30 Jahre Verifikationswerkzeuge - Von Boyer-Moore bis KeY

Niklas Henrich

Universität Koblenz-Landau

Zusammenfassung In dieser Seminarausarbeitung sollen di drei Verifikationswerkzeuge *ACL2*, *Isabelle* und *KeY* beschrieben werden. Es wird jeweils ein kurzer Überblick über die geschichtliche Entwicklung des jeweiligen Tools, sowie über die der verwendeten Logik gegeben. Weiter werden die Möglichkeiten zur Softwareverifikation dieser Tools beleuchtet und jeweils ein Beispiel einer praktische Anwendung jedes dieser Tools gegeben.

1 Einleitung

Verifikationswerkzeuge dienen dazu, den Benutzer bei der Verifikation von Software zu unterstützen. Sie ermöglichen es, durch eine Automatisierung sowie Überprüfung der jeweiligen Beweisschritte, Fehler in einem Beweis zu vermeiden. Es handelt sich also um ein Programm, welches interaktiv und teilweise automatisch für vorgegebene Sätze einen Beweis in einem gegeben Kalkül findet. Je mehr Beweise automatisch geführt werden können, umso größer ist natürlich die Zeitersparnis sowie der echte Mehrwert eines Verifikationswerkzeuges für den Nutzer.

Generell ist es möglich mit Verifikationswerkzeugen zu zeigen, dass ein Programm eine gegebene Spezifikation erfüllt. Es kann gezeigt werden, dass eine Methode eine gegebene Postcondition einhält, vorausgesetzt die Precondition gilt beim Eintritt in die Methode. Ferner kann gezeigt werden, dass gegebene Invarianten bei der Programmausführung nicht verletzt werden.

Verifikationswerkzeuge lassen sich generell in zwei verschiedene Klassen einteilen. Automatische Beweiser erledigen den Beweis ohne Zutun des Nutzers. Meistens ist es in diesem Fall aber nötig, eine hinreichende Zahl von Lemmata zu zeigen und diese dem Beweiser mitzuteilen. Diese können nun vom Beweiser in einer bestimmten Reihenfolge angewandt werden, um die zu zeigenden Eigenschaften genügend zu vereinfachen und letztendlich zu beweisen. Taktische / Interaktive Beweiser erwarten Anweisungen vom Nutzer. Dabei handelt es sich um die Auswahl von Regeln, welche auf das aktuelle Ziel anzuwenden sind. Somit steuert der Benutzer bei diesen Beweisern den Beweis durch die Reihenfolge der anzuwendenden Regeln, wobei hingegen bei automatischen Beweisern der Nutzer den Beweis durch das hinzufügen von Lemmata steuern kann.

1.1 Die ersten Theorembeweiser

Einer der ersten automatischen Theorembeweiser wurde Anfang der 60er Jahre von Newell, Shaw und Simon an der Carnegie Mellon Universität[7] entwickelt. Der so genannte *Logic Theorist* bewies viele Theoreme aus der *Principia Mathematica*. Das Programm verwendet die formale Aussagenlogik als Repräsentationsmedium. Eine verbesserte Version des *Logic Theorist* ergab 1963 den so genannten *General Problem Solver* (Newell und Simon). Im Laufe der Jahre entwickelten sich viele Theorembeweiser sowie die ersten, mit denen die Verifikation von Software ermöglicht wurde.

1.2 Voraussetzungen zur werkzeugestützten Softwareverifikation

Um Eigenschaften eines bestimmten Systems zeigen zu können, benötigt man zunächst ein Modell dieses Systems. Des weiteren ist eine präzise Spezifikation der zu zeigenden Eigenschaften des Systems zu erstellen. Das Modell sowie die Spezifikation des Systems sollte in einer Form vorliegen, die vom Computer verarbeitet werden kann. In den meisten Fällen haben die jeweiligen Verifikationswerkzeuge, aus historischen Gründen, ihre eigene Syntax für die Modellierungs- sowie Spezifikationssprache.

