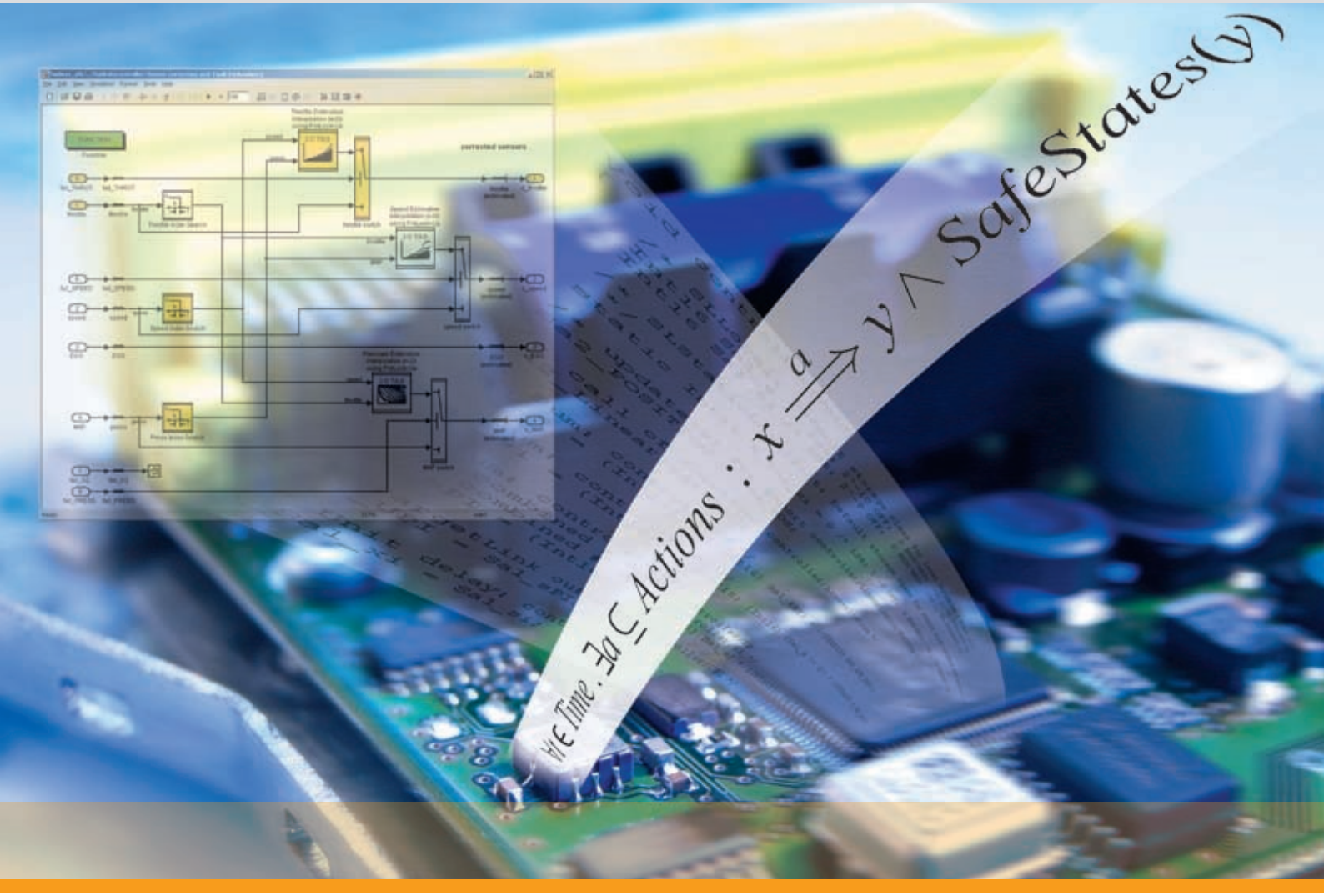

Formale Methoden in der Softwareentwicklung Mathematisch beweisbare Schlussfolgerungen ergänzen traditionelle Methoden

