Top-down vs. Bottom-up: Formale Methoden im wissenschaftlichen Wettbewerb

Ein essayistischer Überblick über den Stand der Kunst

Burkhart Wolff

Université de Paris-Sud,
Laboratoire de Recherche Informatique,
Parc Club, 4 Rue Jaques Monod
F-91405 Orsay Cedex, France
Mail: wolff@lri.fr, WWW: http://www.lri.fr/~wolff

Zusammenfassung In diesem Beitrag wird versucht, einen Überblick über den Wettbewerb zweier konkurrierender Forschungsprogramme — genannt *Top-down* oder *Transformationelle Entwicklungsmethodik* versus *Bottom-up* oder *post-hoc Programmverifikation* — im Bereich "Formaler Methoden" zu geben. Als Einordnungsrahmen benutze ich Lakatos's Konzept des "wissenschaftlichen Forschungsprogramms". Es ergibt sich ein bewusst altmodischer Versuch — im Gegensatz zu modischen bibliometrischen Kriterien (vulgo: Zahlen von Veröffentlichungen) — *inhaltliche* Kriterien für die Qualität und damit den Fortschritt der wissenschaftlichen Arbeit in unserem Gebiet zu entwickeln.

1 Einleitung

Man könnte diesen Artikel mit dem Glaubensbekenntnis beginnen lassen: "Formale Methoden helfen dabei, die Sicherheit von Software zu erhöhen". Oder auch inhaltlicher, aber eklektischer: "Formale Methoden sind eine Sammlung von mathematischen Beschreibungsmitteln und Analyseverfahren, die ihre Grundlage in diskreter Mathematik haben". Wir alle haben solche Sätze vermutlich schon oft gehört, und — wenn auch mit Unbehagen — geschrieben.

Statt dessen wird in diesem Beitrag versucht, "Formale Methoden" (FM) als "Wissenschaftliches Forschungsprogramm" zur Fundierung des Software-Engineerings zu begreifen und zwei seiner wichtigsten Unterprogramme — Transformationelle Programmentwicklung vs. post-hoc Programmverifikation — gegenüberzustellen. Als Begriffsrahmen für die Grundkonzepte "Kern", "wissenschaftliches Programm", "Anomalie", etc., dient dabei die Erkenntnistheorie Lakatos's [68].

In der Informatik fängt bekanntlich alles mit Alan Turing an. Als ausgebildeter Mathematiker mit starker Ausrichtung auf logische Grundlagen quälte er sich allerdings nicht mit der neueren Frage, ob er "Formale Methoden" nun benutzen sollte oder nicht bei dem Tun, das wir heute Informatik nennen — er tat es einfach, so wie man Luft zum Atmen braucht. Er entwickelte damit

Konzepte wie Maschine, Zustand, und sogar Zustandsinvariante, die unbestritten zum Grundinventar moderner Programmiersprachentheorien gehören, und beschäftigte sich mit den Zusammenhängen funktionaler und zustandsorientierter Programme. Allerdings galt Turing auch bald als akademischer Nestbeschmutzer, interessierte er sich doch (Pfui!!!) auch für die konkreten Probleme des Computerbaus und seiner Programmierung.[1]

Während die Bedeutung der mathematisch-logischen Grundlagen für die Geschichte der Informatik und für den akademischen Betrieb akzeptierter Fakt ist, ist ihre Relevanz für die ingenieursmässige Konstruktion von Software-Systemen, also das *Software-Engineering*, höchst umstritten, auch im akademischen Betrieb selbst. Sichtbarster Ausdruck für die interne Geringschätzung ihrer Grundlagen ist die Tatsache, das die "curriculum recommandations" der ACM für das Fach Informatik im Jahr 2005 keinerlei mathematische Fähigkeiten mehr erwähnen, und lediglich *einen* Kurs zur Theorie der Programmiersprachen vorsehen[36].

Das akademische Misstrauen an FM wird zudem vertieft durch die Tatsache, das es aus Sicht des mathematischen Mainstreams als Extrem, als Gipfelpunkt eines älteren Forschungsprogramms wahrgenommen wird, das nach allgemeinen Dafürhalten als gescheitert gilt: dem Programm des Formalismus nämlich. Dies begann mit Leipnitz' Idee einer "lingua universalis", einer Sprache in der sich alle wissenschaftlich-mathematischen Probleme formulieren und mittels deduktiven Schliessen beantworten liessen, über erste technische Beiträge durch Boole, Peano und Frege bis hin zu Whitehead und Russels "Principia Mathematica", die das Programm erstmalig konkret ausformulierten. Die Unmöglichkeit der Vollständigkeit und Entscheidbarkeit hinreichend ausdrucksstarker logischer Sprachen, und damit das Ende des Hilbertschen Traums von "Wir müssen wissen, und wir werden wissen!", diskreditierte in den Augen vieler auch den Formalismus als Grundlage der Mathematik. Die Debatte mag in jüngster Zeit wiederaufgeflammt sein [60,61,43,44], angefeuert durch die zunehmende Schwierigkeit, die sich häufenden Monsterbeweise in mathematischen Journalen tatsächlich nachzuprüfen, und diesbezügliche Erfolge durch *mathematische Beweisassistenten*, d.h. interaktive Theorembeweisumgebungen. Doch zunächst einmal wird sich an der allgemeinen Formalismusskepsis nichts ändern, auch aus Gründen der Bequemlichkeit und Opportunität.[2]

---

[1] Hardy verhinderte noch Turings Promotionsvorschlag einer Interpreterimplementierung für den getypten Lambdakalkül mit den Worten "Nothing is worth a Phd that can be done with a computer".[46] Wir wissen heute recht gut Bescheid über das Problem der Normalisierung des getypten Lambda-Kalküls und seiner Bedeutung für die Theorie der funktionalen Programmierung als auch den Bau von Beweissystemen für die Logiken höherer Stufe; auch die technischen Herausforderungen solcher Programme auf der Hardware der fünfziger Jahre würden heute noch Respekt abnötigen. Nur — wer zum Teufel war doch gleich dieser Hardy???

[2] Ich darf den Kollegen Ralf Küsters zitieren mit seinem Ausspruch: "Für mich ist ein Beweis ein Beweis, wenn ich den Gutachter davon habe überzeugen können, das sich daraus vielleicht ein formaler Beweis machen liesse. Das machen alle im Betrieb so!" [53].

2 Exkurs: Imre Lakatos "Kritischer Rationalismus"

Die Abgrenzung der Wissenschaft von Pseudowissenschaft ist ein grundlegendes erkenntnistheoretisches Problem: sollten wir etwa, um ein extremes Beispiel zu gebrauchen, der Erforschung des Einflusses der Bachblütentherapie auf Programmiererteams den Zugang zu öffentlich geförderten Forschungsmitteln gewähren? Der leicht-fertige Einwand: "Nein, das ist pseudo-wissenschaftlich", führt unmittelbar zur Frage, was solche "Ansätze", die sich bei Bedarf mit klingenden Adjektiven wie "ganzheitlich-partizipativ" schmücken lassen, denn inhaltlich von wissenschaftlicher Arbeit unterscheidet. Und weitergehend auch die Frage, was wissenschaftlichen Fortschritt denn ausmacht und wie er sich denn bewerten liesse — eine Frage, die im Zeitalter zunehmender Ökonomisierung, in dem Erfolg in Geld umgerechnet wird und gleichzeitig Geld Mass des Erfolges ist, leicht ein existentielles Beigeschmäckle hat.

Die Frage ist uralt, und hat, wie wir sehen werden, keine Beil-scharfen Lösungen. Bekanntlich schlug Popper[62] folgendes Kriterium vor: "Eine Theorie ist wissenschaftlich, wenn man bereit ist, im Voraus ein entscheidendes Experiment (oder eine Beobachtung) anzugeben, das sie falsifizieren würde, und sie ist unwissenschaftlich, wenn man zu einem solchen 'Experimentum Crucis' nicht in der Lage ist". Demnach kann eine Theorie wissenschaftlich sein, obwohl überhaupt keine Daten für sie sprechen (eine Situation, die gerade die Diskussion um die String-Theorie beschäftigt), oder unwissenschaftlich, obwohl alle Daten für sie sprechen. Zweifellos ist es interessant, einen Marxisten nach dem Experimentum Crucis zu fragen; er wird u.U. die Frage als unmoralisch ablehnen. In der Tat kann also diese Frage einen Anhaltspunkt dafür liefern, ob eine Theorie zu einem Dogma erstarrt ist oder zu echter Erkenntnis beiträgt.

Ist nun das Poppersche Falsifikationskriterim die Lösung des Problems der Abgrenzung von Wissenschaft und Pseudowissenschaft? Nein, denn es berücksichtigt nicht die bemerkenswerte Zählebigkeit wissenschaftlicher Theorien; Poppers Kriterium funktioniert einfach nicht in der wissenschaftlichen Praxis. Wissenschaftler geben eine Theorie nicht auf, nur weil ihr Tatsachen widersprechen oder Schwierigkeiten auftauchen, weil *Anomalien* existieren. Mal diffamieren sie die Anomalie als Monstrosität oder einen irrelevanten Ausnahmefall (Lakatos erfand für diese Verteidigungsstrategie den schönen Namen *Monster-barring*), oder mal finden sie eine rettende Hypothese, die die Anomalie erklären soll (Lakatos' *Monster-adaption*) oder, wenn das nicht möglich ist, vergisst man die Schwierigkeit einfach und wendet sich anderen Problemen zu. Das wohl berühmteste Experimentum Crucis der Wissenschaftsgeschichte, das Michelson-Morley Experiment, das ursprünglich konzipiert war, als Richter zwischen zwei konkurrierenden Äthertheorien zu fungieren, verschwand 25 Jahre in der Schublade, bevor es von Einstein hervorgeholt wurde um damit die spezielle Relativitätstheorie (*kein* Äther) gegenüber der Newton'schen Mechanik auszuzeichnen.

