Jahrbuch der Heinrich-Heine-Universität Düsseldorf

2005/2006

HEINRICH HEINE
UNIVERSITÄT
DÜSSELDORF

Jahrbuch der Heinrich-Heine-Universität Düsseldorf

2005/2006

Jahrbuch der Heinrich-Heine-Universität Düsseldorf

2005/2006

Herausgegeben vom Rektor
der Heinrich-Heine-Universität Düsseldorf
Univ.-Prof. Dr. Dr. Alfons Labisch

Konzeption und Redaktion:
em. Univ.-Prof. Dr. Hans Süssmuth

Einbandgestaltung: Wiedemeier & Martin, Düsseldorf
Titelbild: Schloss Mickeln, Tagungszentrum der Universität
Redaktionsassistenz: Georg Stüttgen
Beratung: Friedrich-K. Unterweg
Satz: Friedhelm Sowa, LaTeX
Herstellung: WAZ-Druck GmbH & Co. KG, Duisburg
Gesetzt aus der Adobe Times
ISBN 3-9808514-4-3

Inhalt

Sonderforschungsbereiche der Heinrich-Heine-Universität Düsseldorf

Forschergruppen der Heinrich-Heine-Universität Düsseldorf

Graduiertenkollegs der Heinrich-Heine-Universität Düsseldorf

Zentrale wissenschaftliche Einrichtungen der Heinrich-Heine-Universität Düsseldorf

Humanwissenschaftlich-Medizinisches Forschungszentrum

Biotechnologie – Ein gemeinsamer Forschungsschwerpunkt der Heinrich-Heine-Universität Düsseldorf und des Forschungszentrums Jülich

MICHAEL LEUSCHEL und JENS BENDISPOSTO

Das ProB-Werkzeug zur Validierung formaler Softwaremodelle

Formale Methoden und die B-Methode

Immer häufiger werden Softwarelösungen auch in kritischen Domänen wie zum Beispiel in Steuereinheiten von Zügen, Flugzeugen oder Industrieanlagen, medizinischen Systemen, aber auch geschäftskritischen Systemen eingesetzt. Fehler in solchen Systemen können zu schwerwiegenden finanziellen Folgen, zur Schädigung der Umwelt oder sogar zum Verlust von Menschenleben führen. Am 15. Januar 1990 kam es beispielsweise bei der amerikanischen Telefongesellschaft AT&T zu einem neunstündigen Ausfall, bei dem die Hälfte aller Ferngespräche nicht vermittelt werden konnte.[1] Ursache war ein kleiner Fehler in der Software, der vier Wochen vor dem Ausfall durch eine Programmoptimierung in die Software eingebracht wurde. Der Schaden, der bei AT&T und den Kunden entstand, lag bei mehreren Hundert Millionen Dollar. Ein Beispiel, bei dem sogar Menschen zu Schaden kamen, ist ein Softwarefehler in dem Therac-25-Linearbeschleuniger,[2] der in der Strahlentherapie benutzt wurde. Im Zeitraum von 1985 bis 1987 kam es durch eine mangelhaft gelöste Synchronisierung von nebenläufigen Prozessen dazu, dass mehrere Patienten mit einer viel zu hohen und in einigen Fällen tödlichen Strahlendosis bestrahlt wurden. Üblicherweise werden Tests verwendet, um die Qualität der Software zu sichern. Eine Studie des National Institute of Standards and Technology (NIST) hat jedoch geschätzt, dass die jährlichen Kosten in den vereinigten Staaten, die durch eine mangelhafte Test-Infrastruktur entstehen, etwa 59,5 Milliarden Dollar betragen.[3] Des Weiteren schätzt die Studie, dass die Kosten durch Verbesserung der Tests um etwa die Hälfte reduziert werden können. Zusätzlich zu Tests ist es bei kritischen Systemen angemessen, eine formale Methode zur Entwicklung zu verwenden.

Formale Methoden haben das Ziel, fehlerfreie Software zu entwickeln. Zu diesem Zweck bedient man sich eines mathematisch geprägten Entwicklungsprozesses. Es wird eine formale Spezifikation in einer mathematischen Sprache wie zum Beispiel Z[4], VDM[5], ASM[6] oder B[7] erstellt. Die Schlüssigkeit der formalen Spezifikation wird mathematisch bewiesen und die Spezifikation wird schrittweise in ein Programm transformiert, das sich durch einen Computer ausführen lässt. Es ist somit garantiert, dass das Programm per Konstruktion korrekt ist in dem Sinne, dass es die Spezifikation erfüllt.

---

1 Vgl. Neumann (1990).
2 Vgl. Leveson und Turner (1993).
3 Vgl. National Institute of Standards and Technology (2002).
4 Vgl. Abrial *et al.* (1980) sowie Spivey (^{2}1992).
5 Vgl. Jones (1986).
6 Vgl. Börger (2003).
7 Vgl. Abrial (1996).

