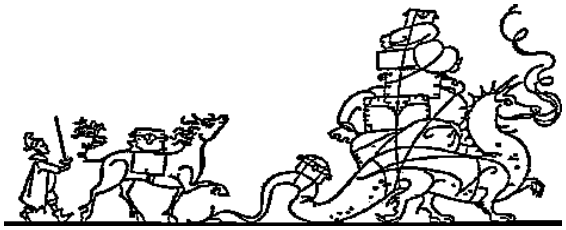
Programmverifikationssystem *Tatzelwurm* Weiterentwicklung während des KORSO-Projekts

P. Deussen, A. Hansmann,
Th. Käufl, S. Klingenbeck

Universität Karlsruhe
Institut für Logik, Komplexität
und Deduktionssysteme
Interner Bericht 7/95



1. Ergebnisse

Die Grundlagen der Verifikation sequentieller und imperativer Programme sind ausgereift [20]. Eine größere Anzahl von Werkzeuge steht für diese Aufgabe zur Verfügung. (Siehe etwa [23].) Doch zum Zeitpunkt des Beginns von KORSO war es meist nur möglich Programme, die in der Regel die Größenordnung von etwa 100 Code-Zeilen hatten, zu verifizieren. Der Hauptgrund dafür war der übergroße Aufwand, der für die Verifikation notwendig war. Deshalb haben wir die Steigerung des Automatisierungsgrades des Systems als einen Schwerpunkt unserer Arbeiten angesehen.

In diesem Bericht beschreiben wir die Erweiterungen das Verifikationssystem *Tatzelwurm* und die Erweiterungen, die wir im Rahmen des KORSO-Projekts gemacht haben. Die wesentlichen Eigenschaften des Systems sind:

1. Software wird mit traditionellen Methoden entwickelt. Die Anwendung von Transformationsregeln wird nicht unterstützt. Als Programmiersprache wird eine Teilmenge von UCSD-Pascal benutzt.
2. Als Spezifikationssprache verwenden wir die Prädikatenlogik erster Stufe mit Gleichheit und Funktionssymbolen.
3. Zum Nachweis, daß ein Programm seine Spezifikation erfüllt, benutzen wir die Hoare-Logik zur Erzeugung der Verifikationsbedingungen. Diese Bedingungen sind Formeln der Prädikatenlogik erster Stufe.
   Zum Beweis der Terminierung wird die Wohlordnungsmethode verwendet.
4. Ein mechanischer Beweiser für die Prädikatenlogik erster Stufe mit Gleichheit und Funktionssymbolen ist vorhanden. Er ist um Entscheidungsprozeduren für Theorien erweitert, erlaubt Interaktion mit dem Benutzer und die Formulierung von Beweisplänen. In einigen Fällen können die Entscheidungsprozeduren für Theorien auch Gegenbeispiele für nicht beweisbare Formeln erzeugen. (Siehe etwa [17].)
   Unserer Erfahrung nach wird bei der Entwicklung korrekter Software die meiste Zeit für die Beweise benötigt. Um diesen Aufwand zu senken, haben wir bei der Entwicklung des Beweisers darauf geachtet, daß er mit einem hohen Automatisierungsgrad arbeitet.

Weiterentwicklungen während KORSO. Im Rahmen dieses Projekts haben wir das System um Komponenten erweitert, die die Verifikation während der Entwurfsphase der Software erlauben. (Siehe Abschnitt 3.) Für Terminierungsbeweise wurde eine Benutzerunterstützung implementiert. Der Benutzer hat die Möglichkeit, selbst Regeln, die vom Beweiser angewandt werden, anzugeben und kann Beweispläne formulieren. (Abschnitt 6) Schwerpunkt unserer Arbeiten war die Entwicklung von Hilfsmitteln zur Angabe von Gegenbeispielen bei nicht beweisbaren Formeln. (Abschnitt 5) Dazu waren bei allgemeinen Formeln, also solchen, bei denen die im Beweiser benutzten Entscheidungsprozeduren für Theorien nicht anwendbar sind, Grundlagenarbeiten notwendig. Wir haben deshalb auch Tableaus mit Ordnungsrestriktionen und die Erzeugung von endlichen Modellen untersucht. (Häufig bemängeln Benutzer das Verhalten von Beweisern, wenn sie auf nicht gültige

Formeln angewandt werden. Dieser Kritik begegnen wir mit Hilfsmitteln zur Erzeugung von Gegenbeispielen.)

Während des KORSO-Projekts haben wir an der Fallstudie *Fertigungszelle* mitgearbeitet. (Siehe Abschnitt 4.) Wir haben ein Steuerungsprogramm entworfen und nachgewiesen, daß alle Sicherheitsanforderungen erfüllt werden.

Im nächsten Kapitel geben wir einen Überblick über das Verifikationssystem *Tatzelwurm*. Anschließend beschreiben wir die Entwicklungen, die im Rahmen des KORSO-Projekts erfolgt sind, genauer.

2. Das Programmverifikationssystem *Tatzelwurm*

Das Verifikationssystem dient zur Verifikation und Entwicklung sequentieller Programe, die in einer imperativen Sprache geschrieben sind. Dabei kann eine Spezifikation Teile eines Programms, die noch nicht implementiert sind, vertreten. Zur Erzeugung von V*erifikationsbedingungen*, die für die partielle Korrektheit hinreichend sind, wird die Hoare-Logik benutzt. Für Terminierungsbeweise wird die Wohlordnungsmethode benutzt. Für alle Beweisaufgaben ist ein mechanischer Beweiser, der auf die Belange der Porgrammverifikation zugeschnitten ist, verfügbar.

2.1. Die Programmiersprache

Das Verifikationssystem benutzt eine Teilmenge von Pascal als Programmiersprache:

1. Prozeduraufrufe, Zuweisungen, bedingte Anweisungen und Wiederholungsanweisungen,
2. Funktionsaufrufe,
3. Konstanten- und Typdefinitionen, Variablen-, Prozedur- und Funktionsdeklarationen,
4. Ganze und Real-Zahlen, Arrays, Records, Unterbereichs und Skalar- bzw. Aufzählungstypen und
5. Module wie in UCSD-Pascal.