Was ist also das Kennzeichen von guter Wissenschaft? Müssen wir kapitulieren und zugeben das eine wissenschaftliche Revolution nichts als eine irrationale Änderung der Überzeugung, eine religiöse Bekehrung sein? Der amerikanische Wissenschaftstheoretiker Thomas Kuhn kam zu diesem Ergebnis, nachdem er

die Naivität des Popperschen Falsifikationismus erkannt hatte. Da er Kriterien der Unterscheidung zwischem wissenschaftlichem Fortschritt und geistigem Niedergang prinzipiell nicht mehr anerkennen will, gibt es für ihn auch keine allgemeinen Kriterien der Rationalität mehr. Sein Schüler Paul Feyerabend [34] hat diese Position zu einem erkenntnistheoretischen Anarchismus des "Anything Goes" ausgebaut.

Lakatos Beitrag in der Debatte [68] ist es, die Geschichte der Wissenschaft ernst zu nehmen und den Begriff des Fortschritts in der erkenntnistheoretischer Debatte nicht nur auf der philosophischen Spielwiese definieren zu wollen. Für Lakatos macht es keinen Sinn, eine wissenschaftliche Theorie nur als ein konkretes Hypothesengebäude zu begreifen, das man durch eine Anomalie widerlegen könne. Denn alle wissenschaftlichen Theorien sind für ihn Inseln in einem "Meer von Anomalien", und Erkenntnisfortschritt zeichnet sich dadurch aus, das diese Inseln Landgewinne machen. Die genaue Formulierung der Hypothesen kann sich auch über die Zeit verändern, man hat es also eigentlich mit einer Familie von Theorien, T_1, T_2, ... zu tun, eben dem "Forschungsprogramm". Ein Forschungsprogramm zeichnet sich aus durch:

1. einen *harten Kern*, d. i. eine konkrete mathematische, stark formalisierte Theorie, die zäh verteidigt wird, sowie
2. ein System von *Hilfshypothesen*, wozu insbesondere auch die Beobachtungstheorien gezählt werden. Modifikationen im System der Hilfshypothesen gehören zum wissenschaftlichen Tagesgeschäft. Und außerdem:
3. einen *heuristischen Apparat* — oder moderner gesagt: eine Pragmatik — mit der Anomalien im Sinne des Programms gedeutet, also "erklärt" werden.

Zum Kern der Newtonschen Mechanik zum Beispiel gehören die vier Grundaxiome über die Zusammenhänge von Masse, Kraft, Beschleunigung und Gravitation, etc., sowie die mathematischen Theorien, die zu ihrer Fundierung benötigt werden (z.B. Axiomatische Mengenlehre, Theorie der Funktionen, Theorie der Integral- und Differentialrechnung, Differentialgleichungen). Zu den Hilfshypothesen gehören die Theorien über Planetenbewegungen, aber auch Theorien über die Messinstrumente und Beobachtungstheorien über Experimentalanordnungen. Nehmen wir an, eine Anomalie taucht auf: "Teeblätter in einer Teetasse sammeln sich nicht, wie von den Gesetzen der Zentrifugalkraft gefordert, aussen, sondern in der Mitte. Die Newtonsche Mechanik ist folglich unwissenschaftlicher Unsinn!" Natürlich — und sinnvollerweise — passiert nicht das, was laut naivem Falsifikationismus zu fordern wäre: alle Newtonianer fallen vom Glauben ab, die Physik verfällt in einen Zustand von Lethargie und Verzweiflung, und wir treten in ein neues Zeitalter des Mystizismus ein. Stattdessen wird ein Newtonianer — je nach Jahrhundert und Entwicklungsstand der Hilfshypothesen — argumentieren: "Mit Flüssigkeiten geben wir uns nicht ab!" (Monster-barring) oder eine Strömungsmechanik entwickeln, nach der sich aufgrund der Reibung das Maximum der Strömungsgeschwindigkeit einer rotierenden Flüssigkeit nicht am Rand, sondern bei etwa einem Drittel des Radius vom Rotationszentrum liegt (Monsteradaption). Tatsächlich finden neue Messungen statt — und die Teeblätter sammeln sich gar nicht in der Mitte, sondern bewegen sich auf Kreisbahnen um

das Zentrum herum, und mit etwas Massage gelingt sogar die genaue empirische Vorhersage. Und Newtonianer sprechen nun erhobenen Hauptes von einer glänzenden Bestätigung des Programms. Tatsächlich ist letzteres — eine Anomalie führt zu neuen Vorhersagen und neuen Beobachtungsergebnissen — für Lakatos Ausdruck eines *progressiven* Forschungsprogramms, Monster-barring sowie Monsteranpassung ohne Erweiterung der empirischen Datenbasis sind dagegen Ausdruck eines *degenerativen*. Wissenschaftliche Revolutionen finden vor allem dann statt, wenn neuere Forschungsprogramme nicht nur progressiv sind, sondern auch die empirische Basis älterer Programme mit erklären und übernehmen können (Ptolemäus vs. Kepler, aber auch Newton vs. Einstein). Im Gegensatz zu Popper fordert Lakatos also keine "Sofortrationalität" von Forschungsprogrammen, was dazu führt, das sich der Fortschritt z.T. erst im Nachhinein und mit einigem zeitlichem Verzug erkennen lässt. Dies hat wiederum Feyerabend und viele Wissenschaftspraktiker zu einer Kritik am Lakatosschen kritischen Rationalismus veranlasst.

So ist der zeitgenössische Glaube an "moderne Evalutationstechniken" in der Wissenschaftspraxis *auch* ein Versuch, eine praktikable Antwort auf das Abgrenzungsproblem zu finden. Leider ist festzustellen, das die Evaluationitis in Berufungskommissionen und Forschungsförderungsgremien nach wie vor ungehemmt ist und bisherige spektakuläre Fehlentscheidungen — wie jüngst die Schliessung des fachlich gesunden Fachbereichs Chemie an der Universität Exeter — noch nicht wirklich schmerzen. So bleibt vorerst nur Sarkasmus: Der ungebrochene Glauben einiger Wissenschaftler und Forschungspolitiker an bibliometrische Evaluations-Kriterien erinnert stark an die religiösen Gefühle von Bankern beim Anblick der Aktienkurse kurz vor dem Platzen der Blase. Denn er produziert einen Typus von Wissenschaftler, der Schwalle zweitklassiger Publikationen ausstößt, aber selbst aus Zeitmangel nicht mehr liest; er produziert eine Forschungskultur, die inhärent nicht mehr fortschrittsfähig ist. Wen eine ernsthafte, systematische Kritik an bibliometrischen Kriterien und ihrer Manipulierbarkeit wie [2,32] nicht abschreckt, dem ist nicht zu helfen.

3 Des Pudels Kern und das "Meer der Anomalien"

Man kann im Fall der Informatik ohne Schwierigkeit drei Fundamental-Systeme identifizieren:

1. Klassische Mengentheorie (ZFC),
2. Typtheorie, und
3. Kategorientheorie.

Dies sind Systeme, die zumeist älter als die Informatik selbst sind, trotzdem von der Informatik in Form von Personal und Fragestellungen (vulgo: Forschungsgeldern) profitieren und auch die Ausprägung gewisser Forschungsunterprogramme wie etwa "algebraisch top-down" (siehe Abschnitt 5) bis in den heutigen Tag bestimmen.

Auf dieser Grundlage werden Programmier-, Architektur- und Spezifikationssprachen semantisch begründet, die zum Ziel haben, in technisch unterstützter Form die Praxis der Softwareentwicklung zu ändern — das ist als des Pudels Kern des FM zu betrachten.

Bei der Forderung nach "technischer Unterstützung" einer Sprache/Methode ist zwar auch keine "Sofort-Rationalität" zu erwarten, aber wenn ein Teil-Forschungsprogramm fortdauernd keine Erfolge bei der Entwicklung von programmiersprachlichen Konzepten und entsprechenden Compilern, oder von technisch-unterstützten Entwicklungsmethoden, oder von Beweis-oder Analyseverfahren aufzuweisen hat, dann ist stark zu vermuten, das es sich um ein ziellos-degenerierendes Programm handelt. Denn Theorien zu bilden allein ist bei der Stärke der Fundamental-Systeme keine besondere Schwierigkeit und hat beträchtliche Beliebigkeit. Ob sie dagegen tatsächlich etwas zum Bau von Systemen beitragen können, macht die Substanz ihres Beitrags aus. Dieses Ziel computer-technisch Unterstützung einer Sprache oder Methode ist meiner Auffassung das fachgerechte Experimentum Crucis was die Abgrenzung zu einem "Anything Goes" erlaubt.

Von Anbeginn an waren dabei zwei Lager im FM zu unterscheiden:

1. *Top-down*: Der Ansatzpunkt dieses Teil-Programms sind Spezifikations formalismen, die semantisch dicht an einem der Fundamental-Systeme liegen, und konsequenterweise weit entfernt von Programmiersprachen und insbesondere zustandsorientierten Programmiersprachen. Der Kern gründet auf den semantischen Theorien um Sprachen wie Z, VDM, HOL, ACT-ONE, LOTUS, CSP und neuerdings Event B. Die Entwicklungsmethodik zielt auf formales Refinement oder Transformationelle Entwicklung.
2. *Bottom-up*: Der Ansatzpunkt dieses Teil-Programms sind Programmiersprachen und Programmiercode, der möglichst realistisch erfasst werden soll. Der Kern gründet auf den operational-semantischen Theorien dieser Sprachen. Die Entwicklungsmethodik zielt einerseits auf optimierte Übersetzung und Programmtransformation, andererseits auf Typsysteme und prädikative oder temporale Annotierungssprachen.

Ein typischer Top-downer (wie z.B. der geschätzte Kollege Jean-Raymond Abrial) hält einen Annotierungsansatz der Programmiersprache C für das "Gott-sei-bei-uns", stellvertretend für viele Andere, für die die semantische Behandlung "realer" Programmiersprachen lange Zeit unvorstellbar war und mit dem der Entwicklungsprozeß nicht belastet werden sollte. Für extreme Top-downer ist eine Sprache wie C schlichtweg "dreckig", Vertreter dieses Programms begegnen dem Problem also zur Zeit mit klassischem Monster-barring.

