/yquaffle.093004.tar.gz

[10] Yu, Y. (2005): yQuaffle - Source Code.
Link: http://www.princeton.edu/~chaff/Quaffle/quaffle.tar.gz