In [16] oder [17] ist eine genauere Beschreibung der verwendbaren Programmiersprache zu finden. Der Gebrauch von Pascal ist nicht Bedingung. Der Pascal-Parser kann durch einen Übersetzer für jede andere Programmiersprache ersetzt werden, der lediglich den oben beschriebenen Sprachumfang nicht überschreiten darf und der die Überwachung der anschließend genannten Einschränkungen garantiert

Die zu verifizierenden Programme müssen noch einige zusätzliche Bedingungen erfüllen. Funktionen und Prozeduren sind als Parameter nicht zugelassen. In Prozeduraufrufen darf keine Variable mehr als einmal als Ergebnisparameter vorkommen. Ergebnisparameter dürfen auch nicht als globale Variablen auftreten. (Aliasverbot.) Funktionen dürfen keine Resultatparameter oder globale Variablen besitzen. Schleifeninvarianten müssen den Effekt einer jeden Wiederholungsanweisung beschreiben. Ebenso muß der Effekt einer jeden Prozedur und Funktion unter Verwendung von Bedingungen, für die Argument- und Resultatparameter sowie die globalen Variablen und Konstanten angegeben werden. (Entry- und Exitbedingungen.) Die dazu verwendete Sprache wird im nächsten Abschnitt beschrieben.

2.2. Die Spezifikationsprache

Als Spezifikationsprache dient die sortiere Prädikatenlogik der ersten Stufe mit Gleichheit und Funktionssymbolen. In dieser Sprache sind die Spezifikationen von Programmen, Schleifeninvarianten und die Entry- und Exitbedingunen von Prozeduren und Funktionen anzugeben. Die Operatoren der Programmiersprache haben in der Spezifikationssprache die gleiche Bedeutung. (Dies wird von Entscheidungsprozeduren dies Beweisers ausgenutzt.) Datentypen der Programmiersprache werden in der Spezifikationssprache als Sorten betrachtet. Ein Untertyp S eines Typs T wird als Untersorte von T betrachtet. (Integer ist etwa eine Untersorte von Real.)

2.3. Programme und Spezifikationen

Tatzelwurm verifiziert Programme, die aus Pascalanweisungen und logischen Formeln, den A*nnotationen*. Annotationen können in Programmen auf die Schlüsselworte **entry**, **exit**, **invariant** und **assert** folgen. Die Entry- und Exitannotationen spezifizieren die Wirkung von Programmen, Prozeduren und Funktionen. Invarianten beschreiben die Effekte von Wiederholungsanweisungen of loops. Eine logische Formel X, die dem Schlüsselwort **assert** folgt, heißt $Zusicherung$. In sta_1; **assert** X; sta_2 dient X als Nachbedingung für

sta_1 und als Vorbedingung für sta_2. Unter Verwendung von Zusicherungen kann der Beweis der Korrektheit eines Programms in Teilaufgaben, die unabhängig voneinander bearbeitet werden können, zerlegt werden. Die geeignete Wahl solcher Zusicherungen erlaubt die Vermeidung allzu komplizierter Beweise.

Ein Symbol x, das in einer Annotation als Variable und als Indentifikator in einem Programm vorkommt, bezeichnet das gleiche Objekt. Kommt etwa x in einer Anweisung *sta*, auf die $P(x)$ als Annotation folgt, vor, dann muß $P(x)$ nach Ausführung von *sta* gelten. Das Verifikationssystem zerlegt ein Programm in Folgen (Pfade) von Anweisungen, die mit einer Annotation beginnen und enden. Die Pfade können als Hoare-Tripel der Form $X \mid sta \mid Y$ angesehen werden, aus denen die Verifikationsbedingungen abgeleitet werden. Diese Bedingungen sind für die Korrektheit des Programms hinreichend. (Einzelheiten zur Zerlegung des Programms in Pfade und zur Erzeugung von Verifikationsbedingungen sind in [16] zu finden.)

2.4. Der automatische Beweiser

Der Beweiser, der in [18]) genauer beschrieben ist, benutzt die analytischen Tableaus von Smullyan [24]. Zusätzlich zu den standardmäßig vorhandenen Tableauregeln gibt es Regeln zur Behandlung von Äquivalenzen und eine Verallgemeinerung des Modus-Ponens. Zum Schließen mit Gleichungen und Funktionen dient eine Entscheidungsprozedur für die quantorenfreie Theorie der Gleichheit mit uninterpretierten Funktionssymbolen.

Der Beweiser ist auf die besonderen Eigenschaften, die die Verifikationsbedingungen haben, zugeschnitten. Im allgemeinen sind dies umfangreiche logische Formeln, deren Beweis rascher gefunden wird, wenn Entscheidungsprozeudren für Theorien benutzt werden. (Die Benutzung solcher Prozeduren macht meistens auch Induktionsbeweise unnötig.) Die wesentlichen Eigenschaften des Beweisers, der eine Fülle von Beweistechniken anbietet, sind:

1. Der Benutzer kann entscheiden, ob er den Beweis mit Hilfe der in [24] beschriebenen systematischen Prozedur oder unter Anwendung der Technik der freien Variablen führen möchte. (Tableaus mit freien Variablen sind in [10] beschrieben.) Starke Reduktionstechniken, die sowohl a-priori als auch a-posteriori wirken, halten den Suchraum klein. Bei Verwendung der von Smullyan [24] angegebenen systematischen Prozedur sind Heuristiken, die auf der Unifikation basieren, verfügbar.
3. Entscheidungsprozedureen für die quantorenfreien Theorien der Arithmetik ohne Multiplikation [14], der Presburger-Arithmetik [15], der Arrays und Records, der Folgen (Listsen) mit Cons, Car und Cdr sind vorhanden. Außerdem gibt es eine Reduktionsprozedur für die Aufzählungstypen von Pascal [13].
   Die Prozeduren sind berechnungsadäquat. Einfache Formeln werden mit geringem Aufwand bearbeitet. Weiters sind sie so entworfen, daß der Benutzer nicht um die Einzelheiten ihrer Anwendung besorgt sein muß.
4. Für die Behandlung von Gleichungen gibt es eine Fülle von Techniken. Sie können von Reduktionsprozeduren für Theorien bearbeitet werden. Systeme linearer Gleichungen etwa werden mit Hilfe der Gaußelimination gelöst. Termersetzungssysteme können zur Demodulation verwendet werden. Unter Verwendung von Unifikation werden universell abgeschlossene Gleichungen zur Feststellung, ob ein Zweig des Tableaus geschlossen werden kann benutzt. Ebenso wird Unifikation angewandt, um die Gleichheit beliebiger variablenfreier Terme abzuleiten.
5. Der Beweiser kann vollautomatisch ohne Steuerung durch den Benutzer verwendet werden. Doch besteht auch die Möglichkeit, Beweise interaktiv zu führen. Dabei kann der Beweiser die zu bearbeitenden Formeln auswählen und Einsetzungen für Variablen angeben. Auch ist die Angabe zusätzlicher Hilfssätze und Definitionen möglich, sodaß in dieser Betriebsart der Beweiser eher als Beweisentwicklungssystem benutzt wird.