Die B-Methode wurde in den 1980er Jahren von Jean-Raymond Abrial entworfen. Ein Designaspekt der B-Methode war es, eine gute Unterstützung durch Werkzeuge zu ermöglichen. Bekannte Werkzeuge für die B-Methode sind AtelierB[8] und die nichtkommerzielle Version B4Free sowie das BToolkit[9]. B4Free beinhaltet ein Werkzeug, das aus einer Spezifikation die Beweisbedingungen (so genannte *Proof Obligations*) extrahiert und automatische Beweiser auf diese Bedingungen anwendet. *Proof Obligations*, die von den Tools nicht automatisch bewiesen werden können, werden vom Benutzer interaktiv bearbeitet. Die automatischen Beweiser in B4Free sind die gleichen, die auch im kommerziellen AtelierB verwendet werden; zusätzlich kann AtelierB aber Ada- oder C-Code generieren.

Die B-Methode ist im europäischen Raum eine der meistbenutzten formalen Methoden und wurde in mehreren Industrieprojekten erfolgreich angewandt. Ein Beispiel ist die Linie 14 der Pariser Métro, auch Météor (Métro est-ouest rapide) genannt.[10] Die Züge fahren vollautomatisch und führerlos. Die Controller von Météor wurden mit der B-Methode entwickelt, die Spezifikation umfasste 100.000 Zeilen und es wurden 28.000 Beweise geführt. Mittlerweile wurde die B-Methode von Siemens und ClearSy auch für verschiedene andere Steuersysteme von Zügen und U-Bahnen erfolgreich angewandt.[11] Andere typische Anwendungsgebiete der B-Methode sind zum Beispiel die Entwicklung von Software für SmartCards[12], die Fehlerdiagnose für moderne Kraftfahrzeuge[13] oder das Coppilot-System für die automatischen Türen der Pariser U-Bahn-Linie 13.[14]

Für weniger kritische Systeme ist es oft unverhältnismäßig, eine formale Methode zur Entwicklung zu verwenden. Anthony Hall (1990) beschreibt, dass es als Alternative zu einem rigorosen Ansatz oft ausreicht, die abstrakte Spezifikation formal zu fixieren. Diese Spezifikation wird dann von den Entwicklern als Referenzdokument benutzt, anstatt sie schrittweise zu transformieren.

Animation und Validierung formaler Modelle

Da die Anforderungen an Software immer mehr zunehmen, wächst auch die Größe und Komplexität der formalen Spezifikationen oder Modelle. Ein kritischer Punkt der formalen Methoden ist die Korrektheit der Ausgangsspezifikation. Ist diese fehlerhaft, so ist die entwickelte Software trotz allen formalen Aufwands generell auch fehlerhaft. In den letzten Jahren wurde deshalb viel Forschungsaufwand betrieben, um die Korrektheit der Ausgangsspezifikation zu garantieren. Die Probleme sind dabei wie folgt:

- Problem 1: Die formalen Spezifikationen und Modelle sind nicht von einem Computer ausführbar. Fehler im Verhalten oder Abwesenheit von erwünschter Funktionalität lassen sich deshalb oft viel zu spät entdecken (nämlich erst, nachdem die Software entwickelt worden ist). Ein klassisches Beispiel ist eine Spezifikation, die zwar sicher ist und keine Fehlerzustände zulässt, aber nicht genügend Funktionalität aufweist. Ein Beispiel wäre ein Zug-Steuersystem, das es keinem Zug erlaubt, sich zu bewegen.

[8] Vgl. Steria (1996).
[9] Vgl. C-Core (UK) Limited und UK Oxon (1999).
[10] Vgl. Behm *et al.* (1999).
[11] Vgl. Dollé *et al.* (2003).
[12] Vgl. Casset (2002).
[13] Vgl. Pouzancre (2003) sowie Pouzancre und Pitzalis (2003).
[14] Siehe http://www.clearsy.com/html/Coppilot.htm.

Dies garantiert zwar die Abwesenheit von Zusammenstößen, es fehlt aber eindeutig an Funktionalität.

Eine Lösung dieses Problems ist die so genannte *Animation* der formalen Spezifikation. Mit Hilfe eines Animators kann ein Benutzer das formale Modell ausloten und verschiedene Anwendungsszenarien überprüfen.

- Problem 2: Die formalen Spezifikationen und Modelle können sehr groß werden, und Fehler können trotz menschlicher Analyse und ausgiebiger Animation vorhanden sein.