1.3 Sequenzenkalkül

Die meisten Theorembeweiser, insbesondere KeY und Isabelle, führen die Beweisschritte mit Hilfe des Sequenzenkaküls durch. Mit Hilfe dieses Kalküls läßt sich die Gültigkeit von Zielen auf die Gültigkeit von Teilzielen reduzieren. Dies kann den Vorteil haben, dass sich die Teilziele leichter zeigen lassen als die Gesamtaussage. Das Sequenzenkalkül hat sich als besonders geeignet für die interaktive Programmverifikation erwiesen.

1.4 Rückmeldungen von Theorembeweisern

Die Rückmeldungen, die der Nutzer von einem Theorembeweiser bekommt, können in drei Klassen aufgeteilt werden. Zum einen läßt sich die Formeln zeigen, der Nutzer erhält also ein klares "Ja". Die andere Möglichkeit ist, dass die Formel offensichtlich nicht zu zeigen ist, was in diesem Fall also ein klares "Nein" bedeutet. Weitaus schwieriger ist der Fall, wenn ein Beweis nicht abgeschlossen werden kann. In diesem Fall ist nicht ganz klar, warum dies der Fall ist. Zum einen kann es sein, dass man eine Taktik gewählt hat, die in diesem Fall zu keinem Ergebnis führt oder es fehlt ein Lemma. Es kann aber auch de Fall sein, dass eine nicht zu zeigende Aussage in einer der Formeln steckt. Handelt es sich um ein umfangreiches Programm, so kann es meist sehr schwierig sein, diese Formel zu identifizieren.

2 Boyer-Moore

2.1 Beschreibung

Die Arbeit zum Boyer-Moore Theorembeweiser [5] begann 1972 an der Universität von Schottland. Entwickler waren die beiden Namensgeber des Theorembeweisers, nämlich J Strother Moore[1] und Robert S. Boyer. Die erste Version dieses Theorembeweisers wurde 1973 unter dem Akronym *Thm* (THeoreM Prover) veröffentlicht. Das System unterlag einer konstanten Weiterentwicklung, und es erschienen mehrere Versionen bis zum Jahr 1986. Die in diesem Jahr erschienene Version erhielt den Namen *Nqthm* (New, Quantified THeorem Prover). Alle Versionen bis zu diesem Zeitpunkt verfügten über keine komfortable Benutzerschnittstelle. Alle Beweisziele plus die benötigen Lemmata mußten "als ganzes" an den Theorembeweiser weitergereicht werden, das bedeutet, dass man alle Informationen in eine Datei schrieb. Konnte ein Beweis nicht abgeschlossen werden, so mußte man die Datei um die benötigten Lemmata erweitern und wieder komplett an den Theorembeweiser weiterreichen. Dieser mußte nun von neuem mit dem Beweisen beginnen. Ein von Matt Kaufmann entwickeltes interaktives Interface beinhaltete die 1992 erschienene Version *PC-Nqthm*, mit Hilfe dessen die zuvor beschriebene Problematik gelöst wurde.

Der Beweiser selbst ist ein automatischer Beweiser. Der Benutzer muss das Programm aber mit der Eingabe passender Lemmata unterstützen, damit es in der Lage ist, die Sätze autotmatisch zu beweisen.

2.2 Anwendungen

Mit Hilfe dieses Systems konnte eine Reihe von Sätzen bewiesen werden, darunter auch Gödel's Unvollständigkeitssatz. Das System wurde unter anderem auch in der Software bzw. Hardwareverifikation eingesetzt, beispielsweise konnte die Korrektheit der *Berkeley C-String Library* kompiliert mit dem *gcc* für den Motorola MC86020 Microprozessor gezeigt werden. In diesem Fall wurde gezeigt, dass das Kompilat die gewünschten Eigenschaften erfüllt und nicht, dass der Quellcode die Spezifikation erfüllt. Der Grund für diese Vorgehensweise wird im Abschnitt über ACL2 näher beleuchtet.