3. Ausbau von Tatzelwurm zu einer Programmentwicklungsumgebung

Das Verifikationssystem Tatzelwurm wurde ursprünglich als Post-Verifikationssystem entwickelt: Man geht von Programmen aus, die bereits bis ins letzte Detail ausformuliert sind. Wird in einem solchen Programm bei der Verifikation ein Fehler ge-

funden, kann dies zur Folge haben, daß frühzeitig gemachte Entwurfsentscheidungen geändert werden müssen und die Wiederholung eines Teils der oder sogar der ganzen Programmentwurfsphase notwendig ist. (In jedem Fall wird der Fehler zu spät entdeckt.) Deshalb war es notwendig, das Verifikationssystem so zu erweitern, daß es schon in der Entwurfsphase benutzt werden kann. Hier wird man für bestimmte Programmteile oder Prozeduren lediglich spezifizieren, was sie tun sollen. Mit Hilfe dieser Spezifikationen wird man dann nachweisen, daß der Algorithmus, der diese Teile bzw. Prozeduren enthält, die gewünschten Eigenschaften hat. Ist dies geschehen, besteht der nächste Schritt im Entwurf von konkreten Programmen für die nur spezifizierten Teile und Prozeduren. Es genügt dann, deren Korrektheit gegenüber den für sie gegebenen Spezifikationen nachzuweisen.

Diese Vorgehensweise muß vom Programmverifikationssystem in geeigneter Weise unterstützt werden. Häufig zeigt sich bei größeren Programmen, daß der Weg nicht geradlinig von der Spezifikation zum konkreten Programm verläuft. Vielmehr erweist es sich immer wieder als notwendig, bestimmte Entwurfsentscheidungen rückgängig zu machen oder zumindest zu modifizieren. Die Verwaltung des daraus resultierenden Nebeneinanders von verifizierten und noch zu verifizierenden Programmteilen, sowie von verschiedenen Versionen von ihnen muß vom Programmverifikationssystem unterstützt werden. Dies bedingt, daß Projekt- und Datenbankaspekte berücksichtigt werden müssen.

Bei der Verfeinerung von Programmentwürfen haben wir vorausgesetzt, daß diese „spekulativ" erfolgt. An Hand der Spezifikation wird Software entworfen, von der anschließend gezeigt werden muß, daß sie die Anforderungen erfüllt. (Die in einem Verfeinerungsschritt entworfenen Programme müssen selbstverständlich noch nicht vollständig ausformuliert sein.) Diese Vorgehensweise bietet den Vorteil, daß bei Erstellung großer Software bereits vorhandene mitbenutzt werden kann. Sobald ihre Funktionalität spezifiert ist, kann der Programmteil, in dem sie benutzt wird, verifiziert werden. Es bleibt dann dem Anwender die Entscheidung überlassen, wann die bereits vorhandene Software verifiziert wird.

Die Methode der spekulativen Verfeinerung bietet den Vorteil, daß im Gegensatz zur Programmentwicklung durch Transformation, Unzulänglichkeiten der Spezifikation nicht systematisch propagiert werden. Dazu ist allerdings Voraussetzung, daß der Programmentwurf und die Spezifikation einer kritischen Inspektion sowohl unter dem Gesichtspunkt der Programmentwicklung als auch der Anwendung unterworfen werden.

3.1. Versionsverwaltung

Die zu verwaltenden Objekte sind Spezifikationen, teilweise fertiggestellte Implementierungen, Verifikationsbedingungen, Beweise und Unterlagen zur Dokumentation. Eine Ansammlung solcher Daten, die sich auf denselben Programmteil, (etwa eine Prozedur,) beziehen, bezeichnen wir als *Programmunit*. Daneben werden noch Hilfsmittel für Beweise wie etwa Lemmata oder Definitionen verwaltet. Solche Objekte werden *logische Units* genannt. Auch bei logischen Units werden Beweis oder Dokumentation nicht als eigenständige Objekte verstanden, sondern als Attribute von Units behandelt. Zwischen Units können logische Abhängigkeiten bestehen, etwa wenn eine Definition zum Beweis eines Lemmas verwendet wurde oder eine Programmunit eine Prozedur aus einer anderen Programmunit importiert. Um die Korrektheit des Gesamtsystems zu gewährleisten, muß die Zyklenfreiheit der Abhängigkeitsrelation sichergestellt werden. Beim Übersetzen von neuen oder modifizierten Progammteilen und beim Beweisen von Beweisverpflichtungen wird über die Abhängigkeitsrelation Buch geführt. Das Verifikationssystem überprüft bei jeder Änderung die Zyklenfreiheit der Relation. Neben diesen verifikationstypischen Aufgaben wird auch die übliche Funktionalität von Versionsverwaltungssystemen bereitgestellt, z.B. Herstellung von alten Versionen, Zusammenführung verschiedener Versionen, Regelung des Mehrbenutzerbetriebs, Zugriff auf Modulbibliotheken. Ein wesentlicher Teil der Funktionalität der Versionsverwaltung stützt sich auf RCS [25] ab .

Im Rahmen der Fallstudie Fertigungszelle - siehe Abschnitt 3 - wurden die hier entwickelten Konzepte an einem größeren Projekt erfolgreich erprobt.

3.2. Die Abhängigkeitsrelation zwischen Units

Es gibt zwei Arten von Abhängigkeiten zwischen Units. Im ersten Fall wird eine logische Unit *B* im Beweis einer anderen (logischen oder Programm-)

Unit *A* verwendet. Im anderen Fall ist Unit *A* von Unit *B* abhängig, weil *A* von *B* Programmteile importiert. (UCSD-Pascal erlaubt neben dem Export von Funktionen und Prozeduren auch den Export von Konstanten,Typen und Variablen.) Hier müssen beide Units Programmunits sein. Für die Verwaltung der Units spielt dieser Unterschied allerdings keine Rolle. In beiden Fällen handelt es sich um eine binäre Relation, die zyklenfrei sein muß.