  Eine Möglichkeit, dieses Problem zu lösen, ist, die Spezifikation systematisch auf verschiedene kompakte und vom Menschen verständliche Eigenschaften zu überprüfen. Eine Technik, die dies erlaubt, ist das *Model Checking*,[15] das ein Modell systematisch auf temporale Eigenschaften überprüfen kann. Eine solche temporale Eigenschaft ist zum Beispiel, dass ein System es zu keinem Zeitpunkt erlaubt, dass sich zwei Züge auf dem gleichen Gleisstück befinden. Falls eine Eigenschaft wahr ist, hat man das Vertrauen in die Spezifikation erhöhen können, da sie garantiert, dass die temporale Eigenschaft unter allen erdenklichen Umständen wahr ist. Falls die Eigenschaft falsch ist, bekommt man ein Gegenbeispiel (*counter example*), das klarstellt, unter welchen Umständen der Fehler auftauchen kann.

Für das *Model Checking* gibt es mittlerweile eine große Anzahl an Werkzeugen. Die zwei bekanntesten sind wohl SMV zur Überprüfung von *Computation Tree Logic*-Formeln (CTL-Formeln) anhand symbolischer Methoden (BDDs)[16] und SPIN[17], das *Linear Temporal Logic*-Formeln (LTL-Formeln) anhand von Automaten über unendliche Eingaben überprüft.[18]

An unserem Lehrstuhl haben wir den Animator und Model-Checker ProB entwickelt. Dieses Werkzeug wurde mit Hilfe der logischen Programmierung entwickelt, die wir in folgender Sektion einführen.

Formale Verfikation durch logische Programmierung

Logische Programmierung

Die logische Programmierung entstand 1972 und basiert auf der Idee, dass eine Unterklasse von logischen Theorien mit Hilfe von Resolution effizient ausgeführt werden kann. Ein Ziel der logischen Programmierung ist es, ein Problem in mathematischer Logik zu beschreiben und den Computer zu benutzen, um diese Probleme automatisch durch logische Inferenz zu lösen. Man spricht deshalb auch von *deklarativer Programmierung*, da das Problem nur *beschrieben* wird und der Programmierer nicht wie in der klassischen imperativen Programmierung die einzelnen Lösungsschritte aufzählen muss. Seit Mitte der 1980er Jahre wurde die logische Programmierung mit Constraints erweitert, was zu einer Reihe von neuen Anwendungsgebieten führte.

Die logische Programmierung ist besonders für Anwendungen in der künstlichen Intelligenz, für Expertensysteme und Datenbanken oder für den schnellen Bau von Prototypen

[15] Vgl. Clarke *et al.* (1999).
[16] Vgl. Bryant (1986), Bryant (1992), Burch *et al.* (1992) sowie McMillan (1993).
[17] Vgl. Holzmann (1997).
[18] Vgl. Vardi und Wolper (1986).

beliebt. Wir interessieren uns am Lehrstuhl besonders für die logische Programmierung, um damit Werkzeuge für die formalen Methoden zu entwickeln. In der Tat hat sich die logische Programmierung als besonders nützlich erwiesen, um andere Sprachen oder Formalismen zu modellieren.

Formale Verifikation

Ein wichtiges Forschungsgebiet ist die Anwendung der logischen Programmierung für die formale Verifikation anderer Formalismen und Programmiersprachen. Dieses Anwendungsgebiet wurde von unserem Lehrstuhl in verschiedenen Forschungsprojekten vorangetrieben.[19] Wir haben auch eine Reihe von Workshops zu diesem Thema organisiert.[20]

Einige unserer wissenschaftlichen Arbeiten auf diesem Gebiet sind wie folgt:

- Leuschel und Massart (2000) beschreiben einen Model-Checker für CTL in Prolog. Dieser Model-Checker eignet sich besonders für eine automatische Spezialisierung.[21] Dies ermöglicht es, den Model-Checker automatisch für bestimmte Modelle und Formeln zu optimieren. Ein anderes Ziel war die automatische Analyse des Model-Checkers, um dadurch Eigenschaften über Modelle mit unendlichen Zustandsräumen herausfinden zu können.
- Die erfolgreiche Anwendung obiger Techniken auf Petrinetze und andere Systeme (so genannte *Well-Structured Transition Systems*) mit unendlichen Zustandsräumen wurde in Leuschel und Lehmann (2000b) sowie Leuschel und Lehmann (2000a) beschrieben.
- In Leuschel (2004) wurde eine neue integrierte Methode zur Analyse und Spezialisierung logischer Programme entwickelt. Ein Anwendungsgebiet ist die präzisere Analyse von Systemen mit unendlichen Zustandsräumen.[22]
- In Leuschel (2001) wurde ein Animator für die Prozessalgebra CSP[23] entwickelt. Mit Hilfe des obigen Model-Checkers war es dann möglich, CSP-Prozesse auf CTL-Eigenschaften zu überprüfen.
- In Farwer und Leuschel (2004) wurden obige Techniken erfolgreich auf Objekt-Petrinetze angewandt.
- Von Relevanz ist auch die Arbeit von Craig und Leuschel (2005), in der genetische Algorithmen zum Durchlaufen großer Konfigurationsräume bei der Programmoptimierung erfolgreich eingesetzt worden sind.