3 ACL2

3.1 Beschreibung

ACL2[1] ist eine Weiterentwicklung des Boyer-Moore Theorembeweisers. Das Programm wurde Ende der 80er Jahre von Moore und Kaufmann entwickelt und steht dabei für *A Computational Logic for Applicative Common Lisp*. Die Motivation für eine Weiterentwicklung war die Tatsache, dass sich der Boyer-Moore

[1] Eine kleine Randbemerkung zum Namen dieses Mannes. Der Punkt hinter dem J fehlt nicht, es handelt sich dabei tatsächlich um seinen vollständigen ersten Vornamen.

Theorembeweiser als nicht Industrietauglich erwiesen hat. ACL2 sollte nun ein industrietauglicher Theorembeweiser sein, welcher auch mit größeren Aufgabenstellungen umgehen kann. Dies wurde unter anderem dadurch ermöglicht, dass die ACL2 Logik, wie später näher erläutert wird, in Common Lisp eingebettet wurde. Somit kann durch den Aufruf von nativen Common Lisp Befehlen, z.B. eine erhöhte Ausführungsgeschwindigkeit erreicht werden.

Bei ACL2 handelt es sich um mehrere Dinge in einem. Zum einen ist ACL2 eine ausführbare Logik. Die Logik selbst ist eine quantorenfreie Logik rekursiver Funktionen erster Stufe. Die Syntax ist stark an die Lisp Syntax angelehnt. Tatsächlich überschneidet sich die Syntax in einem so großen Maße, dass ein in ACL2 formuliertes Programm an einen Common Lisp Interpreter weitergegeben werden kann. ACL2 selbst beinhaltet einen Interpreter für ACL2 Programme und somit können Funktionen (Formeln) konkrete Werte annehmen, und das Ergebnis kann vom Interpreter evaluiert werden. Diese Tatsache macht sich der Theorembeweiser zunutze, indem er für Formlen ohne Variablen, aber mit konstanten Werten, diese in die Formeln einsetzt und nun die Formel durch das berechnete Ergebnis ersetzen kann.

Zum anderen handelt es sich bei ACL2 auch um eine Spezifikationssprache. Das bedeutet, dass sich Eigenschaften von Methoden und Algorithmen, sowie die eines kompletten Systems, spezifizieren lassen.

Schlussendlich handelt es sich bei ACL2 selbstverständlich auch um einen Theorembeweiser. Dieser basiert, wie auch schon in den vorherigen Abschnitten erwähnt, auf dem Boyer-Moore Theorembeweiser. Somit handelt es sich auch bei ACL2 um einen automatischen Theorembeweiser mit einem interaktiven Interface. Das Interface selbst ist ein Textinterface, es hat also keine grafische Schnittstelle. ACL2 kann auch in einen interaktiven Modus betrieben werden. In diesem kann ein Beweis Schritt für Schritt ausgeführt werden und es wird dem Benutzer die Möglichkeit gegeben, bestimmte Umformungen auf den aktuellen Beweisschritt anzuwenden. Somit ist es möglich, einen tieferen Einblick in den automatischen Beweis zu bekommen und somit leichter festzustellen, warum der Beweis nicht automatisch durchgeführt werden kann.

Für ACL2 existiert zu dem eine Semantik für Java, geauer gesagt für den Java-Bytecode, mit dessen Hilfe sich die Korrektheit von Java Programmen zeigen läßt. Der Grund, warum sich die Entwickler für die Verifikation auf Bytecode-Ebene entschieden haben, ist der gleiche Grund, warum auch die Korrektheit der, im Abschnitt über Boyer-Moore erwähnten, C-String Library auf Maschinencode-Ebene gezeigt wurde. Zum einen gehen sie davon aus, das Maschinencodeanweisungen, oder in diesem Fall Bytecodeanweisungen, simpler sind, als Anweisungen einer höheren Programmiersprache. Zum anderen haben sie den Eindruck, dass die informelle Spezifikation von Byte- oder Maschinencode, klarer und präziser ist, als die informelle Spezifikation von Anweisungen einer höheren Programmiersprache. Schlussendlich hat diese Vorgehensweise in ihren Augen den Vorteil, dass nicht die Korrekheit des Compilers gezeigt werden muss, da der Korrektheitsbeweis nicht auf Sourcecodeebene sondern auf Maschinencodeebene