Aus der Sicht der Programmverifikation mit *Tatzelwurm* wird beim Import von Programmteilen nicht deren Implementierung sondern deren Spezifikation benutzt. Es hat sich bewährt, dafür zwei Varianten von Importschnittstellen für Programmunits bereitzustellen. Zum einen kann eine statische Importschnittstelle verwendet werden. Bei diesem Konzept werden die Eigenschaften des importierten Programmteiles im importierenden Modul noch einmal explizit spezifiziert. Innerhalb des importierenden Moduls wird dann immer die eigene Spezifikation benutzt. Änderungen im exportierenden Modul werden nicht über Modulgrenzen hinweg propagiert. Zusätzliche Beweisverpflichtungen stellen sicher, daß zum Beispiel die Exportspezifikation einer Prozedur stärker ist als die entsprechenden Importspezifikationen der Module, die diese Prozedur verwenden. Höhere Flexibilität und geringeren Schreibaufwand bietet der dynamische Import, der vor allem bei kleinen Units verwendet wird. Ähnlich einer Insert-File-Anweisungen wird immer die neueste verfügbare Version einer Prozedurspezifikation importiert und direkt in die Beweisverpflichtungen der importierenden Unit eingebaut.

Der Ausbau zur Programmentwicklungsumgebung erfolgte im Rahmen einer Diplomarbeit [1].

4. Die Fallstudie *Fertigungszelle*

Im Rahmen dieser Fallstudie wollten wir die Tragfähigkeit der von uns in den Schwerpunkten 2.1 und 2.4 entwickelten Techniken nachweisen. Dazu haben wir in Zusammenarbeit mit dem Korso-Partner am FZI in Karlsruhe in *Tatzelwurm*-Pascal ein Programm geschrieben, das ein am FZI entwickeltes Modell der Fertigungszelle [19] steuern kann. (Es bildet eine in Karlsruhe industriell benutzte Fabrikationsanlage nach.)

Die Fertigungszelle besteht aus Geräten wie Fließbänder oder Roboterarme, die der Beförderung und Bearbeitung der Werkstücke dienen. Der steuernde Rechner kann Sensoren abfragen und mittels spezieller Kommandos Motoren und Elektromagnete ein- und ausschalten. Die Hauptschwierigkeit bestand im Erstellen einer verifikationsgerechten logischen Beschreibung der einzelnen Geräte und deren Zusammenwirken. Daneben mußten die Sicherheitseigenschaften und Eigenschaften des Steuerungsprogramms selbst spezifiziert werden. Von anderen Korso-Partnern erstellte Spezifikationen waren aus unterschiedlichen Gründen für unserer Zwecke nicht brauchbar. Meistens wurden weder das Protokoll zur Rechner-Geräte-Kommunikation noch Sicherheitseigenschaften ausreichend berücksichtigt.

Die Bearbeitung dieses Problems gliederte sich in drei Teile:

1. Beschreibung der Funktionsweise der Fertigungszelle und der Sicherheitseigenschaften in Prädikatenlogik 1. Stufe mit Gleichheit
2. Erstellen eines mit Zusicherungen versehenen Steuerprogramms
3. Verifikation des Steuerprogramms bezüglich der Sicherheitseigenschaften.

Die Fertigungszelle hätte als ein einziger endlicher Automat beschrieben werden können. Dies wäre wegen der großen Anzahl der Zustände unzweckmäßig gewesen. Dagegen bietet unsere Spezifikationstechnik mit Hilfe der Prädikatenlogik erster Stufe die Möglichkeit, jedes einzelne Gerät als endlichen Automaten zu spezifizieren und nur bei bestimmten Zustandsübergängen den Gesamtzustand der Fertigungszelle zu berücksichtigen. Durch Einführung eines Zeitparameters konnten wir zusätzlich zeitliche Aspekte ausdrücken.

Es wurden zwei verschiedene Steuerprogramme spezifiziert und implementiert. Die erste, einfache Version arbeitet mit Fallunterscheidungskaskaden und kann lediglich ein Werkstück zur gleichen Zeit überwachen. Im zweiten Programm ist die Anzahl der Werkstücke im Wesentlichen durch die Kapazität der Geräte begrenzt. Hier ist die Steuerung in einer Entscheidungstabelle kodiert, die mit Hilfe eines einfachen Interpretierers an Hand der Vorgängerzustände und der Sensorwerte ausgewertet wird. Diese

Vorgehensweise besitzt eine größere Flexibilität des Programms gegenüber Erweiterungen oder Änderungen im Gegensatz zur Implementierung mittels Fallunterscheidungskaskaden. Zudem sind Entscheidungstabellen übersichtlicher als Fallunterscheidungskaskaden. Das erste Programm wurde vollständig verifiziert. Beim zweiten Programm wurden exemplarisch einige Beweisaufgaben verifiziert.

Da zur Implementierung in Pascal insbesondere Verbunde, Felder und endliche Mengen verwendet werden, stellt dieses Steuerungsprogramm große Anforderungen an die Theorieprozeduren, die im Beweiser von *Tatzelwurm* verwendet werden. Die im Rahmen des Korso-Projekts neu entwickelten Hilfsmittel zur Modularisierung eines Programms und die Möglichkeit von Beweistaktiken zu verwenden wurden in diesem Fallbeispiel erfolgreich angewandt.

Um den Aufwand für die Verifikation vertretbar zu halten, wurde besonderes Augenmerk auf hohen Automatisierungsgrad und Modularisierung gelegt. Die Modularisierung der Implementierungsmodule wurde mit dem Unit-Konzept durchgeführt, das sowohl Top-Down- als auch Bottom-Up-Entwicklung erlaubt. Durch Aufteilung der Spezifikation in Axiome und Definitionen wurde die Spezifikation gegliedert. Implementierung und Spezifikation konnten mit den beschriebenen Mitteln adäquat behandelt werden. Die Gliederung der Beweisaufgaben erwies sich jedoch als schwieriger. Mit abgeleiteten Regeln und Lemmata standen mächtige Hilfsmittel zur Verfügung. Deren Einsatz wirft jedoch ähnliche Probleme auf, wie sie aus dem Bereich des Softwareengineering bekannt sind. Ist die vorgegebene Gliederung zu grob, sinkt der Automatisierungsgrad drastisch. Zu feine Gliederung erfordert überproportional hohen Arbeitsaufwand vor Beginn des eigentlichen Beweises. Um das richtige Maß der Gliederung zu finden, bedarf es der Erfahrung, die nur im Rahmen von weiteren Verifikationsaufgaben erworben werden kann.