Für einen typischer Bottom-upper (wie z.B. der geschätzte Kollege Peter Müller) [3] ist der Forschungsgegenstand zunächst einmal der Programmierer und seine Programmiersprache, und an den realen Entwicklungsprozessen müsse man

[3] Mit beiden Kollegen hatte ich über viele Jahre regelmässige, anregende Debatten zu diesem Thema im "Formal Methods Club" an der ETH Zürich ...

ansetzen. Extremisten dieser Position halten die Idee für utopisch, das man den Entwicklungsprozeß jemals mit abstrakten Systemmodellen beginnen werden.

Interessanterweise haben *beide* Forschungsprogramme industrielle Anwender — sie sind also beide wachsende Inseln mit einem mehr oder weniger breiten Uferstreifen. Der größte Software-Produzent weltweit hat sich in den letzten Jahren massiv in der Forschung von annotierungsorientierten Spezifikationssprachen (Spec#, VCC) engagiert, wobei abgeschwächte Formen dieses Ansatzes im großen Stil (z.B. SAL) auch bei der Produktion der aktuellen Version des Hauptprodukts verwendet worden sind. Währenddessen engagiert sich der zweitgrößte Software-Konzern insbesondere für modellbasierter Systementwicklung; hier interessieren vor allem Modelle von Geschäftsprozessen, Code dagegen wird in Wesentlichen generiert und stellt oft kein Primärprodukt mehr dar.

Je nach Anwendungsdomaine haben also beide Forschungsprogramme zunächst einmal Erfolge auszuweisen, auf die im Folgenden näher eingegangen wird.

4 Modellierung

FM bietet mathematisch fundierte Beschreibungsformalismen zur Modellierung von Aspekten, wie beispielsweise Systemanforderungen, Architekturen, oder von lokalen Eigenschaften. Durch die Definition dieser Systeme in einem Fundamental-System werden Mehrdeutigkeiten im Verständnis einer Modellierung ausgeschlossen, weil jedes Modell eine im mathematischen Sinne präzise, eindeutige Bedeutung, die so genannte *formale Semantik* hat. Diese Fundierung erlaubt darüberhinaus typischerweise die *Ableitung* von *Kalkülen*, also Mengen von Regeln, mit deren Hilfe sich Eigenschaften einer Modellierung berechnen lassen.

4.1 Gibt es Datenorientierte Spezifikationssprachen?

Da Modellierung an für sich im Software-Entwicklungsprozeß keinen Selbstzweck darstellt, sondern wie gesagt im Hinblick auf technisch unterstützte Analyseverfahren genutzt wird, hat sich die Unterscheidung in daten-orientierte und verhaltensorientierte Spezifikationssprachen eingebürgert, je nachdem, ob die Analysetechnik traditionell eher *deduktionsbasiert* oder *automatenbasiert* war (Vergleiche Abschnitt 7). Klassische Vertreter des datenorientierten Modellierungsstils sind Z, VDM und sein "objekt-orientierter" Nachfahr UML/OCL (an Klassendiagrammen), diverse Hoare Logiken, Dijkstra/Floyds wp-Kalkül als auch diverse algebraische Ansätze wie ACT ONE und CASL. Klassische Vertreter des verhaltensorientierten Ansatzes sind Prozesskalküle wie CSP als auch temporale Logiken wie LTL, CTL, Promela oder JavaPathFinder/JML.

Datenorientierte *Modellierungen* zielen auf die Darstellung einer Transition eines Systemzustands, in der Regel durch Vor- und Nachbedingungen, ab. Demgegenüber werden bei verhaltensorientierten Modellierungen Transitionsrelationen (ggf. mit Zusatzinformationen wie "Events" angereichert) zur Menge aller möglichen *Abläufe eines Systems* (*Traces*) vervollständigt; über Mengen

von Traces können dann Eigenschaften formuliert werden, wie etwa dass alle erreichbaren Zustände des Systems eine gegebene Sicherheitseigenschaft erfüllen. In Verhaltensmodellen können zudem Nebenläufigkeit, Parallelität und Zeitverhalten von Systemmodellen adäquat beschrieben werden.

Die Übertragung der Unterscheidung datenorientiert/verhaltensorientierter Modellierung auf Spezifikations*sprachen* dagegen ist, obwohl weithin üblich, blanker Traditionalismus, wie man schon an der Willkürlichkeit der Einteilung von JML und Promela vs. Hoare-Logiken erkennen kann. Es ist kein besonderes Problem, auch in Z (oder HOL) die Logiken LTL oder CSP zu codieren [69] und dies auch analysetechnisch auszunutzen [11]. Zudem haben Sprachen wie CSP oder Promela mit Typ-Konstruktoren für Listen, Mengen und Tupel außerordentlich ähnliche Sprachausdrucksmächtigkeiten wie z.B. Z. Bei neueren Formalismen wie VCC(1)[58,17], VCC(2)[26,25] oder ACSL[12], die mittels Programmannotierungen Aussagen über das nebenläufige Verhalten von C Programmen machen und hierzu sowohl den SMT-Beweiser Z3 als auch Isabelle/HOL als Beweiser-Backend benutzen [16,17], wird diese Unterscheidung endgültig ad absurdum.

Sinnvoller als eine Einteilung in datenorientiert bzw. verhaltensorientiert auf Sprachen erscheint eine Einteilung in Top-down vs. Bottom-up bzw. *Konzeptionsorientiert* vs. *Ausführungsorientiert*. Erstere Sprachklasse zeichnet sich durch einen geringen konzeptionellen Abstand zu einem Fundamentalsystem aus, während letzere einen geringen Abstand zu einem Ausführungsmodell einer "realen Programmiersprache", also im Extremfall C auf einem konkreten Prozessor anstrebt. In erstere Sprachklasse können somit Z, CSP und UML/OCL eingeordnet werden, in letztere z.B. Spec#/Boogie[6,9], JML/Krakatoa[35], JML/ESC-Java[49], ACSL/Caduceus[35,12] und VCC/Boogie[58,26,17].

4.2 Architekturen und Muster

Musterbasierte Ansätze finden bei der Entwicklung sicherheitskritischer Systeme an mehreren Stellen Verwendung: Beispielsweise haben Sicherheitsanforderungen oftmals sich wiederholende Formen (z.B. „Ereignis x darf erst stattfinden, nachdem Ereignis y stattgefunden hat"). Der Vorteil der Verwendung von Mustern liegt hier (wie auch in der Softwareentwicklung ganz allgemein) in der Wiederverwendung etablierten Wissens. Es sind daher sog. *safety patterns* definiert worden, die helfen, Sicherheitseigenschaften vollständig und präzise zu formulieren. Die so erhaltenen Anforderungen lassen sich sowohl in einer quasi-natürlichen Fachsprache als auch in einer geeigneten Temporallogik oder einer Prozessalgebra ausdrücken.

Weiterhin lassen sich wiederkehrende *Architekturen* sicherheitskritischer Systeme identifizieren. Ein Beispiel für eine solche Referenzarchitektur sind zyklische Systeme, wo in regelmäßigen Zeitabständen die Sensorwerte abgefragt und Aktorkommandos generiert werden. Aber auch interruptgesteuerte Architekturen sind weit verbreitet, nicht zuletzt in Mainstream-Prozessoren mit ihrem harwaretechnisch unterstützten Interruptmechanismen.

Ein weiterer Top-down Trend lässt sich mit der zunehmenden Verbreitung der Unified Modeling Language (UML, der derzeitige de-facto-Standard in der industriellen Software-Modellierung) erkennen, die bei der Modellierung von Geschäftsprozessen, bei der Entwicklung sicherheitskritischer Systeme und bei Datenbanksystemen immer weitere Verbreitung findet. UML stellt eine Vielzahl von Teilsprachen und assoziierten diagrammatischen Notationen bereit, mit der verschiedene Aspekte von Systeme geeignet dargestellt werden können. Hier sind insbesondere Klassenmodelle zur Modellierung von Daten sowie Sequenzdiagramme und Zustandsmaschinen zur Modellierung von Verhalten zu nennen. Auf der Ebene der Spezifikationen wird UML verwendet, um sicherheitskritische Systeme zu spezifizieren und diese Spezifikationen formal auf Sicherheitslücken zu untersuchen. Auf Basis von Expertenwissen in der Entwicklung sicherer Software sind Erweiterungen der UML definiert worden, die das einfache Einfügen von Sicherheitsanforderungen erlauben (zum Beispiel UMLsec [48] und SecureUML [10,20,20]). Diese Sicherheitseigenschaften können dann mit Beweiswerkzeugen (vgl. Abschnitt 7) formal verifiziert werden. Diese Verifikation basiert jeweils auf einer formalen Ausführungs-Semantik für das verwendete Fragment von UML. Insbesondere können auf dieser formalen Grundlage wissenschaftliche Fragen beantwortet werden, die unabhängig vom Kontext der modell-basierten Entwicklung von Interesse sind, wie die Verfeinerung von Spezifikationen sicherheitskritischer Systeme, Sicherheitsaspekte geschichteter Architekturen (wie Protokollschichten), und die (De-)Komponierbarkeit und Modularität sicherheitskritischer Systeme.

Die Forschung nach Architekturen oder Mustern gehört also zu den Kernstücken des Top-down Programms; nach einer Zeit der Euphorie nach dem GoF-Book [37] ist allerdings insgesamt eine gewisse Stagnation zu verzeichnen.

5 Programmiersprachen

Naturgemäß ist die Forschung nach Programmiersprachen, und insbesondere ihrer semantischen Begründung eine der zentralen Interessengebiete im FM. Komplementär zum Abschnitt 4.2, wird die Forschung allerdings primär von Bottom-uppern vorangetrieben.

Unschwer lassen sich Top-down und Bottom-up Strömungen unterscheiden, seitdem es Sprachen gibt: Funktionale und logische Sprachen einerseits und imperative Sprachen und Assemblersprachen mit mehr oder weniger starker Abstraktion gegenüber unterliegenden Maschinenmodellen andererseits. Aus dieser Perspektive nehmen objektorientierte Sprachen in gewisser Weise eine Zwitterstellung ein: einerseits sind sie tiefgreifend zustandsorientiert (Objektreferenzen bilden Objektgraphen mit mutierbaren Objektattributen...), andererseits stellen objektorientierte Zustandsmodelle wie in Java eine beträchtliche Abstraktion gegenüber selbst einfachsten Speichermodellen wie linearem Speicher dar. Der Zwittercharakter wird deutlich, wenn man die Hybridsprachen von der funktionalen Seite wie OCaml, Pizza oder SCALA betrachtet, zu denen die Unterschiede sich zunehmend verwischen.