Das ProB-Werkzeug für die B-Methode

Die Entwicklung des ProB-Werkzeugs begann im Jahr 2001.[24] Eine erste voll funktionsfähige Version gab es im Jahr 2003.[25] Seitdem wurde ProB ständig erweitert und verbessert:

[19] Die britischen Forschungsprojekte iMoc und ABCD sowie das EU-Framework-5-Projekt ASAP und das EU-Framework-6-Projekt Rodin.
[20] Vgl. Leuschel *et al.* (2000), Leuschel *et al.* (2001a) sowie Leuschel und Ultes-Nitsche (2002).
[21] Vgl. Leuschel und Bruynooghe (2002).
[22] Vgl. Leuschel und Gruner (2001).
[23] Vgl. Roscoe (1999).
[24] Vgl. Leuschel *et al.* (2001a).
[25] Vgl. Leuschel und Butler (2003).

- Verschiedene Techniken zur Visualisierung großer Zustandsräume wurden entwickelt.[26] Mit diesen Techniken können, je nach Anwendung, Zustandsräume mit mehreren Tausend Knoten kompakt und für den Benutzer übersichtlich dargestellt werden.
- Eine Verknüpfung von B-Modellen mit der Prozessalgebra CSP wurde ausgearbeitet und entwickelt.[27] Anwendungen dieser Verknüpfung sind die Validierung komplizierter temporaler Eigenschaften (die CSP-Spezifikation überwacht das formale B-Modell), aber auch die bequemere Spezifikation von Systemen.[28]
- Ein Algorithmus und eine Implementation zur automatischen Verfeinerungsüberprüfung wurden in 2005 entwickelt.[29] Dies ermöglicht es, mit ProB zu überprüfen, ob eine B-Spezifikation eine andere verfeinert. Falls dies nicht der Fall ist, generiert ProB einen Ablauf der Verfeinerungsmaschine, der den Fehler aufzeigt.
- Ein Algorithmus zur automatische Generierung von Testfällen wurde in Satpathy *et al.* (2005) präsentiert.
- In Bendisposto und Leuschel (im Druck b) wird eine Methode vorgestellt werden, um eine formale Spezifikation durch Verwendung von Webtechnologien zu visualisieren (siehe Abb. 1).
- Ein Editor für B-Spezifikationen wird in Bendisposto und Leuschel (im Druck a) präsentiert werden. Dieser Editor bietet für integrierte Entwicklungsumgebungen typische Eigenschaften wie Code-Vervollständigung, Syntax-Highlighting, Code-Navigation, Fehlerhervorhebung und automatische Korrektur an. Da unser Editor auf dem Open-Source-Framework Eclipse basiert, kann er sowohl in ProB als auch in anderen eclipsebasierten Werkzeugen für B verwendet werden.

ProB wurde mit Erfolg zur Validierung verschiedener Modelle aus der Industrie benutzt. Es konnte zum Beispiel das Modell einer automatischen Fahrzeugsteuereinheit von Volvo überprüfen.[30] Die Überprüfung war vollautomatisch und dauerte einige Minuten. ProB wurde auch von Nokia erfolgreich auf verschiedene formale Modelle der neuen *Mobile Internet Technical Architecture* (MITA) angewandt.[31] Etliche Fehler wurden mit Hilfe der Animation früh entdeckt. ProB wurde und wird außerdem an verschiedenen Universitäten für die Lehre der B-Methode eingesetzt.

Einige aktuelle Forschungsarbeiten

Aktuell wird ProB innerhalb des EU-Projektes IST 511599 RODIN (*Rigorous Open Development Environment for Complex Systems*)weiterentwickelt. Insbesondere wurde eine Eclipse-Version von ProB entwickelt (siehe Abb. 2), und eine Erweiterung für den neuen EventB-Standard ist geplant.

Das größte Problem des *Model Checking* ist die exponentielle Explosion der Anzahl der möglichen Zustände (*state explosion problem*). In der Tat können schon relativ kleine

[26] Vgl. Leuschel und Turner (2005).
[27] Vgl. Butler und Leuschel (2005).
[28] Der Ablauf des Systems wird in CSP spezifiziert; die Daten und Operationen in B.
[29] Vgl. Leuschel und Butler (2005).
[30] Vgl. Leuschel und Butler (2003).
[31] Siehe RODIN Deliverable D8 (http://rodin.cs.ncl.ac.uk/).