Die Ergebnisse und Erfahrungen sind in [5] beschrieben.

5. Analyse fehlgeschlagener Beweise

Bislang hat man sich beim automatischen Beweisen meist auf das Nachvollziehen von gültigen Aussagen beschränkt. In der Programmverifikation liegt aber eine gänzlich andere Situation vor. In der Regel wird eine Verifikationsbedingung zunächst nicht gültig sein, was zur Folge hat, daß ihr Beweis fehlschlagen wird.

Tableaubeweiser wie der von Tatzelwurm erlauben in vielen Fällen, aus dem gescheiterten Beweisversuch Gegenbeispiele für die Verifikationsbedingung, die bewiesen werden sollte, anzugeben. Solche Gegenbeispiele sind äußerst wertvoll bei der Suche nach der Ursache des Scheiterns, die im Programm oder seiner Spezifikation begründet sein kann. Unser Ziel war es, möglichst weittragende Methoden zur Ermittlung solcher Gegenbeispiele zu entwickeln und in das System einzubauen.

Hier tritt ein erhebliches theoretisches Problem auf: Grundsätzlich sind die Fragen, ob eine prädikatenlogische Formel gültig oder erfüllbar ist, unentscheidbar. Andererseits sind eine Reihe von Formelklassen und Theorien bekannt, bei denen diese Fragen doch entscheidbar sind. Wir hatten zuerst die Hoffnung nach Untersuchung dieser Klassen Hinweise auf die Entwicklung und Verwendung von Entscheidungsprozeduren für solche Formelklassen zu gewinnen. Dieser Weg erwies sich als nicht gangbar, da die Ergebnisse über entscheidbare Klassen logischer Formeln an Hand von Normalformen dieser Formeln gewonnen wurden. Die Benutzung von Normalformen widerspricht aber dem Entwurfsziel des Beweisers von Tatzelwurm, der vom Benutzer auch interaktiv verwendet werden soll. Normalformen würden die interaktive Ermittlung eines Beweises ungeheuer erschweren.

5.1. Allgemeine Methoden zur Ermittlung von Gegenbeispielen

Einen neuen Ansatz zur automatischen Erzeugung von Gegenbeispielen stellen die Arbeiten von Fermüller u. a. [9] dar. In ihnen wird gezeigt, daß bestimmte entscheidbare Teilklassen der Prädikatenlogik erster Stufe mit Hilfe der A-Resolution von Joyner [11] entschieden werden können. Wesentlich sind dabei zwei Eigenschaften der A-Resolution: Sie ist für die gesamte Prädikatenlogik erster Stufe vollständig, und das Wachstum der Größe bestimmter Terme kann eingeschränkt werden. Zudem ermöglicht die A-Resolution gegenüber der gewöhnlichen Reso-

lution eine erhebliche Verkleinerung des Suchraums.

Zunächst mußten wir zeigen, daß die Ordnungsrestriktionen unter Erhalt der Vollständigkeit auf tableaubasierte Beweisverfahren übertragen werden können. Bei der Resolution werden alle Zwischenergebnisse - die Resolventen - für den gesuchten Beweis berechnet und gespeichert. Tableauverfahren mit Unifikation müssen demgegenüber mit Rücksetztechniken arbeiten. Dies erschwert die direkte Übertragung der Ergebnisse für die Resolution auf Tableaus. Zuerst untersuchten wir eine einheitliche Darstellung von Resolution und Tableaus, die Vollständigkeitsbeweise für beide Verfahren erlaubt und mit Ordnungsaspekten verbunden werden kann. Dazu modifizierten wir die Technik der Vollständigkeitsbeweis mittels semantischer Bäume so, daß nicht mehr der gesamte Baum sondern nur bestimmte Teilbereiche von ihm im Vordergrund stehen. Unter Benutzung dieses Konzepts konnte die Vollständigkeit der A-geordneten Tableauenumerationsmethode bewiesen werden. (Siehe [6].)

5.2. Erzeugung endlicher Gegenbeispiele

Eine weitere Möglichkeit zur Analyse fehlgeschlagener Beweise bietet die Erzeugung endlicher Gegenbeispiele. (Nach Untersuchung von FINDER [25], einem System, das für logische Formeln endliche Modelle erzeugt, erschien uns diese Methode sehr erfolgversprechend.) Da dieses Gebiet trivialerweise entscheidbar ist, lag der Schwerpunkt der Arbeit auf der Erstellung effizienter Entscheidungsverfahren. Natürlich sollte auch die Erzeugung endlicher Gegenbeispiele zu den bisher vorhandenen tableauorientierten Methoden passen. Es wurde ein neuartiges Verfahren entworfen, das auf der Revision von Interpretationen beruht. Funktions- und Prädikatensymbolen werden dabei Urbild- und Bildmengen zugeordnet, die der Revision unterworfen sind. Dieser Ansatz ermöglicht es, Mengen von Interpretation mit endlichem Universum zusammenzufassen, was als Grundlage für ein effizientes Verfahren unabdingbar ist. Die Verzahnung mit einem Widerlegungskalkül zur Beweissuche wurde durch die Einbettung in das gewöhnliche Tableauverfahren erreicht.

5.3. Verwendung von Entscheidungsprozeduren für Theorien

Auch die Benutzung von Entscheidungsprozeduren für Theorien erlaubt bei bestimmten Klassen logischer Formeln zu erkennen, ob ein Beweis scheitert. Für einige quantorenfreie Theorien werden solche Prozeduren schon jetzt im Beweiser angewandt. (Siehe [18] Im Rahmen des Projekts konnten auch die Grundlagen für die Anwendung der nicht quantorenfreien Theorien der Arithmetik ohne Multiplikation geklärt werden. (Siehe [4].) Das Verfahren wird derzeit optimiert und implementiert.

Ebenso sind die Grundlagen zur Kombination dieser Entscheidungsprozedur mit Entscheidungsprozeduren für andere Theorien geklärt. Zwar sind die meisten in Programmiersprachen anzutreffenden nicht quantorenfreien Theorien unentscheidbar. Es lassen sich jedoch interessante Teiltheorien festlegen, die noch entscheidbar sind. Wendet man die Kombination der Entscheidungsprozeduren auf eine Formel einer nicht entscheidbaren Theorie an, besteht der Gewinn immerhin in kürzeren Beweisen. Im Falle eines Scheiterns eines Beweisversuchs ist dann der Benutzer auch eher in der Lage, die Ursachen des Fehlschlags in vernünftiger Zeit zu finden.

Im Rahmen einer Diplomarbeit [8] wurden die zentralen Teile eines Verfahrens zur Entscheidung der quantorenfreien Theorie der Arithmetik mit Funktionssymbolen implementiert. Es verallgemeinert die in der Literatur bekannten Algorithmen erheblich. (Siehe etwa [21].)

6. Beweispläne

Da längerfristig nicht zu erwarten ist, daß mechanische Beweiser in der Lage sind, jeden Beweis vollautomatisch zu finden, wurde die Kommandoschnittstelle des Beweisers von Tatzelwurm zu einer Eingabeschnittstelle für Beweispläne erweitert.

Zur Formulierung von Beweisplänen gibt es schon seit längerem Hilfsmittel, nämlich Beweistaktiken und Strategien. Doch wurden diese für Beweiser, die Kalküle des natürlichen Schließens benutzen, verwendet. (Siehe etwa [22].) Viele der Probleme, die dort gelöst werden, treten in Tableaubeweisern nicht auf, während umgekehrt für die Schwierigkeiten, die mit der Benutzung von Tableaubeweisern verbunden sind, Beweistaktiken für Kalküle des natürlichen Schließens unzureichende Lösungen liefern. Dies ist hauptsächlich darin begründet, daß bei vielen taktischen Theorembeweisern Taktiken notwendig sind,

damit die konkrete Bearbeitung einer Aufgabe überhaupt möglich wird. Im Gegensatz dazu ist der Beweiser von *Tatzelwurm* grundsätzlich in der Lage, für jede gültige Formel einen Beweis zu finden. Beweispläne dienen lediglich dazu, den Suchraum einzuschränken, damit der Beweis mit geringerem Aufwand gefunden wird.

Nach Untersuchung einer Reihe von Beispielen entschlossen wir uns zur Einschränkung des Suchraums und damit verbunden, zur Steigerung des Automatisierungsgrades des Beweisers, dem Benutzer die Möglichkeit zu geben, selbst Regelschemata zu formulieren. Zur Entlastung des Benutzers bei der Steuerung des Beweisers sollte eine Beweisplansprache dienen, die die besonderen Eigenschaften der Tableaubeweiser berücksichtigt.

6.1. Regelschemata

Regelschemata, die man als abgeleitete Regeln eines Kalküls ansehen kann, spielen in einem Beweis die Rolle von Hilfssätzen. Im Gegensatz zu Hilfssätzen, die vom Beweiser als beliebige logische Formeln angesehen werden und deren Anwendung dann auch in Situationen versucht wird, in denen dies unsinnig ist, kann bei Regeln die Art des Gebrauchs näher festgelegt werden. Damit ist es bei Verwendung von Regeln möglich, zweckmäßige Einsetzungen für gebundene Variable zu bestimmen oder zu entscheiden, ob es vor einer Fallunterscheidung sinnvoll ist, noch Einsetzungen für gebundene Variablen zu machen. Weiters kann festgelegt werden, wie oft eine Regel in einem Beweis angewandt werden soll und was zu tun ist, wenn eine bestimmte Regel nicht anwendbar ist. Damit erlauben Regelschemata die Verkleinerung des Suchraums und steigern mithin den Automatisierungsgrad bei der Ermittlung von Beweisen.

Um die Regelanwendung zu ermöglichen, war es notwendig, einen Matchingalgorithmus für logische Formeln zu implementieren. Schließlich war zu berücksichtigen, daß die Untersuchung, ob eine Regel anwendbar ist, die Arbeitsgeschwindigkeit des Beweisers nicht verlangsamt, denn meistens sind Regeln nicht anwendbar. Im Anwendungsfall muß die Ansteuerung einer passenden Regel rasch erfolgen. Schließlich mußte das Fail-Safe-Verhalten des Beweisers zu bewahrt bleiben: Die Automatisierung der Beweise muß den Benutzer von Routinearbeiten befreien. Sobald ein Beweis nicht mit automatischen Mitteln allein gefunden werden kann, steht dem Benutzer eine Kommandoschnittstelle zur Verfügung, die ihm erlaubt, den Beweiser händisch zu steuern und einen Beweisplan zu entwickeln. Diese Ziele haben wir unter Verwendung objektorientierter Techniken erreicht.

6.2. Die Beweisplansprache

An Hand von Beweisaufgaben aus früheren Verifikationsprojekten und der Fallstudie „Fertigungszelle" haben wir eine Sprache zur Formulierung von Beweisplänen definiert.

Die wichtigsten Kommandos dieser Sprache sind

1. Kommandos zum Laden von Hilfssätzen, Definitionen und benutzerdefinierter Regeln
2. Definitionen von Formelschemata, die in Bedingungen benutzt werden können.
3. Iterations- und bedingte Kommandos. In diesen, von Bedingungen gesteuerten Kommandos stehen Hilfsmittel zur Inspektion des Beweiszustandes zur Verfügung.
4. Blockieren und Aktivieren von Formeln. Damit kann der Benutzer für bestimmte, von ihm gewählte Abschnitte die Anwendung von Regeln auf diese Formeln unterdrücken.

Blockieren und Aktivieren von Formeln stellen sehr starke Hilfsmittel zur Einschränkung des Suchraums dar. Bei sorgfältiger Benutzung erlauben sie eine erhebliche Verminderung des Aufwands bei der Ermittlung von Beweisen. Werden sie nicht sorgfältig angewandt, besteht die Gefahr, daß der Beweis nicht gefunden wird. Hier wird noch einmal der schon weiter oben angedeutete Unterschied zu den Beweisstrategien in taktischen Theorembeweisern deutlich.