Die Fundierung von Top-down - als auch Bottom-up mittels formaler, maschinengeprüfter Sprach- und Maschinenmodelle hat in den letzten 10 Jahren große Fortschritte gebracht. Mehrfach wurde die Semantische Begründung der funktionalen Sprachen, nämlich des getypten Lambda-Kalküls mit partieller Rekursion (LCF, Domaintheorie) oder mit total-Rekursiven Funktionen (Theorie der Wohlfundierten Ordnungen und darauf basierender Rekursion) ihrerseits in Logiken höherer Stufe entwickelt, und zwar sowohl in einem klassischem [59] als auch konstruktivistischen Setting [64]. Vor Kurzem gelang sogar die formale semantische Fundierung von komplexen Typsystemen mit Typkonstruktorklassen durch Reflektion auf Funktionen über dem reflexiven Bereich $D \cong A + [D \to D]$[47]; hierdurch können die üblichen Monaden-Kombinatoren in Haskell-artigen Bibliotheken endgültig im Rahmen einer Logik höherer Stufe wie HOL (oder je nach Geschmack: Cartesian Closed Categories (CCC)) interpretiert werden.

Die folgende Übersicht — eine subjektive Auswahl — listet wesentliche Erfolge des Bottom-up Programms des letzten Jahrzehnts:

1. konkrete, "reale" Maschinenmodelle für Prozessoren (AMD,INTEL) und abstrakte Maschinen (JVM) werden routinemäßig entwickelt und im industriellen Maßstab verwendet. Neue Anwendungen wie genaue Laufzeitabschätzungen werden absehbar [74],
2. konservative, d.h. nur auf definitorischen Axiomen basierende Modelle für operationale und axiomatische Semantik für Java-artige Sprachen sind möglich geworden [51,72],
3. JVM Modelle und darauf basierene, korrekt bewiesene Analysemethoden wie [51],
4. axiomatische objektorientierte Speichermodelle wie für Spec# [7],
5. konservative objektorientierte Speichermodelle wie für HOL-OCL# [24],
6. axiomatische lineare Speichermodelle wie für C in VCC(1)# [56,58,17],
7. axiomatische lineare Speichermodelle wie für verteiltes C in VCC(2)# [26,25],
8. separation logics und ähnliche Abstraktionen über linearem Speicher zur Formalisierung von Framing Conditions, [70],
9. etc., etc.,

Die Programm-Verifikationswerkzeuge wie Spec#, VCC oder Simpl[66] basieren darauf, das für gegebene operationale Semantiken ein *wp*-Kalkül abgeleitet werden kann, der vollautomatisch aus einem mit Zusicherungen und Invarianten annotierten Programm eine Formel in Logik erster Stufe generiert. Gilt diese, dann existiert auch ein Beweis in einer Hoare-Logik, der die Gültigkeit der Zusicherungen etabliert (also insbesondere auch Vor- und Nachbedingungen). Da die Formel unabhängig vom Programm ist, kann sie einfach als Beweisziel eines Verifikationsverfahrens (siehe Abschnitt 7) behandelt werden. Schränkt man die Logik geeignet ein (etwa auf Aussagen über Speicherressourcen, wie in SAL-Annotations[27]), können im Hinblick auf spezielle Eigenschaften hochautomatisierte Verfahren zur (partiellen) Programmverifikation konstruiert werden.

Die Liste markiert nicht nur eine beträchtliche Dynamik, die durch die Verfügbarkeit von formalen Modelle von "realen" Prozessoren und maschinen-nahen

Programmiersprachen in interaktiven Theorembeweisumgebungen ausgelöst wurde, sondern auch die entscheidende Bedeutung für das Bottom-up Programm: Sie widerlegt die Hauptkritik, welche die Möglichkeit bestreitet, die Semantiken von "real languages" formal zu handhaben und damit eine wissenschaftlich rationale Fundierung für Werkzeuge und Methoden zu liefern.

Wesentliche andere Forschungslinien der Programmiersprachenforschung sind natürlich Typsysteme und die Möglichkeit, sie für statische Programmanalysen zu nutzen. Für Top-down lässt sich der Trend erkennen, größere Fragmente aus System F herauszuschneiden und für neuere, mächtigere Typsysteme in zumeist funktionalen Sprachen einzusetzen (Ausnahme: SCALA); der Preis sind neuartige, typ-orientierte, vom Benutzer hinzuzufügende Annotierungen (die stark semantischen Charakter haben können), und pragmatisch schlechter kontrollierbare Typinferenz-Systeme. Der Vorteil sind neue Darstellungen von Typ-Konstruktorklassen oder, im Falle der Anwendung dieser Typsysteme in Logiken Höherer Stufe [71], poly-typische Beweistechniken für Programm-Transformationen im Stile der Squiggol-School (or "Algebra of Programming") [14] (d.h. die üblichen "Catamorphismen" und "Hylomorphismen" werden in diesen Logiken tatsächlich First-Class-Citizens).

Übrigens ist dieser späte Erfolg der Squiggol School, zusammen mit der Funktor-Semantik für parametrisierte Datentypen (die immerhin inspirierend für Sprachen wie SML und das Parametrisierungskonzept in Java gewesen ist), einer der wenigen echten Beiträge des Algebraischen Top-down Programms, was sich im Wesentlichen durch die Verwendung von Kategorientheorie als Fundamentalsystems auszeichnet. Es scheint sich zu rächen, das in diesem Teilprogramm niemals ernsthaft versucht worden ist, Beweisumgebungen direkt kategorientheoretisch zu begründen und einen entsprechenden Fundus an formalisierter Mathematik aufzubauen (von wenigen Ausnahmen, etwa im MIZAR Journal, abgesehen). Dadurch haben viele neuere Arbeiten lediglich den Charakter, andere bekannte Logiken oder Rechenmodelle kategorientheoretisch nachzuformulieren — der Vergleich mit dem Neo-Marxismus, der primär damit beschäftigt ist, Erklärungen für '56 und '89 zu finden, liegt da nahe.

6 Formale Entwicklungsmodelle

Im Folgenden behandeln wir die Frage, wie man mit *formalen Techniken* von der abstrakten Spezifikation und seinen formalisierten Anforderungen zu einem ablauffähigen Programm, einer Implementierung kommen kann, das die Anforderung erfüllt. Im Gegensatz zu Abschnitt 7 geht also hier darum, den Entwicklungsprozess selbst formal zu untermauern und technisch zu unterstützen.

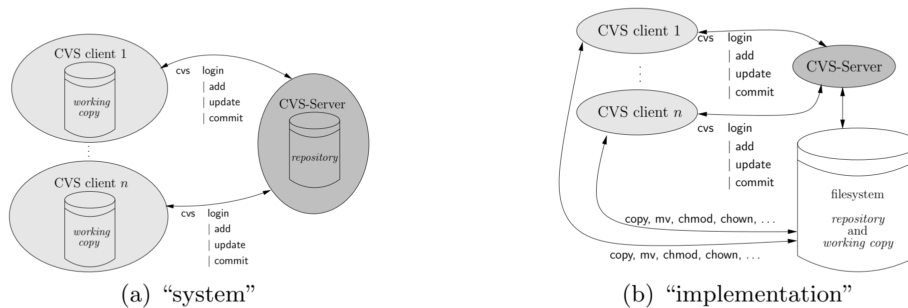
Grundsätzlich kann man drei Ansätze unterscheiden: Programm-Entwicklung durch *Verfeinerung*, durch *Programmtransformation* und durch *Model-Driven Engineering*.

6.1 Verfeinerung

Jede formalen Spezifikation ist ein mathematische Modell und damit notwendigerweise eine Abstraktion der Realität. Spezifikationen können aber auch Abstraktionen von komplexeren Systemmodellen sein: Modelle auf höherem Abstraktionsgraden sind in der Regel erheblich einfacher zu verstehen oder maschinell zu analysieren als detailliertere.

Die Verfeinerungsbeziehung ist eine formale Relation $P \sqsubseteq Q$ zwischen Modellen, die ausdrückt, ob ein Modell Q eine korrekte Implementierung des Modells P ist. Der dieser Beziehung unterliegende Korrektheitsbegriff hängt einerseits vom verwendeten Formalismus und andererseits von der Art der Anforderungen ab, die in der Spezifikation beschrieben werden.

Aus dem Bereich Algebraisch-Top-down sind Ansätze bekannt, die zumeist auf Theoriemorphismen basieren (von klassischem "Forget-Restrict-Identify" [45] über injektive Theoriemorphismen à la KIDS und seinen Nachfolgesystemen), in den letzten Jahren aber kaum weiterentwickelt worden sind. Vorherrschend sind Systeme, die Transitionssysteme über Forward- oder Backwardsimulation (Vergleiche [55] für eine Übersicht über die verschiedensten Simulationsbegriffe) über eine Abstraktionsrelation R zwischen abstrakten und konkreten Zuständen in Beziehung setzen; die Verfeinerungsbeziehung wird also als Tripel $P \sqsubseteq_R Q$ aufgefasst. Solche Transitionssysteme werden üblicherweise durch Vor- und Nachbedingungen einer Operation und dann durch Module oder Packages von Operationen spezifiziert. Ein Verfeinerungssystem kann aus den vom Benutzer gegebenen Transitionssystemen sowie der Abstraktionsrelation Beweisverpflichtungen generieren, die dann verifiziert werden müssen (siehe Abschnitt 7). Systeme, die dieser Methodik unterliegen, sind z.B. das KIV Tool, das KeY Tool, HOL-Z[21,73], sowie B-Tools und sein Nachfolger Rodin[1].