geführt wird. Somit spielt die formale Korrekheit des Compilers zur Umsetzung des Quelltextes in Maschinencode, kein Rolle mehr.

3.2 Anwendungen

Es gibt eine Vielzahl von Anwendungsbeispielen für ACL2, dabei sei besonders auf [6] verwiesen. Eine dieser Anwendungen befaßt sich mit der Verifikation der Fließkomma-Division auf dem AMD K5. Der zu verifizierende Programmcode selbst umfaßt ca. eine Seite. Es existiert eine 10 Seiten lange informelle Spezifikation der AMD Designer, welche als Grundlage diente. Da zu diesem Zeitpunkt noch keine Beweise für Fließkomma-Arithmetik in ACL2 existierten, wurde ein Großteil der verwendeten Zeit für die Erstellung grundlegender Lemmata für Fließkomma-Arithmetik verwendet. Zu dem wurde die Semantik der verwendeten Microcode Befehle des AMD K5 formalisiert. Insgesamt wurden 130 Funktionen definiert und über 1200 Lemmata. Nur 47 der erstellten Funktionen sind speziell auf den zu beweisenden Algorithmus zugeschnitten, bei allen anderen handelt es sich, wie schon bereits erwähnt, um allgemeine Funktionen die Fließkomma-Arithmetik betreffend. Alle erstellten Funktionen und Lemmata konnten automatisch bewiesen werden. Bemerkenswert ist die Tatsache, dass die drei beteiligten Personen, Moore, Kaufmann und Lynch (AMD), nur 10 Wochen für dieses Projekt benötigt haben.

4 Isabelle

4.1 Beschreibung

Der interaktive / taktische Theorembeweiser Isabelle [4] wurde von Lawrence C. Paulson (University of Cambridge) und Tobias Nipkow (Technische Universität München) entwickelt. Er ist kostenlos verfügbar. Der große Unterschied zu anderen Theorembeweisern liegt in der Tatsache, dass es sich bei Isabelle um einen generischen Theorembeweiser handelt. Dies bedeutet, dass er nicht auf eine Logik beschränkt ist, sondern mit vielen verschiedenen Logiken umgehen kann. Die Logiken sind in Isabelle in einer Meta-Logik, Isabelle-Pure, definiert. Dabei handelt es sich um eine intuitionistische Higher Order Logic. Dies ist eine getypte Logik mit Konstanten, Variablen und Lambda-Ausdrücken. Zudem existieren All-Quantoren und die Implikation. Mit dieser Meta-Logik lassen sich nun eigene Logiken einführen. Higher-Order Logic, Prädikatenlogik und Zermalo-Frenkel Mengentheorie sind nur einige der Logiken, die im Lieferumfang von Isabelle enthalten sind, es existiert für diese Logiken also schon eine Definition in Isabelle-Pure.

Als komfortables Interface für Isabelle hat sich das (X)Emacs Programm Proof General erwiesen. Bei diesem Programm handelt es sich um eine generisches Interface zur Steuerung von Beweisassistenten. Der Vorteil dieses Interfaces liegt in der Tatsache, dass die Rückmeldungen und Eingaben an den Beweisassistenten vom Proof Generell grafisch aufbereitet werden. Zu dem enthält der Proof

General durch (X)Emacs eine Unterstützung für die Darstellung von logischen Symbolen (*X-Symbols*). Diese können nun als Symbol (z.B. $\forall$) anstelle einer Escape-Sequenz (z.B. \forall) dargestellt werden. Dies führt zur einer erheblich besseren Lesbarkeit der logischen Formeln. KeY hingegen stellt die Formeln nur als ASCII Sequenzen dar, da dort die Meinung vertreten wird, dass Softwaretechniker, im Gegensatz zu Mathematikern, mit dieser Notation besser umgehen können.