Die schon vorhandenen Kommandos, mit denen ein Benutzer einen Beweis interaktiv steuern kann, haben wir ebenfalls in die Beweisplansprache mit aufgenommen. Zur Eingabe von Beweisplänen dient die Dialogschnittstelle zur interaktiven Eingabe von Beweiserkommandos, die schon vor Beginn des Korso-Projekts zur Verfügung stand. Zusätzlich können Beweispläne auch aus einer Datei gelesen werden, sodaß der Benutzer nach Angabe des Dateinamens den Fortgang des Beweises nicht interaktiv überwachen muß. Da Beweispläne meist im Dialog an Hand einer zu beweisenden Formel entwickelt werden, kann sich der Benutzer die einzelnen Kommandos dieser Pläne in

Dateien aufzeichnen lassen. Diese können dann bei der Wiederholung (Replay) des Beweises oder bei der Bearbeitung ähnlicher Formeln benutzt werden.

7. Weiterentwicklung des Beweisers

Schon auf Grund von Erfahrungen aus früheren Verifikationsprojekten war uns bekannt, daß die Beweisarbeiten, die meiste Zeit in Anspruch nehmen. Während der ganzen Laufzeit des Korso-Projekts wurde stets unter Verwendung von Beispielen untersucht, wie sich neue Lösungen bewährt haben. Dabei wuchs der Umfang der betrachteten Probleme, was zu längeren Zeiten für die Bearbeitung der Beweise führte. Es war also notwendig, die Leistungsfähigkeit des Beweisers zu erhöhen, damit dieser nicht zum Nadelöhr im System wurde.

7.1. Allgemeine Implementierungsarbeiten

Wiederbenutzung von Beweisschritten während eines Beweises. Dies erlaubt die schnellere Ermittlung von Beweisen - vor allem dann, wenn Reduktionsprozeduren für Theorien benutzt werden. Die damit verbundene Möglichkeit Strukturüberlappungen zu nutzen, hat zudem geringeren Platzbedarf zur Folge.

Lookahead. In bestimmten Situationen ist es sinnvoll, im Rahmen einer Vorausschau über die als nächstes möglichen Beweisschritte, zweckmäßige Tableauerweiterungsschritte zu ermitteln. Hierzu erwies sich die Implementierung eines Unifikationsalgorithmus, der auch Gleichungen berücksichtigt, als zweckmäßig.

Quantorenfreie Theorie der endlichen Folgen (*Listen*). Die Entscheidungsprozedur für diese Theorie mußte sorgfältig optimiert werden.

Quantorenfreie Theorie der Aufzählungstypen von Pascal. Für Teile dieser Theorie mußte eine Entscheidungsprozedur geschrieben werden. Sie wurde in der Fallstudie „Fertigungszelle" benutzt.

7.2. Implementierung einer Entscheidungsprozedur für die Theorie der Records

In Anlehnung an eine Reduktionsprozedur für die quantorenfreie Theorie der Arrays wurde eine Entscheidungsprozedur für die Records implementiert. Es stellte sich jedoch sehr bald heraus, daß Records nicht nur als Datentyp zweckmäßig sind, sondern auch in Spezifikationen die Rolle kartesischer Produkte übernehmen können. Dies erforderte eine Erweiterung der Prozedur. Während die theoretischen Probleme wie vorgesehen erledigt werden konten, traten bei der anschließenden Erprobung an umfangreichen Fallbeispielen Effizienzprobleme auf. Deren Untersuchung und Behebung erwies sich als sehr langwierig.

7.3. E-Unifikation

Bei der Ermittlung der Einsetzung für allgebundene Variable gibt es zwei Techniken.

1. Systematische Einsetzung von Grundtermen.
   Bei Anwendung von Entscheidungsprozeduren für Theorien, einschließlich der Gleichheit wird diese Technik benutzt. Durch Vorausschau, (siehe 6.1) werden möglichst gut geeignete Grundterme ausgewählt.
2. Einsetzung von freien Variablen.
   Die Belegung dieser Variablen mit Grundtermen erfolgt mit Hilfe der Unifikation. Gleichungen dürfen in den Formeln, die mit dieser Beweistechnik bearbeitet werden nicht vorkommen.

Die E-Unifikation erlaubt die Einsetzung von freien Variablen auch dann, wenn Gleichungen vorhanden sind. Dabei ist die E-Unifikation mit Hilfe der Lazy-Paramodulation unserem Tableaubeweiser besonders angemessen. Die Grundlagen für ihre Anwendung sind erarbeitet. Die Implementierung und Optimierung dieser Methode erfolgt im Rahmen einer Studienarbeit.

7.4. Knuth-Bendix-Vervollständigung

Die Knuth-Bendix-Vervollständigung dient zum Schließen in gleichungsdefinierten Theorien, kann aber zur Verkürzung von Beweisen, in denen Gleichungen benutzt werden, zur schnelleren Ermittlung des Beweises beitragen. Man verwendet sie hier als Methode zur Ableitung von Hilfssätzen, also Gleichungen, die aus den bereits gegebenen folgen.

Bei dieser Vorgehensweise, bereitet die Möglichkeit, daß die Vervollständigung nicht zu terminieren braucht, keine Schwierigkeiten. Wird der Beweis gefunden, dann hat der Benutzer die Möglichkeit, das Resultat der Vervollständigung abzuspeichern, um es in ähnlichen Beweisen wieder zu verwenden. Wird kein Beweis gefunden, ist die bearbeitete Formel also kein Theorem, muß der Benutzer ohnehin irgendwann die Beweissuche abbrechen.

Weiters können bei der Vervollständigung Gleichungen entstehen, die nicht gerichtet werden können. Hier kann der Benutzer entscheiden, ob und in welcher Art die Gleichung als Ersetzungsregel verwendet werden soll.

Die Grundlagen für die Anwendung der Vervollständigungsprozedur sind geklärt. Mit der Implementierung wurde begonnen.

8. Terminierungsbeweise

Während der Laufzeit des Korso-Projekts wurde bemängelt, daß der Anwender bei der Benutzung von *Tatzelwurm* keine Unterstützung beim Führen von Terminierungsbeweisen erhält. Zum Nachweis der Terminierung wenden wir die Wohlordnungsmethode an. Aufgabe des Benutzers ist die Angabe einer Funktion, von der zu zeigen ist, daß sie nach unten beschränkt ist und daß ihr Wert in Abhängigkeit vom Programmzustand bei jedem Iterations- bzw. Rekursionsschritt streng monoton abnimmt. Den Kalkül zur Erzeugung von Verfikationsbedingungen haben wir um Regeln zur Erzeugung von Bedingungen erweitert, die für die beiden Anforderungen an die Funktion hinreichend sind.