(a) "system" (b) "implementation"

Abbildung 1. Verfeinerungen einer Client-Server Sicherheitsarchitectur

Eine beispielhafte Fallstudie für eine Systementwicklung basierend auf einer Forwardsimulation ist die CVS-Server Studie basierend auf HOL-Z. Wie in Abbildung 1 gezeigt, verfeinert sie ein ein datenorientiertes (Sicherheits-)Modell

einer Datenbank, deren Zugriffskontrolle durch ein *role-based access control model (RBAC)* modelliert wurde, auf ein konkrete Konfiguration eines Dateisystems, das mit UNIX/Posix typischen Lese-und Schreibrechten attibutiert ist. Dabei werden die abstrakten Operationen wie *login*, *add*, *update* und *check-in* berücksichtigt. "korrekt" ist diese Verfeinerung dann, wenn die Implementierung eine Verfeinerung des abstrakten Modells ist, sich also so verhält wie sie sich laut Spezifikation verhalten sollte (vgl. Abschnitt 6.1). Dieses Datenmodell kann in ein Verhaltensmodell engebettet werden: Ein Anwendungsszenario hierfür ergibt sich, wenn man über dieser lokalen Systemtransition (eine Transaktion der Datenbank) die Menge aller Systemläufe konstruiert und hierfür verlangt, dass niemand in einem Systemablauf Daten bekommen kann, für die er sich vorher nicht für hinreichend stark authentifiziert hat (durch *login*). Solche temporallogischen Aussagen lassen sich durch Induktion über die Position in den Läufen zeigen (z.B. [22,11,28]).

Selbstverständlich gibt es auch eine reichhaltige Literatur für verhaltensorientierte Verfeinerungen; insbesondere für Prozessalgebren wie CSP oder CCS sind in vielfältige Verfeinerungsbegriffe definiert und Modelchecker wie FDR implementiert worden. Der Ansatz stagniert aber seit einer Weile; zum Einen macht es sich von der Modellierung her für große Systeme nicht bezahlt, Zustände nur als Prozesse zu betrachten; das resultierende exponentielle Wachstum der Zustandsräume macht allzu viele praktische Probleme "untractable", und Kombinationen mit Zustandsabstraktionsrelationen haben sich nicht durchsetzen können. Auch aus diesem Grunde sind Implementierungen von Protokoll-Analysetools, die an sich in natürlicher Weise durch prozessalgebraische Verfeinerungen ausgedrückt werden können, von prozessalgebraischen Techniken abgerückt und verwenden zumeist problemspezifisches Modelchecking (siehe z.B. der OFMC aus AVISPA).

6.2 Transformation

Kernidee des transformationellen Entwicklungsansatzes ist, Spezifikationen oder Programme syntaktisch so umzuformen, das sich ihre Semantik nicht oder nur verfeinernd verändert. Transformationen sind oft regelbasiert beschrieben, können aber im Grundsatz beliebige Programme sein, die die Übersetzung durchführen.

Frühe Top-down, d.h. spezifikations-orientierte Transformationssysteme waren CIP-S, PROSPECTRA, KIDS, sowie in HOL realisierte Implementierungen von Back und Wrights Refinement Calculus [4,5] oder von KIDS-artiger Standardtransformationen wie GlobalSearch und DivideAndConquer [52]. Diese Forschungsrichtung ist aufgegangen in dem Teilprogramm Model-driven Engineering der im nachfolgenden Abschnitt diskutiert wird.

Bottom-up orientierte Programmtransformationsansätze haben eine aktive Entwicklung erfahren, insbesondere im Bereich der Compilerverifikation. Denn auch wenn man im allgemeinen auf ihre Korrektheit vertraut, so weisen Compiler doch immer wieder Fehler auf, vor allem wenn komplexe Optimierungsstrategien angewendet werden, oder wenn insbesondere sehr hardware-naher Code compiliert wird (bei kombiniertem C und Assemblercode, beispielsweise). Die schon

erwähnte Verfügbarkeit von konkreten, realistischen Maschinenmodellen und immer leistungsfähigeren Beweisumgebungen hat auch hier entscheidende Durchbrüche gebracht. Dabei können Programme nach einem *Divide et Impera*-Prinzip sowohl lokal transformiert als auch lokal verifiziert werden und anschließend ihre Übersetzungen bzw. Beweisteile zusammen gefügt werden. Verifikationen nach dieser Methode finden sich z.B. in [33,65]. Verifikationen, die auch optimierende, strukturverändernde Transformationen behandeln, sind in [15] beschrieben. Zusammenfassungen zur Compiler-Verifikation von sind in [41,40] zu finden.

Eine besondere Problematik von optimierenden Übersetzungstransformationen ergibt sich im Zusammenhang mit modernen, stark nebenläufigen Speichermodellen, z.B. für die JVM, wo lediglich eine partielle Ordnung auf der Ausführung von Load- und Store-operationen durch die Semantik definiert wird. Dadurch ist sowohl die Modellierung der Korrektheit als auch der konkrete Beweis einer optimierenden Transformation für nebenläufige Java-Programme eine Herausforderung [67].

6.3 Model-Driven Engineering

Model-Driven Engineering (MDE) bezeichnet die systematische Verwendung von Modellen als primären Entwicklungsartefakten durch den Entwicklungsprozess hindurch. In seinem sehr liberalen Verständnis bezeichnen "Modelle" hier einfach Beschreibungen in einem maschinengestützten Format (im Grenzfall also: einer abstrakten Syntax), während sich der Terminus "systematisch" auf maschinengestütze Transformationen zwischen Modellen und von Modellen zu Code bezieht. Für die Instanz von MDE, die sich auf UML bezieht und die durch die Object Management Group (OMG) definiert wird, hat sich der Begriff *model-driven architecture* (MDA) etabliert (Vergleiche Abschnitt subsection 4.2). In UML, unterschiedliche *Modellelemente* wie Klassen oder Zustände können zu Klassensystemen oder Zustandsmaschinen kombiniert werden. Modellelemente können mit logischen *Constraints* der Object Constraint Language (OCL) annotiert werden. UML als Sprache überlässt es bewusst eher Werkzeugbauern, verschiedenen Modellelementen oder Modelltransformationsprozessen eine formale Semantik zu unterlegen; für einige Teile der Sprache gibt es *keine* (Use-Cases), für manche Modelle *eine* (Klassenmodelle), für andere Teile *mehrere* formale Semantiken (Sequenzdiagramme, Statemachines); für alle gibt es aber einheitliche abstrakte Syntaxen ("Meta-modelle"), technische Infrastrukturen und Diagramm-Typen um Aspekte der Modelle graphisch darzustellen und zu kommunizieren — Code oder Code-Fragmente sind übrigens auch Modellelemente in diesem Paradigma.

Ein wesentlicher Aspekt dieses Ansatzes ist die Möglichkeit, Modelle zu transformieren — AndroMDA oder ArcStyler sind entsprechende generische Rahmenwerke, in denen Modelle gespeichert und die Implementierung von Modelltransformationen auf ihnen unterstützt werden. Industriell relevante Anwendungen solcher Rahmenwerke erzeugen z.B. verteilte Applikationen für Corba- und EJB Frameworks, dynamischen Testcode (durch entsprechende Wrapper um Code-Fragmente herum), oder Zugriffs-"Watchdogs" um Zugriffskontrolle in einem verteilten Softwaresystem durchzusetzen.

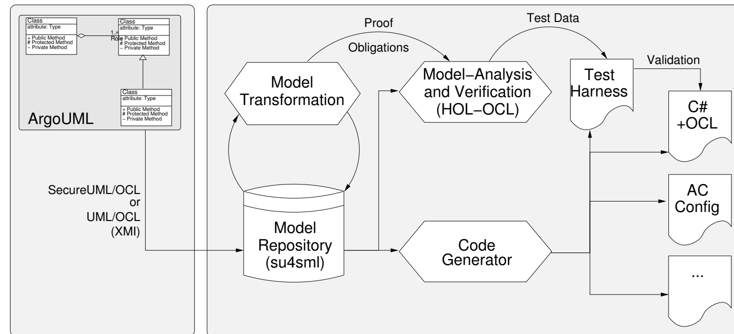


Abbildung 2. Das HOL-OCL MDA Framework

In Abbildung 2 wird das HOL-OCL framework [20] dargestellt: ein UML Frontend wie ArgoUML wird mit einem Kernsystem verbunden, das auf einer Einbettung von UML/OCL in Isabelle/HOL basiert, ein in SML implementiertes Repositorium für UML Modelle vorsieht (samt Parsern, Typecheckern und import-export-Funktionen), sowie einem generischen Code-Generator. Ein Spezifikum dieses Rahmenwerks ist es, das Beweisverpflichtungen ebenfalls Modellelemente sind und während einer Modell-oder Codetransformation erzeugt werden können; diese können dann im Isabelle/HOL-OCL Kern mit interaktiven und automatischen Methoden bewiesen werden. Dadurch können Forward-Simulation Verfeinerungen implementiert werden genauso wie Transformationen, die ein kombiniertes System- und Sicherheitsmodell à la SecureUML in ein angereichertes Standard-Systemmodell umwandeln und dabei Seitenbedingungen generieren, die die Lifeness des generierten gesicherten Systems etablieren [20].

7 Verifikationstechnik

Es ist zwar grundsätzlich möglich, aus Verfeinerungen oder aus Programmannotierungen generierte Beweisverpflichtungen "von Hand" (etwa durch Inspektionen) zu beweisen. Allerdings stößt die Lösung dieser Aufgabe ohne computer-gestützte Techniken an Grenzen der Praktikabilität und Zuverlässigkeit: Erfahrungen mit VCC(1) und seinem sehr feinkörnigen Speichermodell zeigen, das generierte Beweisverpflichtungen ohne weiteres ein Megabyte groß werden können (für eine C Funktion mit ca. 100 Zeilen länge, gemessen im SMT Austauschformat das zwischen Boogie, Z3 sowie HOL-Boogie[16,17]). Es ist somit fair zu sagen, daß Verfahren zur Lösung von mathematischen Formeln das Herz des FM darstellen, und zwar gleichermaßen für Top-down wie für Bottom-up. Es ist dabei bemerkenswert, das klassische Unterscheidungen wie Safety und Security aus Verifikationssicht kaum von Bedeutung sind[4].