Mit Hilfe des *"Verification environment for sequential imperative programs in Isabelle/HOL"* [8] existiert eine Möglichkeit, in Isabelle Programme einer imperativen Programmiersprache zu verifizieren. Dabei handelt es sich um ein generisches Framework bestehend aus einer Programmiersprache und einem Verification Condition Generator.

Die Programmiersprache unterstützt die folgenden Spracheigenschaften:

- Lokale und globale Variabeln
- Rekursive Funktionsaufrufe
- Break, continue und return
- Exceptions
- Dynamische Methodenaufrufe

Seiteneffekte in Ausdrücken sind in der Sprache nicht erlaubt. Desweiteren ist die Sprache typsicher, das bedeutet, dass an Variablen eines bestimmten Typs nur Daten, die den gleichen Typ besitzen, zugewiesen werden können. Der Verification Condition Generator überprüft diese Einschränkiung. Die Behandlung einer Division durch 0 oder einer Dereferenzierung eines *null*-Pointers erfolgt während des Beweisens.

Mit Hile des Verification Condition Generators lassen sich nun aus einer, in Prädikatenlogik verfassten, Spezifikation sowie dem Programm die zu beweisenden prädikatenlogischen Formeln erstellen. Der VCG ist in der Lage, ein Programm als ganzes, oder schrittweise abzuarbeiten. Die so erstellten prädikatenlogischen Formeln können nun mit Hilfe von Isabelle bewiesen werden.

4.2 Anwendungen

Als Anwendungsbeispiel soll hier das Projekt *Verisoft* dienen. Es handelt sich dabei um ein auf 8 Jahre angelegtes Forschungsprojekt, welches durch das Bundesministerium für Bildung und Forschung (bmb+f) gefördert wird. Ziel ist die Erstellung von Tools und Methoden zur durchgängigen formalen Verifikation von Computersystemen. Dabei beschränkt sich das Projekt nicht nur auf die Verifikation der Software, sonder auch auf die der Hardware.

Das Projekt hat zahlreiche Industriepartner, darunter BMW, Infineon und T-Systems. Im akademischen Bereich wird versucht, ein komplett verifiziertes Computersystem zu erstellen. Dazu gehört neben einem verifizierten Prozessor auch ein verifiziertes Betriebssystem. Die Hauptanwendung stellt in verifizierter Email Client dar, welcher auf verifizierte SMTP und TCP/IP Protokoll-Implementationen zurückgreift.

Die meisten Anwendungen, die dort entwickelt werden, werden in der Programmiersprache C0 verfaßt. Es handelt sich dabei um eine Sprache, die auf dem oben erwähnten Verification Environment aufbaut. Die Sprache hat somit die gleichen Einschränkungen, insbesondere keine Pointerarithmetik. Die Syntax der Sprache ist, wie der Name suggeriert, an C angelehnt. Es existiert ein Tool, mit dessen Hilfe sich C0 Programme automatisch in die Programmiersprache des Verification Environments umsetzen lassen und somit erfolgen eine große Anzahl der Beweise innerhalb dieses Projektes mit Hilfe von Isabelle.

5 KeY

5.1 Beschreibung

Das KeY-Projekt begann 1998 an der Universität von Karlsruhe. Initiatoren waren die Professoren Menzel, Schmitt und Hähnle. Es handelt sich um ein Verbundprojekt der Universitäten Karlsruhe, Chalmers University of Technology und Koblenz sowie anderer Projektpartner. KeY [2] erweitert das kommerzielle CASE (Computer Aided Software Engineering) Tool *Borland Together Control Center* um Möglichkeiten zur Softwareverifikation. Ziel ist es dabei, die formale Spezifikation von Software in den Entwicklungsprozess zu integrieren.