Wenn es darum geht, komplexe Systeme effizient zu entwickeln, stoßen traditionelle, nicht-formale Techniken der Softwareentwicklung an ihre Grenzen. Das Institut für Formale Methoden der Informatik an der Universität Stuttgart stellt fest: Werkzeuge wie Programmierumgebungen, automatische Kodegenerierung aus grafischen Entwicklungswerkzeugen, Visualisierungstechniken wie UML-Diagramme, und andere sind zwar notwendig. Doch es fehlt ihnen oft eine Komponente, die es erlaubt, detaillierte und beweisbar korrekte Schlussfolgerungen über ein komplexes System zu ziehen. Formale Methoden in der Softwaretechnik bieten Ansätze, diese Lücke zu füllen.

1 Einleitung

Der steigende Einsatz von Elektronik im Kraftfahrzeug führt zu einem stetig steigenden Anteil der Softwareproduktionskosten an den Gesamtkosten einer Fahrzeugentwicklung. Die Aufwendungen für den Softwareentwicklungsprozess, zum Beispiel für die Erstellung, Anpassung, Test und Wartung von Software, bilden dabei nur einen Teil der Kosten. Nicht zu überblickende Mehrkosten entstehen, wenn Software ein Fehlverhalten einer Komponente verursacht, das bis zum Ausfall eines Fahrzeuges führen kann. Der Einsatz formaler Methoden in der Softwareentwicklung kann einen wesentlichen Beitrag zur Reduktion solcher Kosten leisten.

Bei der Bezeichnung „Formale Methoden" handelt es sich um einen Sammelbegriff, der eine Vielzahl von Ansätzen, Herangehensweisen und Techniken, die auf Mathematik und Logik basieren, zusammengefasst. Im Abschnitt 2 werden die formalen Methoden in ihrer Allgemeinheit betrachtet – ohne Anspruch auf Vollständigkeit. In Abschnitt 3 wird eine beispielhafte Anwendung einer formalen Methode im Entwicklungsprozess eines größeren Echtzeitsystems skizziert. Die Universität Stuttgart und die Daimler AG setzten hier das Model-Checking-Verfahren von der Temporallogik TCTL auf Zeitautomaten ein, um den Designprozess eines adaptiven Bremssystems zu begleiten und frühzeitig Fehler im Design zu finden.

2 Formale Methoden

Erste Erfahrungen mit Formalen Methoden in der Verifikation von Software konnten bereits in den 50er und 60er Jahren gesammelt werden. Die anfänglich überschwänglichen und unrealistischen Erwartungen wurden bald enttäuscht. Der Irrtum, eine „absolute Sicherheit" erreichen zu können und ein „in jeglicher Hinsicht korrektes Programms" zu erhalten, führte zu Frustrationen. Allein die Formalisierung der intendierten Eigenschaften eines Programms ist ein Prozess, der sich weder formal beschreiben noch verifizieren lässt. Erst wenn eine formale Spezifikation vorliegt, lassen sich weitere Entwicklungsschritte bis hin zum endgültigen Maschinenprogramm mit formalen Methoden begleiten.

Ein weiteres Missverständnis besteht darin, dass sich formale Methoden ausschließlich zur Verifikation von Programmen nutzten lassen. Sicherlich ist die Verifikation eines Programms gegen eine Spezifikation das klassische Anwendungsgebiet für einen formalen Ansatz. Ein weiterer großer Nutzen im Einsatz formaler Spezifikationen gegenüber einem nicht-formalen Entwicklungsprozess besteht aber in der Unzweideutigkeit der gemachten Aussagen. Wenn eine formale Beschreibung eines Systems vorliegt, können nicht nur genaue Schlüsse gezogen und Eigenschaften bewiesen werden, sondern sie bildet auch eine vorteilhafte Arbeitsgrundlage für alle Beteiligten. Weder ein Auftraggeber noch ein