9. Zukünftige Arbeiten

Wie schon erwähnt konnten wir bei der Erzeugung von Gegenbeispielen für allgemeine logische Formeln erst Grundlagenfragen klären. Die Anwendung dieser Ergebnisse und deren Einbau in den Beweiser wird derzeit erledigt. Schließlich muß die derzeit provisorische Anbindung des Verfahrens zur Entscheidung von Formeln der Arithmetik ohne Multiplikation verbessert werden. Wie schon angedeutet kann es Spezialfall eines allgemeineren angesehen werden, dessen Integration in den Beweiser untersucht werden muß.

Die von uns entworfene Sprache für Beweispläne wurde unter Verwendung der Erfahrungen aus früheren Verifikationsprojekten und der Fallstudie „Fertigungszelle" entwickelt. Im Rahmen weiterer Experimente muß geklärt werden, ob noch weitere Kommandos in die Sprache aufgenommen werden müssen. Außerdem sind Optimierungsarbeiten notwendig, die auch nur im Rahmen von Experimenten erfolgen können.

Schließlich müssen die Techniken zur Vorausschau (Lookahead) noch erheblich verallgemeinert werden. Im Rahmen einer Diplomarbeit [7] wurden dazu im Hinblick auf Schwierigkeiten, die mit der Benutzung von Sorten verbunden sind, Methoden entwickelt, deren Anwendung bei der Bestimmung von Einsetzungen für gebundene Variable erfolgversprechend zu sein scheint.

Eine Erprobung des Verifikationssystems im Rahmen der Protokollverifikation ist vorgesehen.

Literatur

Im Rahmen des KORSO-Projekts entstandene Arbeiten

1. Baur, M: Projektverwaltung zur Entwicklung verifizierter Software. Diplomarbeit. Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe: 1994
2. Käufl, Th. The Prover of the Program Verification System *Tatzelwurm*. Workshop Theorem Proving with Analytic Tableaux and Related Methods. Interner Bericht 8/92. Institut für Logik, Komplexität und Deduktionssysteme. Universität Karlsruhe: 1992
3. Käufl, Th. The Program verifier *Tatzelwurm*. Proc. of the 10th Annual Symposion on Theoretical Aspects of Computer Science. Springer, Lecture Notes on Computer Science: 1993; pp. 708-709
4. Käufl, Th. The Combination of a Decision Procedure for the Presburger Arithmetic with a Tableau Prover. 12th Int. Conf. on Automated Deduction. Workshop Theory Reasoning in Automated Deduction. 1994
5. Klingenbeck, S., Käufl Th. Tatzelwurm. Verification of Safety Requirements with a Program Verification System. in Case Study "Production Cell". A Comparative Study in Formal Software Development. C. Lewerentz, Th. Lindner (Hg.). Karlsruhe 1994; FZI-Publication 1/94
6. Klingenbeck, S., Hähnle, R. Semantic Tableaux with Ordering Restrictions. Proc. of the 12th Int. Conf. on Automated Deduction. Springer, Lecture Notes on Artifical Intelligence: 1994
7. Rieger, U. Bestimmung minimaler Mengen von Herbrandkonstanten für Tableaubeweise. Di-

plomarbeit. Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe: 1994
8. Wagner, M. Entscheidungsprozeduren für die Korrektheit einer Klasse von Programmen. Diplomarbeit. Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe: 1994

Weitere Literatur

9. Fermüller, C., Leitsch, A., Tammet, T., Zamov, N. Resolution Methods for the Decision Problem. Lecture Notes in Artificial Intelligence. Berlin, Heidelberg, New York: 1993 (Springer)
10. Fitting, M. C. First-Order Logic and Automated Theorem Proving. Berlin, Heidelberg, New York: 1990 (Springer)
11. Joyner, W. H. Resolution Strategies as Decision Procedures. Journal of the ACM, 23, pp. 398-417: 1976
12. Käufl, Th. The Program Verification System Tatzelwurm: User Manual. Unpublished working paper
13. Käufl, Th. Reasoning about Theories with a Finite Model. 3. Österreichische Artificial Intelligence-Tagung 1987. Informatik-Fachberichte; Berlin, Heidelberg, New York: 1987; Springer
14. Käufl, Th. Reasoning about Systems of Linear Inequalities. 9th International Conference on Automated Deduction. Lecture Notes on Computer Science; Berlin, Heidelberg, New York: 1988; Springer
15. Käufl, Th. Simplification and Decision of Systems of Linear Inequalities over the Integers. Interner Bericht 9/88. Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe: 1988
16. Käufl Th. Program Verifier Tatzelwurm: The Correctness and Completeness of the Generation of the Verification Conditions. Interner Bericht 9/89. Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe: 1989
17. Käufl, Th.: The Program Verifier Tatzelwurm. In Sichere Software. H. Kersten, Hrsg. Heidelberg: 1990
18. Käufl Th., Zabel N.: Cooperation of Decision Procedures in a Tableau-Based Theorem Prover. Revue d'Intelligence Artificielle, Vol. 4, no. 3: 1990, pp. 99 - 126
19. Lewerentz C., Lindner Th. Case Study "Production Cell" - A Comparative Study. FZI Publication 1/94. Forschungszentrum Informatik an der Universität Karlsruhe. 1994
20. J: Loeckx, K. Sieber: The Foundations of Program Verification. Stuttgart, Chicester 1984.
21. Mateti, P. A Decision Procedure for the Correctness of a class of Programs. Journal ACM, 28(2), pp. 215-232: 1981
22. Schmidt, D. A Programming Notation for Tactical Reasoning. 7th International Conference on Automated Deduction. Lecture Notes on Computer Science; Berlin, Heidelberg, New York: 1984; Springer
23. Sichere Software: Formale Spezifikation und Verifikation vertrauenswürdiger Systeme. H. Kersten (Hrsg.). Heidelberg: 1990
24. R.M. Smullyan: First Order Logic. Berlin, Heidelberg, New York: 1968
25. Slaney, J. FINDER. Finite Domain Enumerator. Notes and Guide. Centre for Information Science Research. Australian National University: 1993
25. Tichy: RCS-A System for Version Control (Softw. Pract. Exper. 15)