Zielsprache von KeY ist Java. Genauer wird im Moment ein Dialekt von Java, nämlich JavaCard, abgedeckt. Dabei handelt es sich um eine Java Version, die speziell für die Anwendungen von SmartCard Applikationen zugeschnitten ist. Dies bedeutet, dass sie nicht den vollen Funktionsumfang von Java bietet (z.B. keine Multi-Threading Möglichkeit), dafür aber speziell auf die Benutzung von SmartCards zugeschnittene Erweiterungen besitzt.

Spezifikationen in KeY können in mehreren Varianten angegeben werden. Zum einen besteht die Möglichkeit JML (Java Modelling Language) zu verwenden. Dabei handelt es sich um eine Spezifikationssprache, welche direkt in den Java Quellcode, in so genannte JML Blöcke, geschrieben werden kann. Es handelt sich dabei nicht um eine allgemeine Spezifikationssprache für Software, sondern sie enthält unter anderem auf Java restriktierte Sprachmerkmale. Zum anderen besteht die Möglichkeit, Spezifikation in OCL (Object Constraint Language) anzugeben, welche Bestandteil von UML (Unified Modelling Language) ist. OCL Spezifikation werden auch direkt in den Quelltext eingefügt. Bei OCL handelt es sich aber, im Gegensatz zu JML, um eine allgemeine Spezifikationssprache, sie ist also nicht auf Java beschränkt.

KeY verwendet eine dynamische Logik, genauer gesagt JavaDL[3]. Dynamische Logik kann als Erweiterung der Hoare-Logik angesehen werden. Im Gegensatz zu Hoare-Logik können in dynamischer Logik die Vor- und Nachbedingungen auch Programme enthalten. Der verwendete JavaDL-Kalkül unterstützt alle Sprachkonstrukte von JavaCard.

KeY verwendet symbolische Ausführung um die Beweisziele in JavaDL zu generieren. Dabei wird das Programm schrittweise abgearbeitet, aber anstatt konkreter Werte, werden Variablen benutzt. Kann ein Programmablauf, z.B.

auf Grund einer Variablenbelegung, verschiedene Programmteile durchlaufen, so werden für jeden möglichen Programmdurchlauf, Beweisziele generiert. Der Beweis verzweigt somit in mehrere Teilziele, und zwar für jeden möglichen Programmdurchlauf, eines. Ist nun ein Beweisziel in mehrere Teilziele zerlegt, so müssen nun logischerweise die einzelnen Teilziele bewiesen werden, bevor das Beweisziel abgeschlossen ist.

KeY kann Beweise bis zu einem bestimmten Grad an Komplexität automatisch beweisen. Primär handelt es sich bei KeY aber um einen interaktiven / taktischen Beweiser. Der Beweiser selbst verfügt über eine komfortable grafische Oberfläche, über welche der Benutzer mögliche anzuwendende Regeln zur Auswahl angeboten bekommt. Zu dem ist diese Auswahl sortiert, dass bedeutet, dass die am wahrscheinlichsten passendste Regel als oberstes geführt wird. Die Liste der anzuwendenden Regeln ist dabei abhängig vom aktuell markierten Bereich. Es besteht die möglichkeit, eine Formel als ganzes zu markieren, oder nur einzele Teile dieser Formel. Dies steht im starken Gegensatz zu Isabelle. Dort kann der Benutzer nicht, Aufgrund seiner Auswahl, eine passende Liste an Regeln erhalten. Zu dem muss man in Isabelle die Stelle, an welcher eine Regel angewandt werden soll, eigenständig angeben, indem man an die gewünschte Stelle navigiert und die Regel dort per Hand einträgt.