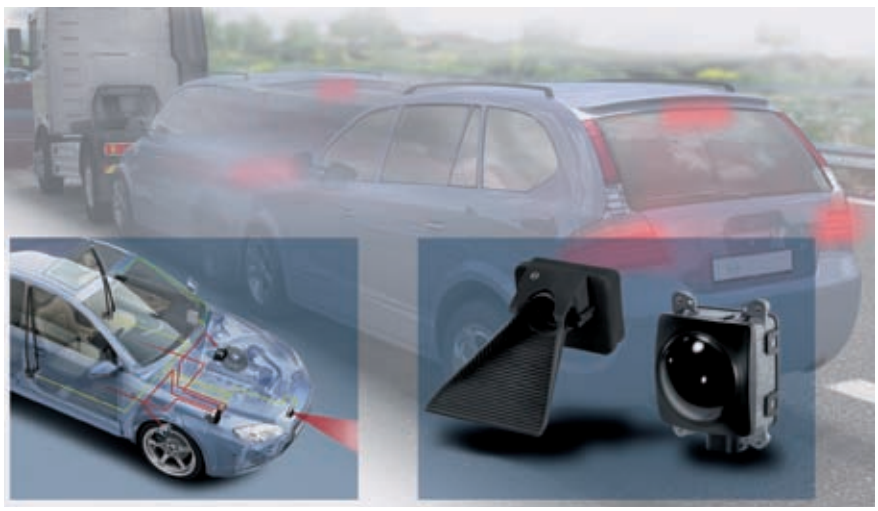

Bild 1: Teilbereich einer Fallstudie: ein Notfallbremsassistent (EBA)

Der Autor

Dr. Dirk Nowotka arbeitet am Institut für Formale Methoden der Informatik an der Universität Stuttgart.

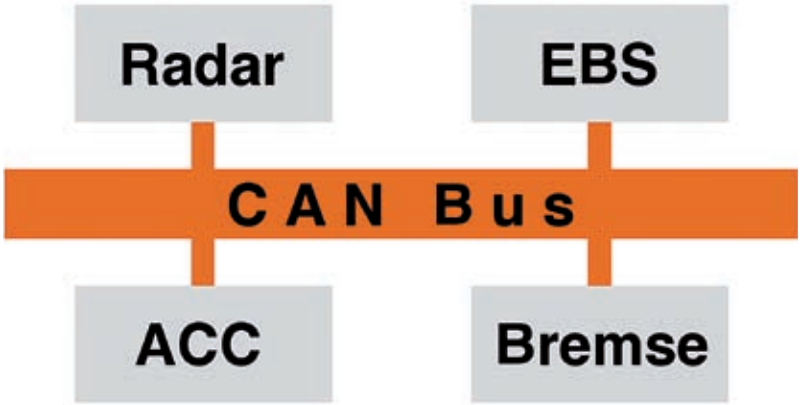

Bild 2: Aufbau eines Notfallbremsassistenten

Entwickler kann eine formale Beschreibung verschieden auslegen. Schon allein diese Klarheit erhöht die Produktivität des Softwareentwicklungsprozesses entscheidend. Die Studien [1] und [2] belegen den Nutzen von formalen Spezifikationen in der Entwicklung von Software. Die Überführung einer Idee in ein konkretes Programm beinhaltet implizit immer den Schritt der Formalisierung in einer eindeutigen Sprache: der Programmiersprache. Die Semantik einer Programmiersprache wird allerdings leider oft nicht explizit, dass heißt formal, gegeben – sondern sie liegt meistens nichtformal vor, also implizit mit der Implementierung des verwendeten Compilers und der entsprechenden Hardware. Den Schritt der Formalisierung, explizit zu machen, und möglichst früh im Entwicklungsprozess zu gehen, bietet einen Produktivitätsgewinn.

Eine Vielzahl von formalen Methoden zur Spezifikation und Analyse von Software stehen zur Verfügung. Jeder dieser Ansätze besitzt Vor- und Nachteile, die seinen Einsatzbereich bestimmen. Ein einziger Lösungsansatz für alle formalen Probleme der Softwareentwicklung existiert genauso wenig wie die Lösung aller Probleme der Softwareentwicklung mit formalen Methoden allgemein. Einige Bestandteile aus der Menge der formalen Methoden seien im Folgenden aufgeführt. Diese Liste erhebt allerdings keinen Anspruch auf Vollständigkeit.

Jeder formale Kalkül kann als Grundlage einer formalen Spezifikation verwendet werden. Dementsprechend existiert eine Reihe von formalen Spezifikationssprachen, zum Beispiel Sprachen basierend auf der Mengentheorie und Logik [3], [4], [5], wobei Logiken wie Aussagen-, Prädikatenlogik erster Stufe oder verschiedene temporale Logiken (LTL, CTL oder μ-Kalkül) verwendet werden. Des weiteren werden Petrinetze [6], verschiedene Automatenmodelle wie endlichen Automaten, Kellerautomaten oder Zeitautomaten (siehe unten), Prozessalgebren wie CSP [7] oder CCS [8] und andere Formalismen genutzt.

Zur formalen Analyse eines Systems muss eine formale Spezifikation zugrunde liegen. Nur so kann dieses mit geeigneten Verfahren wie zum Beispiel abstrakter Interpretation, (Bi-)Simulation, Model-Finding (beispielsweise durch SAT-Solving) oder Model-Checking [9] untersucht werden. Insbesondere der Model-Checking-Ansatz bietet die Möglichkeit, vollautomatisch Eigenschaften des untersuchten Systems nachzuweisen. Der Erfolg des Model-Checkings in der Hard- und Softwareindustrie wurde kürzlich von der ACM mit der Vergabe des Turing-Awards an die Gründungsväter des Model-Checkings, Clarke, Emmerson und Sifakis, anerkannt. Der folgende Abschnitt widmet sich einer konkreten Fallstudie aus dem Bereich der Softwareentwicklung für automobile Anwendungen und soll als Beispiel für den Einsatz eines formalen Verfahrens dienen.

3 Eine Fallstudie

Eine Reihe von Fallstudien zum Einsatz von formalen Methoden in der Entwicklung komplexer Systeme wurden bereits durchgeführt und veröffentlicht [10] bis [14]. Die daraus entstandenen Artikel thematisieren Analysen eingebetteter Systeme in der Automobilindustrie – unter anderem die Steuerung einer Gangschaltung, eine adaptive Geschwindigkeitsregelung oder den CAN-Bus. Die vorgestellte Fallstudie [15] unterscheidet sich von den anderen genannten dadurch, dass hier der Einsatz eines Model-Checkers im Entwurfsprozess eines relativ großen Systems untersucht – und mit seinen Vorteilen und Grenzen dargestellt – wird. Insbesondere benutzen wir den Model-Checker UPPAAL [16].

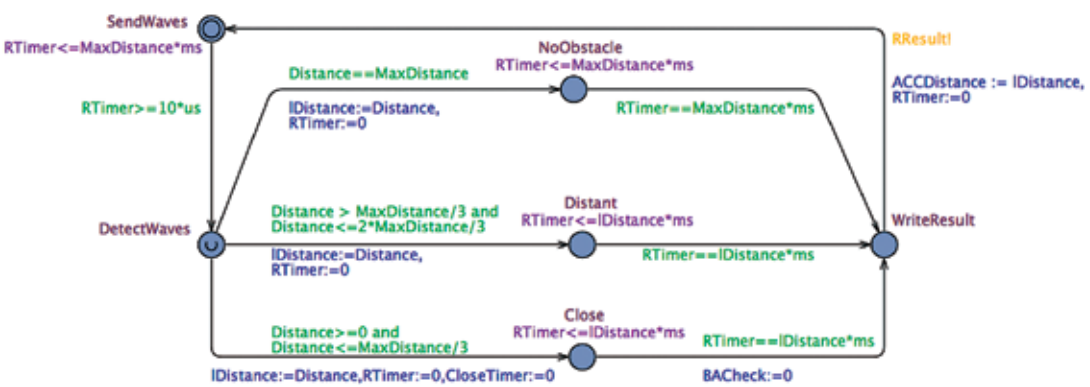

Bild 3: Modell einer Radareinheit in UPPAAL

Das modellierte System ist ein Notfallbremsassistent (EBA), **Bild 1**, der aus vier eingebetteten Steuereinheiten (ECU) besteht: eine Radareinheit zur Abstandsmessung vorausfahrender Fahrzeuge, eine adaptive Geschwindigkeitssteuerung (ACC), ein Notfallbremssystem (EBS) und eine Bremseinheit zur Berechnung des notwendigen Bremsdrucks. Diese sind über den CAN-Bus miteinander verbunden. **Bild 2** veranschaulicht den Aufbau des Systems.

Jede dieser Steuereinheiten und der CAN-Bus statten die Ingenieure mit konkreten Zeitschranken für die notwendigen Berechnungen und Signalübertragungen aus, zum Beispiel soll die ACC alle 5 ms in höchstens 3 ms die Berechnung einer Geschwindigkeitsanpassung durchführen und über den CAN-Bus senden.

Als Formalismus zur Modellierung des EBA wurde die Theorie der Zeitautomaten [17] gewählt. Zum einen eignen sich Zeitautomaten zur Darstellung von Echtzeitsystemen, und zum anderen finden sie eine entsprechende Unterstützung in Form des Werkzeugs UPPAAL. UPPAAL bietet ein graphisches Frontend zur Modellierung von Zeitautomaten und vor allem einen Model-Checker, mit dem sich das vorliegende Modell analysieren lässt. **Bild 3** zeigt das Beispiel eines möglichen Radarmodells in UPPAAL.

Bei dem Model-Checker handelt es sich um ein Werkzeug, das automatisch prüft, ob ein Modell bestimmte Eigenschaften besitzt. Das Modell muss formal – in vielen Anwendungen als Automat – gegeben sein. In der vorliegenden Fallstudie als Zeitautomat. Die untersuchte Eigenschaft muss, ebenfalls formal, als Menge von Systemzuständen oder -abläufen zur Verfügung stehen. Im Allgemeinen werden Eigenschaften als Formeln in einer geeigneten Logik oder selber als Automaten spezifiziert. In UPPAAL kommt zur Spezifikation von Eigenschaften ein Fragment der Berechnungsbaum-Zeitlogik (timed computation tree logic; TCTL) [18] zum Einsatz. Die folgenden zwei Formeln dienen als Beispiele für die Formulierung von Eigenschaften im UPPAAL-TCTL-Fragment: „A[](not deadlock)“ formalisiert die Deadlockfreiheit des Systems. „Radar.Close –> (Brake.EmergencyBrake and (CloseTimer <= 30000))“ formalisiert die Eigenschaft für den Fall, dass ein Objekt vom Radar als nah (Radar.Close) erkannt wird. Innerhalb von 30 ms (CloseTimer <= 30000) würde dann eine Notbremsung eingeleitet (Brake.EmergencyBrake).

Das Model-Checking-Verfahren prüft nicht nur, ob ein gegebenes Modell eine Formel (Eigenschaft) erfüllt oder nicht, sondern gibt, wenn die Eigenschaft nicht erfüllt wird, einen Berechnungspfad zurück, der zur Verletzung der Eigenschaft führt. Durch die Eingabe geeigneter Formeln lassen sich so zum Beispiel genaue Zeitschranken für Abläufe im Modell berechnen, die wiederum in den weiteren Entwurfsprozess einfließen können.

In der vorliegenden Fallstudie verifiziert die Universität Stuttgart und die Daimer AG zahlreiche Sicherheitseigenschaften. Unter anderem stellten die Mitarbeiter eine mögliche Überschreitung einer Zeitschranke fest. Die Ursache lag in der falschen Priorisierung der Kommunikation der verwendeten Komponenten – ein eigentlich trivialer Grund, der aber erst im Verifikationsprozess erkannt wurde. Automatische Verifikationsverfahren weisen im Allgemeinen eine hohe Berechnungskomplexität auf: bei UPPAAL führt das Model-Checking-Problem – von TCTL-Formeln auf Zeitautomaten zu erhöhtem Aufwand (PSPACE vollständig). Die vorgestellte Fallstudie zeigt aber, dass trotz hoher Komplexität der Einsatz solcher Verfahren in einer frühen Entwurfsphase sinnvoll sein kann. Entwurfsfehler oder Inkonsistenzen können frühzeitig erkannt werden. Allein das rechtfertigt den Einsatz formaler Methoden.