[4] ... und übrigens auch auf Modellierungsebene immer weniger Sinn machen: Egal ob durch einen Protokoll-Fehler oder durch einen Exploit eines Buffer-Overflows

Grundsätzlich können drei Verfahren der Verifikation von Systemeigenschaften unterschieden werden: *deduktionsbasiertes* Theorembeweisen, *automatenbasiertes* Modelchecking und modell-basiertes Testen — letzteres verwendet, wie wir noch sehen, nicht nur dieselben Basistechniken wie Theorembeweiser oder Modelchecker und ähnliche Meta-Annahmen wie Modelchecking, sondern kann eine "approximierende Verifikation" darstellen.

7.1 Theorembeweisen

Deduktionsorientiertes Theorembeweisen benutzt Kalküle, um Beweisziele (also z.B. Beweisverpflichtungen aus einer Verfeinerung oder einer Programmverifikation) zu zerlegen. Konzeptionell sind Beweisziele Formeln in Logiken wie z.B. Aussagenlogik (PL), Logik erster Stufe (FOL), oder Logik höherer Stufe (HOL), die zumeist mit Hilfstheorien erweitert werden müssen (Arithmetik, Bitvektoren, aber auch Speichermodellen mit Theorien über Load-Store-Operationen, etc.).

Grundsätzlich kann man interaktive Beweissysteme wie Isabelle/HOL und automatische wie Vampire oder Z3[29,30] unterscheiden; bei ersteren wird die Liste der Beweisziele und ihrer Zerlegungen zu einem *Beweiszustand* zusammengefaßt, der mittels *Taktiken* immer weiter verfeinert wird zu Endzuständen, aus denen sich die logische Wahrheit syntaktisch ergibt. Letztere verwandeln die Formeln in clevere Datenstrukturen (Tableaux, SAT), in denen hocheffizient implementierte Heuristiken die Beweissuche realisieren. Erstere sind einfach zu erweitern und haben einen beträchtlichen Fundus an formalisierter Mathematik in ihren Bibliotheken, verfügten jedoch vor einiger Zeit noch über einen relativ niedrigen Automatisierungsgrad. Letztere lösen Formeln von z.T. staunenerregender Komplexität in kürzester Zeit — Beweisfehlversuche sind allerdings notorisch schwer zu interpretieren und zu analysieren, und bis vor einiger Zeit war die Erweiterbarkeit dieser Systeme ausserordentlich problematisch.

Die Situation hat sich in beiden Lagern in den letzten Jahren entscheidend verbessert: durch die Möglichkeit, Taktiken programmgesteuert anzuwenden, sind mächtige Beweisprozeduren (Termersetzungsprozeduren, Tableaux-Beweiser, Arithmetische Entscheidungsprozeduren) in interaktiven Beweisern entwickelt worden, die zudem konstruktionsbedingt eine außerordentliche hohe Zuverlässigkeit und Vertrauenswürdigkeit erreicht haben (zumindest bei Beweisern, die eine "LCF-style Architecture" aufweisen wie Isabelle/HOL oder HOL-Light; eine Diskussion findet sich in [43]). Bei den Vollautomaten wie Z3 oder Alt-Ergo hat sich vorläufig die DPLL(X) - Architektur [38] durchgesetzt, eine generische Erweiterung der SAT-Beweiser mit eingebauten Algorithmen für Kongruenzabschlüsse, die mit Entscheidungsprozeduren für Arithmetik, Bitvektoren, Quantoreninstantiierungen etc. kombiniert werden kann. Durch geeignete Codierungstechniken von Hilfsattributen ist zumindest für den Fall der Programmanalyse das Problem der Rückinterpretation von Beweisfehlversuchen erleichtert worden[54,16].

---

ein Virus Millionen Rechner und damit IT-Infrastukturen eines Landes zum Ausfall bringt: es besteht in jedem Fall Gefahr für Leib und Leben von Menschen.

Zudem wird intensiv an Kombinationen von automatischen und interaktiven Beweistechniken gearbeitet, sei es auf Beweiser-Ebene wie bei zchaff oder Vampire in Isabelle oder auf der Methoden-Ebene wie bei VCC(1)/HOL-Boogie[17].

Interaktive Beweiser haben — wie schon erwähnt — erheblichen Anteil an der Beherrschung der Grundlagen von "real machine models", also Prozessoren, Virtuelle Maschinen oder Ausführungsmodelle von Sprachen wie C gehabt[28]; diese waren für die Durchbrüche des Bottom-up Programmes von wesentlicher Bedeutung. (Daneben gibt es beachtliche Erfolge wie die Formalisierung großer mathematischer Beweise wie Vierfarbentheorem, Primzahl-Verteilungstheorem oder der Kepler-Vermutung). Automatische Beweiser, deren Effektivität um Grössenordnungen zugenommen hat, sind für den Bau von Programmanalysesystemen wie Krakatoa oder Spec# sowieso kriegsentscheidend.

Interaktives oder automatisches Theorembeweisen wird in der industriellen Praxis zur Zeit vor allem bei hoch-kritischen Anwendungen im militärischen Bereich, bei Java-Smartcards, bei Mikroprozessoren oder bei maschinennaher Software wie Treiber und Betriebssysteme angewandt (z.B. [13]; an dieser Stelle sei auch auf den bemerkenswerten Survey von Klein verwiesen [50]). Die Firma Intel hat beispielsweise mehr als 40 % des Pentium IV bis auf das Gatterlevel hinunter so verifiziert. Prozessoren sind also vermutlich die am weitesten verbreiteten handelsübliche Produkte, für die Verifikation in der Herstellung bedeutsam ist. Für die schwierigen Probleme werden hier Theorembeweiser eingesetzt, weil die im folgenden beschriebenen Modelchecker sie aus Aufwandsgründen praktisch nicht mehr bewältigen können.

7.2 Modelchecking

Modelchecking (auch: Modellprüfverfahren) versucht zumeist, für Widerlegungsvollständige Logiken (PL, FOL, Monadische Logik 2. Stufe,LTL, CTL,...) für das negierte Beweisziel ein Modell zu finden; wenn das gelingt, ist das Beweisziel gezeigt. Durch geeignete Datenstrukturen können Modelle vollautomatisch und z.T. hocheffizient gesucht werden. Das Konzept ist auch auf verhaltensorientierte Modelle in LTL und CTL, ja sogar auf Prozessalgebren anwendbar (die nichts mit Widerlegungsvollständigkeit zu tun haben), und läuft dort zumeist auf die Exploration von kompakt repräsentierten Automaten oder "Labelled Transition Systems" heraus. Für die kompakte Repräsentation dienen zumeist *binary decision diagrams* (BDD), die boolesche Funktionen eindeutig (*kanonisch*) repräsentieren; sie setzen eine Variablenordnung voraus, aufgrund der die Boolesche Funktion mehr oder weniger kompakt repräsentiert werden können.

In etwas mehr Detail läßt sich das Verfahren so beschreiben; Ist das zu modellierende System endlich, kann die Systemtransitionsrelation als aussagenlogische Formel ϕ dargestellt werden kann — sämtliche Daten müssen nur binär codiert werden. Stellt man die Frage, ob ϕ eine Anforderung ψ erfüllt, bedeutet dies, dass man das Beweisziel $\phi \rightarrow \psi$ negieren und dafür ein Modell suchen muss; gibt es keins, gilt das Beweisziel. Das Verfahren ist für Aussagenlogik vollständig, allerdings oft unpraktikabel ("Zustandsexplosion"). BDD's erlauben für viele

relevante Boolesche Funktionen (z.B. die Addition auf binär codierten Zahlen) sehr kompakte Darstellungen, die die Zustandsexplosion umgehen.

Verallgemeinert man diese Techniken zur Behandlung von verhaltensorientieren Systemeigenschaften, so kann z.B. die transitive Hülle der Systemtransitionsrelation bilden (für Aussagen der Form: “alle erreichbaren Zustände erfüllen ψ”) oder aus ihr einen endlichen Automaten erzeugen. In letzterem kann man auch Aussagen der Form entscheiden: “irgendwann werden auf allen Pfaden Zustände erreicht, auf denen ψ gilt”. Mit endlichen Automaten können auch nebenläufige Systeme analysiert werden; hierbei werden Automaten konstruiert, die die Menge aller möglichen Verzahnungen lokaler Teilsystemtransitionsrelationen enthalten.

Besonders erfolgreich kann Modelchecking für die Programmanalyse nebenläufiger Programme mit trivialem Datenanteil herangezogen werden (SPIN, Java/Pathfinder, ...), oder in der Protokollanalyse herangezogen werden, da man es hier mit nebenläufigen Protokollabläufen zu tun hat, deren Interaktionen schwer zu überblicken sind. Es stehen mittlerweile Spezialwerkzeuge zur semi- und vollautomatischen Analyse kryptographischer Protokolle zur Verfügung [3], die erfolgreich zum Aufdecken von Fehlern bzw. Nachweisen der Sicherheit veröffentlichter Protokolle eingesetzt worden sind und zudem den Entwurf und die Standardisierung von Protokollen unterstützen [57].

Während die Erfolge von Modelcheckern in sehr spezialisierten Anwendungsdomänen unbestreitbar sind, muss ihre Verwendbarkeit als allgemeine Modellierungs- und Analysetechnik skeptisch beurteilt werden. Es gibt dafür auch theoretische Gründe: Für BDD’s ist bekannt, das keine Variablenordnung existiert, die gleichermaßen für Multiplikation und Addition kompaktifiziert; bekannte Challenge-Probleme wie $(x \texttt{ div } y) * y + (x \texttt{ mod } y) = x$ können mit aktuellen High-End Modelcheckern auf BDD-basis lediglich für maximal zwölfbittige Zahldarstellungen für x und y bewiesen werden (siehe [63], aber auch neuere Ergebnisse von Systemen wie Boolector) Pragmatisch sind die Constraints für die Modellierung realer Probleme derart einschneidend, das auch die Arbeitseffektivität der Verifikation gering ist, trotz eines automatischen Verifikationsverfahrens (siehe auch [11]). Erfolgreiche und etablierte Verifikationsabteilungen in der Industrie (Intel,AMD,Microsoft) setzen daher auf eine geschickte Kombination von Testen, Modelchecking *und* interaktivem Theorembeweisen.

7.3 Testen

Wie beim Modelchecking muss beim systematischen, modell-basierten Testen der Zustandsraum eines realen Systems sowie die Menge der *möglichen* Eingabedaten endlich beschränkt werden; während man pragmatisch bei modelcheckingbasierter Verifikation den Bereich der Quantoren einer Systemeigenschaft beschränkt, sucht man beim systematischen Testen andere endliche Teilbereiche von Eingabedaten, die sich nach *Testadäquatheitskriterien* richten. Solche Kriterien, die eine Aussage darüber machen wann eine Eigenschaft eines Systems, also ein Beweisziel, *genug* getestet worden ist, lassen sich teilweise durch Fehlermodelle von Schaltwerken, Programmen oder Spezifikationen begründen. Solche Adäquatheitskriterien können spezifikationsbasiert sein (z.B. Partition-Coverage;

ein Test für jedes Glied der DNF der Spezifikation) oder programmbasiert sein (z.B. Path-Coverage; ein Test für jeden Pfad durch den Kontrollflussgraphen eines Programms) und sich auf das Systemverhalten beziehen (z.B. Transition-Coverage der Spezifikation; alle Transitionen der Spezifikation in Form eines LTS müssen in einer Testsequenz vorkommen).

Vollständige, z.B. mit einem Theorembeweiser durchgeführte Verifikation ist so gesehen kein Gegensatz zum Testen. Systematische, modellbasierte Tests verwenden nicht nur dieselben Basistechniken wie Theorembeweiser oder Modelchecker, sondern können eine "approximierende Verifikation" darstellen [23]; in dem dort beschriebenen System HOL-TestGen wird eine Testspezifikation TS automatisch in ein Testtheorem der Form:

$$Case_1 \Longrightarrow \ldots \Longrightarrow Case_n \Longrightarrow THYP(H_1) \Longrightarrow \ldots \Longrightarrow THYP(H_m) \Longrightarrow TS$$

äquivalent zerlegt, wobei die $Case_i$ (die *Testfälle*) der Form

$$Constraint_1\ x \Longrightarrow \ldots Constraint_k\ x \Longrightarrow post\ x\ (sut\ x)$$

sind. Constraint-Solving-Techniken erlauben nun die Konstruktion von Zeugen für x, die die Constraints erfüllen. Aus $post\ x\ (sut\ x)$ generiert man dann ein Testorakel, das das System-Unter-Test *sut* mit diesem Zeugen aufruft und die Zulässigkeit seines Outputs mittels des Codes für *post* prüft. Der Aspekt, das gegebener Code gegenüber einem Modell abgeprüft wird und das man mit dem Modell solange experimentieren kann, bis man durch das Modell den Code "erfaßt" hat, ist ein Spezifikum des Testansatzes und macht Modell-basiertes Testen für das Reverse-Engineering unverzichtbar, auch im industriellen Maßstab [42].

HOL-TestGen ist für sehr ambitionierte Fallstudien angewendet worden; diese nutzen auch die Möglichkeit, Scenarien für Sequenztest und reaktivem Sequenztest in diesem Rahmenwerk darzustellen [19].

Die $THYP(H_i)$ im obigen Testtheorem sind übrigens die explizite Testhypothesen, d.h. Formeln, die die genaue "logische Differenz" zwischen Test und Verifikation darstellen. Das obige Testtheorem *bedeutet* demnach, daß wenn *sut* den Test für alle Testfälle besteht und wenn die Testhypothesen gelten, dann erfüllt *sut genau* die Testspezifikation. Klassische Testhypothesen [39] sind z.B. Uniformität (wenn der Test für einen Testdatum der Klasse gelingt, gelingt er für alle der Klasse) oder Regularität (es werden alle Daten bis zu einer gewissen Komplexitätsstufe getestet).

Erschöpfendes Testen aller möglichen Eingabesequenzen ist selbst im Hinblick auf die üblichen Adäquatheitskriterien wegen der exponentiell wachsenden Anzahl der Testabläufe in der Praxis oft nicht durchführbar. Wünschenswert sind somit Adäquatheitskriterien, die bei einem gegebenen Aufwand zu einer Menge von Testabläufen führen, die mit hoher Wahrscheinlichkeit sicherheitskritische Fehler aufdecken, und somit ein größtmögliches Vertrauen in die Sicherheit der Implementierung liefern. Dies führt zur Idee der probabilistischen Adäquatheitskriterien, (z.B. probabilistische Uniformität[31]), bei der für jede Testklasse verlangt wird, daß sie mit der gleichen Wahrscheinlichkeit wie alle anderen getestet werden.

In jüngerer Zeit ist ein Trend zur Fusion von Test-Generierungs-Techniken mit Model-Checking einerseits und automatischem [8,75] und interaktivem Theorembeweisen [18] zu beobachten.

8 Schlussfolgerungen

Man kann einen Trend zu einem dichteren Netz von Anwendungsszenarien für FM beobachten, das immer weitere Bereiche der Softwaretechnik durchzieht. Dies gilt für Bottom-up, aber z. Zt. in scheinbar stärkerem Maße für Top-down. Es fragt sich allerdings, ob sich die starke Dynamik, die sich insbesondere auf Durchbrüche bei den vollautomatischen Beweissystemen wie Z3 stützt, nicht abschwächt und abstraktere Modellierungstechniken bald wieder stärker im Vordergrund stehen.

Neben dem prinzipiellen Nutzen von FM, der in einer weitergehenden wissenschaftlichen Fundierung der Softwaretechnik als solcher besteht—sei es in der Bedarfsanalyse, dem Entwurf, der Implementierung oder der Verifikation—wird auch ein immer größerer praktischer Nutzen erkennbar. Drei praktisch relevante Anwendungsszenarien lassen sich identifizieren:

- **FM "under-the-hood".** In immer mehr Entwicklungswerkzeugen und Compilern werden intern Formale Methoden Techniken wie Abstrakte Interpretation, Modelchecking, Extended Typechecking und Theorembeweisen verwendet, ohne dass der Programmentwickler das direkt sieht (Java Bytecode-Verifier, PolySpace, ...).
- **FM "light-weighed".** Dies bezeichnet eine Reihe von Migrationsversuchen von Formalen Methoden wie Runtime-Testing von Annotierungssprachen wie OCL, JML oder Spec#, oder wie hochautomatische Verfahren basierend auf eingeschränkten Annotierungen à la SAL oder modelcheck-fähige temporale Constraints aus Bibliotheken. FM "light-weighed" umfasst auch Techniken wie MDA (auch spezialisiert für gewisse Zwecke wie die systematische Einführung von Sicherheitsmechanismen) oder Reverse-Engineering durch Modell-basiertes Testen.
- **FM "hard-core".** Damit werden theorembeweiser-basierten Ansätze wie Verfeinerung und Codeverifikation zusammengefasst, die einen speziell ausgebildeten Verifikationsingenieur erfordern und aus diesem Grund in der industriellen Praxis zur Zeit auf hardware-nahe, sicherheitskritische oder höchste Zertifizierungsstufen erfordernde Systeme beschränkt ist.

Es sei hierbei explizit festgestellt, dass alle drei Bereiche sich gegenseitig bedingen; Techniken, die in FM "hard-core" entwickelt werden, finden mitunter ihren Weg in FM "under-the-hood"-Szenarien. Umgekehrt ist ein höherer Kenntnisstand und die Verbreitung von FM "light-weighed"-Techniken in der industriellen Praxis für Projekte, die FM "hard-core" etwa aufgrund einer angestrebten Zertifizierung anstreben, von Vorteil: Von Softwareentwicklern erstellte Modelle etwa in UML/OCL können eine wertvolle Grundlage für ein verfeinertes und von Verifikationsingenieuren erstelltes Systemmodell sein.

Danksagung: Ganz herzlich möchte ich mich bei Bernd Krieg-Brückner bedanken, der mich mit FM vertraut gemacht hat, mir viel Raum gab, mich zu entwickeln und mich ein Stück meines Weges begleitet hat.

Literatur

1. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2009. to appear.
2. Nancy J. Adler and Anne-Wil Harzing. When Knowledge Wins: Transcending the Sense and Nonsense of Academic Rankings. *Learning & Education*, 8(1), 2009.
3. A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Vigano, and L. Vigneron. The AVISS Security Protocol Analysis Tool. In *CAV 2002*, LNCS 2404, pages 349–353, 2002.
4. R.-J. Back and J. von Wright. *Refinement Calculus*. 1998.
5. Ralph-Johan Back, Jim Grundy, and Joakim von Wright. Structured calculational proof. *Formal Asp. Comput.*, 9(5-6):469–483, 1997.
6. M. Barnett, K.R.M Leino, and W. Schulte. The Boogie Tool home page. `http://research.microsoft.com/en-us/projects/boogie/`.
7. Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
8. Michael Barnett, Manuel Fähndrich, Peli de Halleux, Francesco Logozzo, and Nikolai Tillmann. Exploiting the synergy between automated-test-generation and programming-by-contract. In *ICSE Companion*, pages 401–402. IEEE, 2009.
9. Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Francesco Logozzo, Peter Müller, Wolfram Schulte, Herman Venter, and Songtao Xia. Spec#. Microsoft Research, Redmond, 2008. `http://research.microsoft.com/specsharp`.
10. D. Basin, J. Doser, and T. Lodderstedt. Model driven security: from UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 2005.
11. David Basin, Hironobu Kuruma, Kunihiko Miyazaki, Kazuo Takaragi, and Burkhart Wolff. Verifying a signature architecture: a comparative case study. *Formal Aspects of Computing*, 19(1):63–91, March 2007. `http://www.springerlink.com/content/u368650p18557674/?p=8851693f5ba14a3fb9d493dae37783b8&pi=0`.
12. Patrick Baudin, Jean-Christophe Filliatre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL: ANSI/ISO C Specification Language – version 1.3. Technical report, INRIA Saclay, 22. Oct, 2008.
13. S. Beyer, C. Jacobi, D. Krˆning, D. Leinenbach, and W.J. Paul. Putting it all together - formal verification of the vamp. 2005.
14. Bird, Moore, Backhouse, Gibbons, et al. Algebra of programming research group. Home Page. `http://www.comlab.ox.ac.uk/research/pdt/ap/pubs.html`.
15. J. Blech, S. Glesner, J. Leitner, and S. M¸lling. Optimizing Code Generation from SSA Form: A Comparison Between Two Formal Correctness Proofs in Isabelle/HOL. In *ETAPS 2005*.
16. Sascha Böhme, Rustan Leino, and Burkhart Wolff. HOL-Boogie — an interactive prover for the boogie program verifier. In Sofiene Tahar, Otmane Ait Mohamed, and César Muñoz, editors, *21th International Conference on Theorem proving in Higher-Order Logics (TPHOLs 2008)*, LNCS 5170. Springer-Verlag, Montreal, Canada, 2008.

17. Sascha Böhme, Michał Moskal, Wolfram Schulte, and Burkhart Wolff. HOL-Boogie — an interactive prover-backend for the Verified C Compiler. *Journal of Automated Resoning (JAR)*, 44(1–2):111–144, 2009.
18. Achim D. Brucker, Lukas Brügger, Paul Kearney, and Burkhart Wolff. Verified firewall policy transformations for test case generation. In Ana Cavalli and Sudipto Ghosh, editors, *International Conference on Software Testing (ICST10)*, Lecture Notes in Computer Science. Springer-Verlag, 2010.
19. Achim D. Brucker, Lukas Brügger, and Burkhart Wolff. Model-based firewall conformance testing. In Kenji Suzuki and Teruo Higashino, editors, *Testcom/FATES 2008*, LNCS 5047, pages 103–118. Springer-Verlag, Tokyo, Japan, 2008.
20. Achim D. Brucker, Jürgen Doser, and Burkhart Wolff. An MDA framework supporting OCL. *Electronic Communications of the EASST*, 5, 2007.
21. Achim D. Brucker, Frank Rittinger, and Burkhart Wolff. HOL-Z 2.0: A proof environment for z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, February 2003.
22. Achim D. Brucker and Burkhart Wolff. A verification approach for applied system security. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):233–247, 2005.
23. Achim D. Brucker and Burkhart Wolff. Test-sequence generation with hol-testgen – with an application to firewall testing. In Bertrand Meyer and Yuri Gurevich, editors, *TAP 2007: Tests And Proofs*, number 4454 in Lecture Notes in Computer Science, pages 149–168. Springer-Verlag, Zurich, 2007.
24. Achim D. Brucker and Burkhart Wolff. An extensible encoding of object-oriented data models in hol with an application to IMP++. *Journal of Automated Reasoning (JAR)*, Selected Papers of the AVOCS-VERIFY Workshop 2006(3–4):219–249, 2008. Serge Autexier, Heiko Mantel, Stephan Merz, and Tobias Nipkow (eds).
25. Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009.
26. Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. A Precise Yet Efficient Memory Model For C. In *Proceedings of the European Conference of Object-Oriented Programing-Languages (ECOOP '09)*, Lecture Notes in Computer Science. Springer-Verlag, July 2009. submitted paper.
27. Manuvir Das. Formal specifications on industrial-strength code-from myth to reality. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, page 1. Springer, 2006.
28. Matthias Daum, Jan Dörrenbächer, and Burkhart Wolff. Proving fairness and implementation correctness of a microkernel scheduler. *Journal of Automated Reasoning (JAR)*, 42(2–4):349–388, 2009. G. Klein, R. Huuck and B. Schlich: Special Issue on Operating System Verification (2008).
29. Leonardo de Moura and Nikolaj Bjørner. Efficient incremental E-matching for SMT solvers. In *21st Conference on Automated Deduction (CADE 2007)*, LNCS 4603, 2007.
30. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, March–April 2008.
31. A. Denise, M.-C. Gaudel, S.-D. Gouraud, R. Lassaigne, and S. Peyronnet. Uniform random sampling of traces in very large models. In *1st International ACM Workshop on Random Testing*, pages 10 –19, July 2006.

32. Commission d'Evaluation 2007. Une évaluation des indicateurs bibliométriques. Technical report, INRIA, 2007. `http://www.inria.fr/inria/organigramme/documents/ce_indicateurs.pdf`.
33. A. Dold, F. W. von Henke, and W. Goerigk. A Completely Verified Realistic Bootstrap Compiler. *International Journal of Foundations of Computer Science*, 14(4):659–680, 2003.
34. Paul Feyerabend. *Wider den Methodenzwang*. Suhrkamp (stw 597), Frankfurt am Main, 1975. ISBN 3-518-28197-6.
35. Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.
36. Joint Taskforce for Computing Curricula. Computing curricula 2005: The overview report. Technical report, ACM/AIS/ IEEE, 2005, 2005. `www.acm.org/education/curric_vols/CC2005-March06Final.pdf`.
37. Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley International, 1994.
38. Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Dpll(t): Fast decision procedures. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.
39. Marie-Claude Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT 95*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, Aarhus, Denmark, 1995.
40. S. Glesner and J. Blech. Logische und softwaretechnische herausforderungen bei der verifikation optimierender compiler. In *SE 2005*. LNI, März 2005.
41. S. Glesner, G. Goos, and W. Zimmermann. Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers). *it - Information Technology*, 46:265–276, 2004. Print ISSN: 1611-2776.
42. Wolfgang Grieskamp, Nicolas Kicillof, Dave MacDonald, Alok Nandan, Keith Stobie, and Fred L. Wurden. Model-based quality assurance of windows protocol documentation. In *ICST*, pages 502–506, 2008.
43. Thomas C. Hales. Formal proof. *Notices of the American Mathematical Society*, 55(11):1370–1382, 2008.
44. John Harrison. Formal proof-theory and practice. *Notices of the American Mathematical Society*, 55(11):1305–1407, 2008.
45. Rolf Hennicker. Context induction: A proof principle for behavioural abstractions and algebraic implementations. *Formal Asp. Comput.*, 3(4):326–345, 1991.
46. Andrew Hodges. *Alan Turing: the Enigma*. London: Burnett; New York: Simon and Schuster, 1983.
47. Brian Huffman, John Matthews, and Peter White. Axiomatic constructor classes in isabelle/holcf. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2005.
48. J. J¸rjens. *Secure Systems Development with UML*. 2004. ISBN: 3-540-00701-6.
49. Joseph Kiniry. ESC/Java2 Home Page, 2005. `http://kind.ucd.ie/products/opensource/ESCJava2`.
50. Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, February 2009.
51. Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theor. Comput. Sci.*, 3(298):583–626, 2003.

52. Kolyang, T. Santen, and B. Wolff. Correct and user-friendly implementation of transformation systems. In M.-C. Gaudel and J. Woodcock, editors, *FME 96 — Industrial Benefits and Advances in Formal Methods*, LNCS 1051, pages 629–648. Springer Verlag, 1996.
53. Ralf Küsters. ETH Zürich, personal communication, 2006.
54. K. Rustan M. Leino, Todd D. Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, 55(1-3):209–226, 2005.
55. Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.
56. Stefan Maus, Michal Moskal, and Wolfram Schulte. Vx86: x86 Assembler Simulated in C Powered by Automated Theorem Proving. In *AMAST*, pages 284–298, 2008.
57. C. Meadows, P. Syverson, and I. Cervesato. Formalizing GDOI group key management requirements in NPATRL. In *Proceedings of the 8th ACM Computer and Communications Security Conference (CCS 2001)*, pages 235–244. ACM Press, 2001.
58. Michal Moskał, Wolfram Schulte, and Herman Venter. Bits, words, and types: Memory models for a verifying c compiler. *Science of Computer Programming (SCP)*, 2008. to appear.
59. Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. Holcf=hol+lcf. *J. Funct. Program.*, 9(2):191–223, 1999.
60. Special issue on formal proof, 2008. `http://www.ams.org/notices/200811/`.
61. Proof by computer: Harnessing the power of computers to verify mathematical proofs. `http://www.physorg.com/news145200777.html`.
62. Karl Popper. *Die Logik der Forschung*. Verlags Mohr Siebeck, 1934.
63. Nicole Rauch and Burkhart Wolff. Formalizing java's two's-complement integral type in isabelle/hol. In *Electronic Notes in Theoretical Computer Science*, volume 80. Elsevier Science Publishers, 2003.
64. Bernhard Reus and Thomas Streicher. General synthetic domain theory - a logical approach. *Mathematical Structures in Computer Science*, 9(2):177–223, 1999.
65. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.
66. Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/*HOL. PhD thesis, Technische Universität München, 2006.
67. Jaroslav Sevcík and David Aspinall. On validity of program transformations in the Java memory model. In Jan Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 27–51. Springer, 2008.
68. Imre Lakatos (Übersetzung Arpád Szabó). *Die Methodologie der wissenschaftlichen Forschungsprogramme*. Vieweg Braunschweig; Wiesbaden, 1982. ISBN 3-528-08429-4.
69. H. Tej and B. Wolff. A corrected failure-divergence model for csp in isabelle/hol. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Proceedings of the FME 97 — Industrial Applications and Strengthened Foundations of Formal Methods*, LNCS 1313, pages 318–337. Springer Verlag, 1997.
70. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 97–108. ACM, 2007.
71. Norbert Völker. HOL2P - a system of classical higher order logic with second order polymorphism. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*, pages 334–351. Springer, 2007.

72. David von Oheimb and Tobias Nipkow. Machine-Checking the Java Specification: Proving Type-Safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 119–156. Springer, 1999.
73. Makarius Wenzel and Burkhart Wolff. Building formal method tools in the Isabelle/ISAR framework. In Klaus Schneider and Jens Brandt, editors, *TPHOLs 2007*, LNCS 4732, pages 351–366. Springer-Verlag, 2007.
74. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.
75. G. Wimmel, H. Lötzbeyer, A. Pretschner, and O. Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *Journal on Software Testing Verification and Reliability*, 10(4):229–248, 2000.