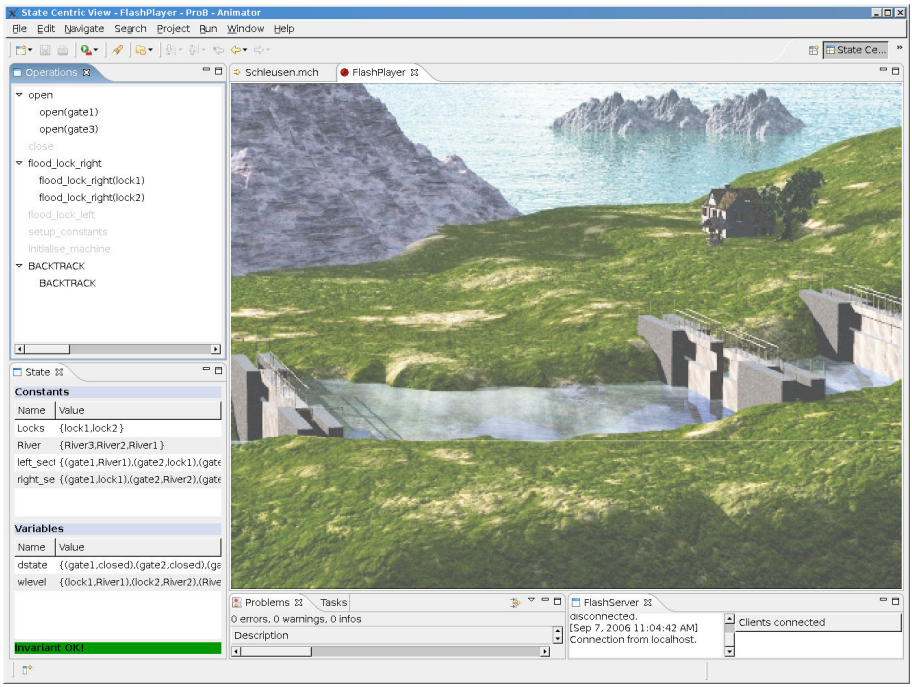

Abb. 1: Grafische Visualisierung einer Spezifikation

Modelle mehr Zustände haben, als es Atome im Universum gibt. Dieses Problem erfasst leider auch unser Werkzeug ProB. Es gibt aber eine große Anzahl an Verbesserungen und Erweiterungen, die es erlauben, *Model Checking* trotzdem auf realistische Modelle anzuwenden. Beispiele sind die *Partial Order Reduction*,[32] die *Symmetry Reduction*[33] und Petrinetzentfaltungen.[34] Die Parallelisierung des Model-Checkers ist eine andere Möglichkeit, die auch viel Anklang gefunden hat.[35] Neuerdings gibt es auch veschiedene Versuche, den *Model Checking*-Prozess durch Heuristiken (*Directed Model Checking*)[36] oder auch anhand von genetischen Algorithmen[37] zu verbessern.

Eine besondere Eigenschaft von ProB im Vergleich zu Model-Checkern wie Spin und SMV ist die hohe Ausdruckskraft der Spezifikationssprache. Der Aufwand bei der Analyse einzelner Zustände und der Berechnung von Übergängen in andere Zustände ist deshalb sehr groß. Andererseits glauben wir, dass durch diese größere Granularitiät ein großes Potenzial für intelligente Techniken zur Verbesserung der Effizienz vorliegt. Wir haben schon

[32] Vgl. Wolper und Godefroid (1993) sowie Godefroid (1996).
[33] Vgl. Ip und Dill (1993) sowie Clarke *et al.* (1993).
[34] Vgl. Esparza *et al.* (2002).
[35] Vgl. Lerda und Sisto (1999), Stern und Dill (2001), Ben-David *et al.* (2003), Dräger *et al.* (2006) sowie Leucker und van de Pol (2006).
[36] Vgl. Groce und Visser (2002) sowie Dräger *et al.* (2006).
[37] Vgl. Godefroid und Khurshid (2004).

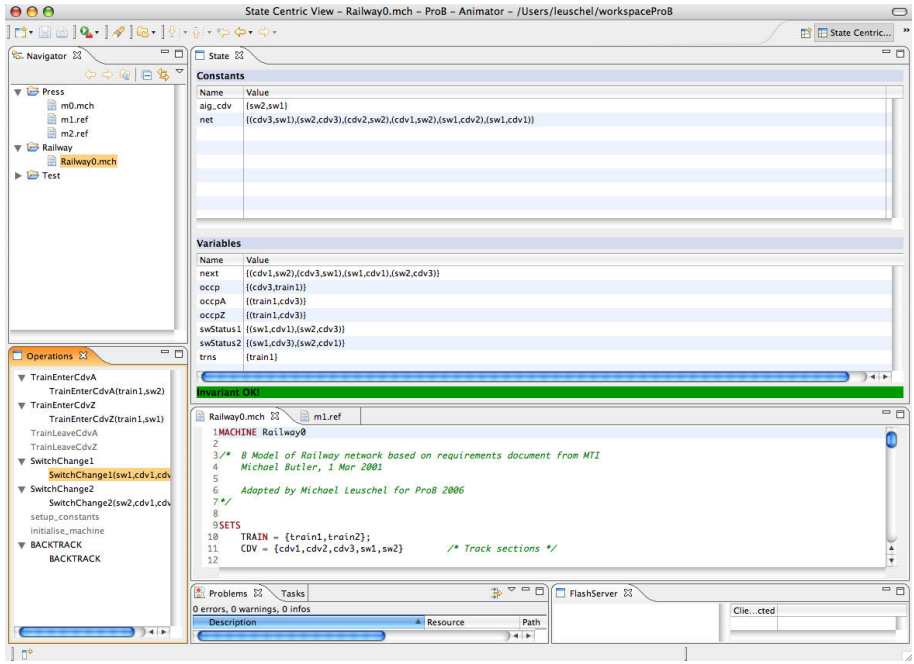

Abb. 2: Eclipse-Version von ProB in Aktion

mit Erfolg Methoden entworfen, die die Symmetrie in B-Spezifikationen ausnützen, um beträchtliche Effizienzgewinne zu erzielen.

Fazit

Die formalen Methoden ermöglichen es prinzipiell, Software mit sehr geringen Fehlerraten zu entwickeln, finden aber wegen ihrer Komplexität und noch zu geringen Werkzeugunterstützung keinen verbreiteten Einsatz. Es ist unser Ziel, sowohl den Nutzen formaler Methoden zu erhöhen als auch die formalen Methoden leichter zugänglich zu machen. Wir hoffen, dass wir diesem Ziel mit unserem Werkzeug ProB einen Schritt näher gekommen sind. Unser Werkzeug kann für akademische Zwecke kostenlos unter der Adresse http://www.stups.uni-duesseldorf.de/ProB/ heruntergeladen werden.

Literatur

ABRIAL, Jean-Raymond, Steve A. SCHUMAN und Bertrand MEYER. „Specification Language“, in: R. M. MCKEAG und A. M. MACNAGHTEN (Hrsg.). *On the Construction of Programs: An Advanced Course*. Cambridge 1980, 343-410.

ABRIAL, Jean-Raymond. *The B-Book*. Cambridge 1996.

B-CORE (UK) LIMITED und UK OXON. *B-Toolkit, On-line manual*. 1999. http://www.b-core.com/ONLINEDOC/Contents.html.

BEHM, Patrick, Paul BENOIT, Alain FAIVRE und Jean-Marc MEYNADIER. „Météor: A Successful Application of B in a Large Project“, in: Jeannette M. WING, Jim WOODCOCK und Jim DAVIES

(Hrsg.). *World Congress on Formal Methods*. Bd. 1708. Berlin und Heidelberg 1999. (Lecture Notes in Computer Science)

BEN-DAVID, Shoham, Orna GRUMBERG, Tamir HEYMAN und Assaf SCHUSTER. „Scalable distributed on-the-fly symbolic model checking“, *International Journal on Software Tools for Technology Transfer (STTT)* 4 (4)(2003), 496-504.

BENDISPOSTO, Jens und Michael LEUSCHEL. „BE4: The B Extensible Eclipse Editing Environment“, in: JULLIAND und KOUCHNARENKO (Hrsg.). *Proceedings of the 7th International B Conference (B2007)*. Berlin und Heidelberg (im Druck a).

BENDISPOSTO, Jens und Michael LEUSCHEL. „A Generic Flash-based Animation Engine for ProB“, in: JULLIAND und KOUCHNARENKO (Hrsg.). *Proceedings of the 7th International B Conference (B2007)*. Berlin und Heidelberg (im Druck b).

BÖRGER, Egon. *Abstract State Machines*. Berlin 2003.

BRYANT, Randal E. „Graph-Based Algorithms for Boolean Function Manipulation“, *IEEE Transactions on Computers* 35 (8)(1986), 677-691.

BRYANT, Randy. „Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams“, *ACM Computing Surveys* 24 (3)(1992), 293-318.

BURCH, J. R., E. M. CLARKE, K. L. MCMILLAN, D. L. DILL und L. J. HWANG. „Symbolic model checking: 10^{20} states and beyond“, *Information and Computation* 98 (2)(1992), 142-170.

BUTLER, Michael und Michael LEUSCHEL. „Combining CSP and B for Specification and Property Verification“, in: FITZGERALD, HAYES und TARLECKI (Hrsg.). *Proceedings of Formal Methods 2005*. Berlin und Heidelberg 2005, 221-236. (LNCS 3582)

CASSET, Ludovic. „Development of an Embedded Verifier for Java Card Byte Code Using Formal Methods“, in: *Formal Methods Europe* 2002, 290-309.

CLARKE, Edmund M., Thomas FILKORN und Somesh JHA. „Exploiting Symmetry In Temporal Logic Model Checking“, in: Costas COURCOUBETIS (Hrsg.). *Computer Aided Verification*. Bd. 697. Berlin und Heidelberg 1993. (Lecture Notes in Computer Science)

CLARKE, Edmund M., Orna GRUMBERG und Doron PELED. *Model Checking*. Boston 1999.

CRAIG, Stephen-John und Michael LEUSCHEL. „Self-tuning resource aware specialisation for Prolog“, in: *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*. New York 2005, 23-34.

DOLLÉ, Daniel, Didier ESSAMÉ und Jérôme FALAMPIN. „B dans le tranport ferroviaire. L'expérience de Siemens Transportation Systems“, *Technique et Science Informatiques* 22 (1)(2003), 11-32.

DRÄGER, Klaus, Bernd FINKBEINER und Andreas PODELSKI. „Directed Model Checking with Distance-Preserving Abstractions“, in: Antti VALMARI (Hrsg.). *SPIN*. Bd. 3925. Berlin 2006, 19-34. (Lecture Notes in Computer Science)

ESPARZA, Javier, Stefan RÖMER und Walter VOGLER. „An Improvement of McMillan's Unfolding Algorithm“, *Formal Methods in System Design* 20 (3)(2002), 285-310.

FARWER, Berndt und Michael LEUSCHEL. „Model checking object Petri nets in Prolog“, in: *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*. New York 2004, 20-31.

GODEFROID, Patrice. *Lecture Notes in Computer Science*. Bd. 1032: *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. Berlin 1996.

GODEFROID, Patrice und Sarfraz KHURSHID. „Exploring very large state spaces using genetic algorithms“, *International Journal on Software Tools for Technology Transfer (STTT)* 6 (2)(2004), 117-127.

GROCE, Alex und Willem VISSER. „Heuristic Model Checking for Java Programs“, in: Dragan BOSNACKI und Stefan LEUE (Hrsg.). *SPIN*. Bd. 2318. Berlin 2002, 242-245. (Lecture Notes in Computer Science)

HALL, Anthony J. „Seven Myths of Formal Methods“, *IEEE Software* 7 (5)(1990), 11-19.

HOLZMANN, Gerard J. „The Model Checker SPIN“, *IEEE Transaction on Software Engineering* 23 (5)(1997), 279-295.

IP, Norris C. und David L. DILL. „Better Verification Through Symmetry“, *Computer Hardware Description Languages and their Applications* (1993), 97-111.

JONES, Cliff B. *Systematic Software Development using VDM*. Upper Saddle River, NJ, 1986.

LERDA, Flavio und Riccardo SISTO. „Distributed-Memory Model Checking with SPIN“, in: Dennis DAMS, Rob GERTH, Stefan LEUE und Mieke MASSINK (Hrsg.). *SPIN*. Bd. 1680. Berlin 1999. 22-39 (Lecture Notes in Computer Science)

LEUCKER, Martin und Jaco VAN DE POL (Hrsg.). *Proceedings of the 4th Workshop on Parallel and Distributed Methods for Verification*. Bd. 135/2. Berlin 2006. (Electronic Notes in Computer Science)

LEUSCHEL, Michael, Andreas PODELSKI, C. R. RAMAKRISHNAN und Ulrich ULTES-NITSCHE (Hrsg.). *Proceedings of the International Workshop on Verification and Computational Logic (VCL'2000)*. Southampton 2000. (Verfügbar als Technical Report DSSE-TR-2000-6)

LEUSCHEL, Michael und Helko LEHMANN. „Coverability of Reset Petri Nets and other Well-Structured Transition Systems by Partial Deduction“, in: John LLOYD (Hrsg.). *Proceedings of the International Conference on Computational Logic (CL'2000)*. Berlin 2000a, 101-115. (LNAI 1861)

LEUSCHEL, Michael und Helko LEHMANN. „Solving Coverability Problems of Petri Nets by Partial Deduction“, in: Maurizio GABBRIELLI und Frank PFENNING (Hrsg.). *Proceedings of PPDP'2000*. Montreal 2000b, 268-279.

LEUSCHEL, Michael und Thierry MASSART. „Infinite State Model Checking by Abstract Interpretation and Program Specialisation“, in: Annalisa BOSSI (Hrsg.). *Logic-Based Program Synthesis and Transformation. Proceedings of LOPSTR'99*. Venedig 2000, 63-82. (LNCS 1817)

LEUSCHEL, Michael. „Design and Implementation of the High-level Specification Language CSP(LP) in Prolog“, in: I. V. RAMAKRISHNAN (Hrsg.). *Proceedings of PADL'01*. Berlin 2001, 14-28. (LNCS 1990)

LEUSCHEL, Michael und Stefan GRUNER. „Abstract Partial Deduction using Regular Types and its Application to Model Checking“, in: Alberto PETTOROSSI (Hrsg.). *Proc. of 11th Int'l Workshop on Logic-based Program Synthesis and Transformation, LOPSTR'2001*. Berlin 2001, 91-110. (LNCS 2372)

LEUSCHEL, Michael, Laksono ADHIANTO, Michael BUTLER, Carla FERREIRA und Leonid MIKHAILOV. „Animation and Model Checking of CSP and B using Prolog Technology“, in: Michael LEUSCHEL, Andreas PODELSKI, C. R. RAMAKRISHNAN und Ulrich ULTES-NITSCHE (Hrsg.). *Proceedings of VCL'2001*. Southampton 2001a, 97-109.

LEUSCHEL, Michael, Thierry MASSART und Andrew CURRIE. „How to Make FDR Spin: LTL Model Checking of CSP by Refinement“, in: Jose N. OLIVIERA und Pamela ZAVE (Hrsg.). *FME'2001: Formal Methods for Increasing Software Productivity*. Berlin 2001b, 99-118. (LNCS 2021)

LEUSCHEL, Michael, Andreas PODELSKI, C. R. RAMAKRISHNAN und Ulrich ULTES-NITSCHE (Hrsg.). *Proceedings of the ACM-Sigplan Workshop on Verification and Computational Logic (VCL'2001)*. Southampton 2001. (Verfügbar als Technical Report DSSE-TR-2001-3)

LEUSCHEL, Michael und Maurice BRUYNOOGHE. „Logic program specialisation through partial deduction: Control issues“, *Theory and Practice of Logic Programming* 2 (4)(5)(2002), 461-515.

LEUSCHEL, Michael und Ulrich ULTES-NITSCHE (Hrsg.). *Proceedings of the ACM-Sigplan Workshop on Verification and Computational Logic (VCL'2002)*. Pittsburgh 2002.

LEUSCHEL, Michael und Michael BUTLER. „ProB: A Model Checker for B“, in: Keijiro ARAKI, Stefania GNESI und Dino MANDRIOLI (Hrsg.). *FME 2003: Formal Methods*. Berlin 2003, 855-874. (LNCS 2805)

LEUSCHEL, Michael. „A Framework for the Integration of Partial Evaluation and Abstract Interpretation of Logic Programs“, *ACM Transactions on Programming Languages and Systems* 26 (3)(2004), 413-463.

LEUSCHEL, Michael und Michael BUTLER. „Automatic Refinement Checking for B“, in: Kung-Kiu LAU und Richard BANACH (Hrsg.). *Proceedings ICFEM'05*. Berlin 2005, 345-359. (LNCS 3785)

LEUSCHEL, Michael und Edward TURNER. „Visualizing Larger States Spaces in ProB“, in: Helen TREHARNE, Steve KING, Martin HENSON und Steve SCHNEIDER (Hrsg.). *Proceedings ZB'2005*. Berlin 2005, 6-23. (LNCS 3455)

LEVESON, Nancy und Clark S. TURNER. „An Investigation of the Therac-25 Accidents“, *IEEE Computer* 26 (7)(1993), 18-41.

MCMILLAN, Kenneth L. *Symbolic Model Checking*. Dissertation. Boston 1993.

NEUMANN, Peter G. „Risk to the public in computers and related systems“, *ACM SIGSOFT Software Engineering Notes* 15 (2)(1990), 3-22.

POUZANCRE, Guilhem. „How to Diagnose a Modern Car with a Formal B Model?“, in: Didier BERT, Jonathan P. BOWEN, Steve KING und Marina A. WALDÉN (Hrsg.). *ZB*. Bd. 2651. Berlin 2003, 98-100. (Lecture Notes in Computer Science)

POUZANCRE, Guilhem und Jean-Philippe PITZALIS. „Modélisation en B événementiel des fonctions mécaniques, électriques et informatiques d'un véhicule“, *Technique et Science Informatiques* 22 (1)(2003), 119-128.

ROSCOE, A. W. *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, 1999.

SATPATHY, Mannu, Michael LEUSCHEL und Michael BUTLER. „ProTest: An Automatic Test Environment for B Specifications“, *Electronic Notes in Theoretical Computer Science* 111 (2005), 113-136.

SPIVEY, J. M. *The Z Notation: A Reference Manual*. Upper Saddle River, NJ, ^{2}1992).

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. „The Economic Impacts of Inadequate Infrastructure for Software Testing“, in: *Planning Report*. Gaithersburg 2002.

STERIA, France Aix-en-Provence: *Atelier B, User and Reference Manuals* (1996). - Available at http://www.atelierb.societe.com/index_uk.html.

STERN, Ulrich und David L. DILL. „Parallelizing the Murϕ Verifier“, *Formal Methods in System Design* 18 (2)(2001), 117-129.

VARDI, M. Y. und P. WOLPER. „An automata-theoretic approach to automatic program verification“, in: *Proceedings of LICS'86*. Cambridge 1986, 332-344.

WOLPER, Pierre und Patrice GODEFROID. „Partial-order methods for temporal verification“, in: E. BEST (Hrsg.). *4th. Int. Conf. on Concurrency Theory (CONCUR), LNCS 715*. Berlin 1993, 233-246.

ISBN 3-9808514-4-3