4 Fazit

Formale Methoden haben einen Entwicklungsgrad erreicht, der ihren Einsatz in der Softwareentwicklung außerhalb der klassischen Anwendungsfelder, wie der Entwicklung von Hardware und Systeme der Luft- und Raumfahrt, nahelegt. Sie sind kein Ersatz für herkömmliche Entwicklungsmethoden, aber sie bieten vielfältige Ansätze und Einsatzmöglichkeiten zur besseren Beherrschung komplexer Systeme. Gerade die Automobilindustrie kann dabei vom Einsatz formaler Methoden profitieren.

Literaturhinweise

[1] Brookes, T. M.; Fitzgerald, J. S.; Larsen, P. G.: Formal and Informal Specifications of a Secure System Component: Final Results in a Comparative Study. In: 3rd International Symposium of Formal Methods Europe, Industrial Benefit and Advances in Formal Methods (1996), Springer Verlag, Lecture Notes in Computer Science, Bd. 1051, S. 214 – 227

[2] Sobel, A. E. K.; Clarkson, M. R.: Formal Methods Application: An Empirical Tale of Software Development: IEEE Transactions on Software Engineering (2002), Bd. 28, Nr. 3, S. 308 – 320

[3] Spivey, J. M.: An introduction to Z and formal specifications: IEE/BCS Software Engineering Journal (1989), Bd. 4,Nr. 1, S. 40 – 50

[4] Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, 1996

[5] Jackson, D.: Software Abstractions: Resources and Additional Materials. MIT Press, 2006

[6] Reisig, W: Petrinetze – Eine Einführung. Springer Verlag, 1990

[7] Hoare, C. A. R.: Communicating Sequential Processes. Prentice Hall International, 1985

[8] Milner, R: A Calculus of Communicating Systems. Springer Verlag, 1980

[9] Clarke, E. M.; Grumberg, O.; Peled, D. A.: Model Checking. MIT Press, 1999

[10] Lindahl, M.; Pettersson, P.; Yi, W.: Formal Design and Analysis of a Gear Controller. In: International Journal on Software Tools for Technology Transfer (2001), Bd. 3, Nr. 3, S. 353 – 368

[11] Hansson, H.; Åkerholm, M.; Crnkovic, I.; Törngren, M.: SaveCCM – A Component Model for Safety-Critical Real-Time Systems. In: Euromicro Conference, Special Session Component Models for Dependable Systems, Rennes, Frankreich (2004), IEEE

[12] Tindell, K.; Burns, A.: Guaranteed Message Latencies for Distributed Safety-Critical Hard Real-Time Control Networks. University of York, YCS 229, 1994

[13] Van Osch, M.; Smolka, S. A.:Finite-State Analysis of the CAN Bus Protocol. In: The 6th IEEE International Symposium on High-Assurance Systems Engineering (2001), IEEE, S. 42 – 54

[14] Krákora, J.; Hanzálek, Z.: Timed Automata Approach to Real Time Distributed System Verification. In: 5th IEEE International Workshop on Factory Communication Systems (2004), IEEE

[15] Montag, P.; Nowotka, D.; Levi, P.: Verification in the Design Process of Large Real-Time Systems: A Case Study. In: Automotive - Safety & Security 2006 - Sicherheit und Zuverlässigkeit für automobile Informationstechnik, Stuttgart (2006), Shaker Verlag

[16] UPPAAL: http://www.uppaal.com

[17] Alur, R.; Dill, D.: A Theory of Timed Automata: Theoretical Computer Science (1994), Bd. 126, Nr. 2, S. 183 – 235

[18] Alur, R.; Courcoubetis, C.; Dill, D.: Model-Checking in Dense Real-Time: Information and Computation (1993), Bd. 104, Nr. 1, S. 2 – 34