5.2 Anwendungen

Als Anwendungsbeispiel soll hier ein Beispiel aus dem Umfeld der Deutschen Bahn dienen. Fahrpläne sowie Geschwindigkeitsbegrenzungen werden mit einer Software, "Satzerstellung betrieblicher Fahrplanunterlagen", für die Züge im Netz der Deutschen Bahn erstellt. Die Software ist in Smalltalk entwickelt worden und wurde, damit sie von KeY verifiziert werden kann, in Java reimplementiert. Die Orginal Software bestand aus ca. 700 Klassen. Um die 80 Klassen davon waren für die Berechnungen der Bremspunkte bzw. Höchstgeschwindigkeiten zuständig. Für die Software selbst war eine informelle Spezifikation in natürlicher Sprache vorhanden, aus welcher eine formelle OCL Spezifikation erstellt worden ist. Die Software wird nun verifiziert und somit kann nicht nur die Korrektheit der Software, sondern auch die der benutzten Algorithmen zur Errechnug der Geschwindigkeiten und Bremspunkte etc., gezeigt werden. Als Zwischenergebniss der noch laufenden Arbeitet läßt sich sagen, dass Mehrdeutigkeiten und Ungenauigkeiten in der informellen Spezifikation gefunden wurden und zudem einige ineffiziente Programmteile.

6 Wann beweise ich mein erstes Programm?

In diesem Kontext ist die Tatsache wichtig, dass Beweiser die Arbeit unterstützen und erleichtern können, sie erledigen sie nicht selbständig. In [6] wird, als Antwort auf die oben gestellte Frage, das Beweisen mit dem Programmieren verglichen. Es geht vergleichsweise schnell, die Syntax einer Sprache zu lernen. Es dauert aber dafür im Gegenzug eine sehr lange Zeit bis man Programmieren

kann. Bis man dann auch noch "gut" Programmieren kann, also erweiterbare, wieder verwertbare und gut strukturierte Software entwickelt, dauert es eine kleine Ewigkeit und Bedarf einer Menge an Praxis. Das gleiche läßt sich so auch auf die Anwendung von Theorembeweisern übertragen. Die Bedienung ist relativ schnell erlernt, ein Verständniss für die Struktur der geforderten Beweise aufzubauen, bedarf einer langen Zeit und einer Menge an Übung.

7 Zusammenfassung

Es wurden die drei Theorembeweiser ACL, Isabelle und KeY detaillierter vorgestellt. ACL2 ist, gerade wegen seiner Wurzeln im Boyer-Moore Theorembeweiser, einer der am längsten bestehenden Theorembeweiser. Er wird hauptsächlich erfolgreich in der Industrie angewandt und es wurden schon eine Vielzahl an größeren Problemen mit diesem Beweiser gelöst.

Isabelle hingegen ist *der* Beweiser im akademischen Umfeld. Dies ist sicherlich auf seine generische Erweiterbarkeit zurückzuführen. Es existieren eine Vielzahl von Logiken sowie Semantiken für viele Programmiersprachen für diesen Beweiser.

KeY hingegen versucht einen modernen Weg einzuschlagen, in dem es versucht, die Verifikation von Software in den Entwicklungsprozess zu integrieren. Außerdem hat es sich, mit der Zielsprache Java, eine auch in Zukunft relevante Programmiersprache ausgesucht und wird somit sicherlich auch in Zukunft an Bedeutung zunehmen.

Literatur

1. ACL2 homepage. http://www.cs.utexas.edu/users/moore/acl2/.
2. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The key tool. *Software and System Modeling*, 4:32–54, 2005.
3. Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.
4. Nipkow et al. A Proof Assistant for Higher-Order Logic.
5. Robert S. Boyer et al. The Boyer-Moore theorem prover and its interactive enhancement.
6. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Comuter-Aided Reasoning ACL2 Case Studies.* Kluwer Academic Publishers, 2000.
7. George F. Lugner. *Küntliche Intelligenz.* Pearson Akademischer Verlag.
8. Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL.