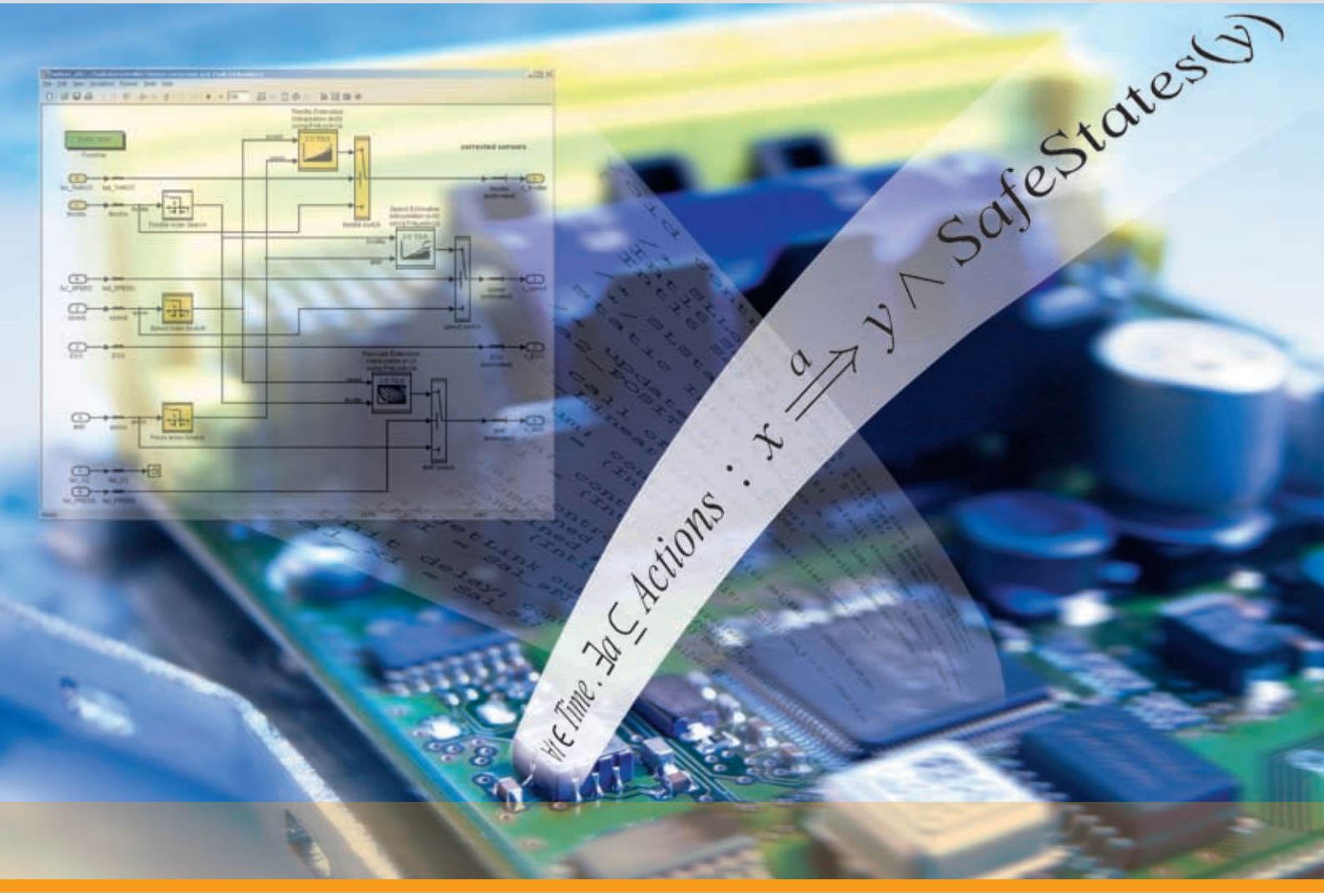


# Formal Methods in Software engineering

## Mathematically reasoning complements traditional methods

Traditional, informal techniques in software engineering reach their limits when complex systems are to be developed efficiently. The Institute for Formal Methods in Computer Science of the Universität Stuttgart states: Tools like IDEs, automatic code generation from graphical descriptions, visualization techniques like UML diagrams, and others are certainly necessary. However, they often lack the ability to support detailed and provably sound reasoning about complex systems. Formal methods in software engineering provide approaches to close that gap.

## 1 Introduction

The increased use of electronics in cars leads to a continuously growing portion of software engineering costs of the total automobile development costs. The expenditure of developing software, e.g., design and implementation, modifications, tests, and maintenance of software, are only part of the overall costs. Hard to estimated but certainly high excess costs arrise when software causes malfunction of some car component or even the whole car. The employment of formal methods in software engineering can substantially contribute to the reduction of such costs.

The term "Formal Methods" comprises a variety of approaches, methodologies, and techniques based on mathematics and logics. Section 2 considers formal methods in general – without pretens of completeness. An exemplary application of formal methods in the development process of a larger real-time system is sketched in Section 3. The Universität Stuttgart and the Daimler AG used the model-checking approach of the temporal logic TCTL on timed automata to find flaws early in the design process of an adaptive brake assistant.

## 2 Formal Methods

First experiences in the use of formal methods for verifying software had been gathered already in the 1950s and 60s. However, the initial exuberant and unrealistic expectations soon turned out to be futile. The illusion of gaining an "absolute security" and a "totally correct program" led to frustrations. The formalization of intended properties of a program alone is a process which can neither be formally conducted nor verified. Only when a formal specification exists further development steps of the software can use formal methods.

A further misconception exists in the assumption that formal methods can only be used for program verification. Of course, the formal verification of a program against its specification is a classical application of formal methods. Another big advantage of a formal specification over informal ones is the unambiguity of statements. When a formal description of a system exists, one can not only precisely reason about and prove properties, but it also constitutes communication tool beneficial to everyone involved in the design process. Neither the customer nor the developer can interpret a formal description differently. This unambiguity alone increases the productivity of software development. Case studies, like [1] and [2] support that claim about the use of a formal specification in the software development process. The transfer of an idea into a particular program always implicitly contains the step of formalization into an unambiguous language: the programming language. Unfortunately, the semantics of a programming language is often not explic-

**Figure 1:** An emergency brake assistant like investigated in a case study

## The Author

**Dr. Dirk Nowotka** works at the Institute for Formal Methods in Computer Science at the Universität Stuttgart.

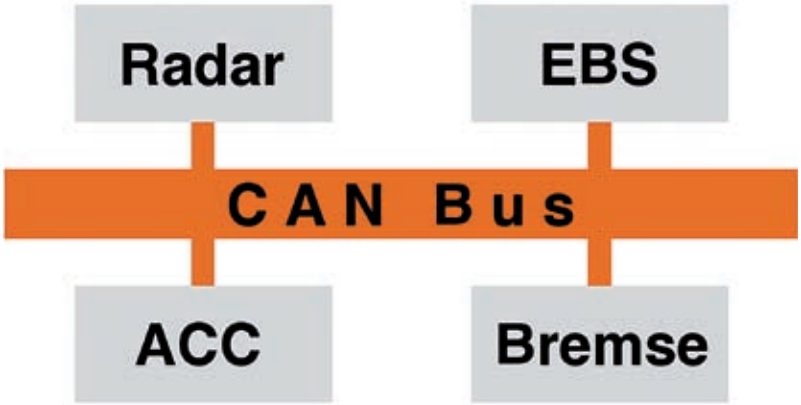


**Figure 2:** Structure of an emergency brake assistant

itly, that is, formally, given – rather it is defined by the particular implementation of the language by a compiler and the underlying hardware. Productivity gains a lot when this formalization step is made explicit and taken early in the design process.

A large variety of formal methods for the specification and analysis of software exists. Every such method has its advantages and disadvantages which determine its area of employment. There is no single solution for all formal problems as much as formal methods are not the single solution for all problems in software development. Some parts of formal methods are named below. However, that list is certainly not complete.

Every formal calculus can form the base of a formal specification. Consequently, a number of formal specification languages exist, for example, languages base don set theory and logic [3], [4], [5], where logics like propositional logic, first-order logic or various modal logics (LTL, CTL, or μ-calculus) are used. Moreover, Petri nets [6], various automata models, like finite automata, push-down automata or timed automata (see further below), process algebras, like CSP [7] or CCS [8], and other formalisms are employed.

Once a formal specification is written, it can be investigated with suitable methods, like for example, abstract interpretation, (bi-)simulation, model-finding (using a SAT solver), or model-checking [9]. In particular, the model-checking approach allows an efficient and fully automatic verification of properties of the investigated system. The success of model-checking in the hard- and software industry has recently been recognized by the ACM in giving the Turing-Award to the founding fathers of model-checking Clarke, Emmerson, and Sifakis. The following section is devoted to a case study in the area of software development for automobile applications and shall illustrate the use of a formal method therein.

## 3 A Case Study

A range of case studies about the employment of formal methods in the development of complex systems have been published; see [10] through [14]. These papers investigate the analysis of embedded systems in the automotive industry – among others, a gear shift controller, and adaptive cruise controller or the CAN bus. The presented case study [15] differs from others by the investigation of the employment of a model-checker in the design process of a rather large system – illustrating its advantages and limits. To be more precise, we used the UPPAAL model-checker [16].

The modeled system is an emergency brake assistant (EBA), **Figure 1**, that consists of four embedded control units (ECU): a radar unit for measuring the distance to objects in front, an adaptive cruise controller (ACC), an emergency brake system (EBS), and a brake unit for calculating the necessary brake pressure. These units are connected by the CAN bus. **Figure 2** illustrates the structure of the system.

Each of the control units and the CAN bus is augmented with precise time bounds for the necessary calculations and signal transmission. For example, the ACC is to calculate and send a speed

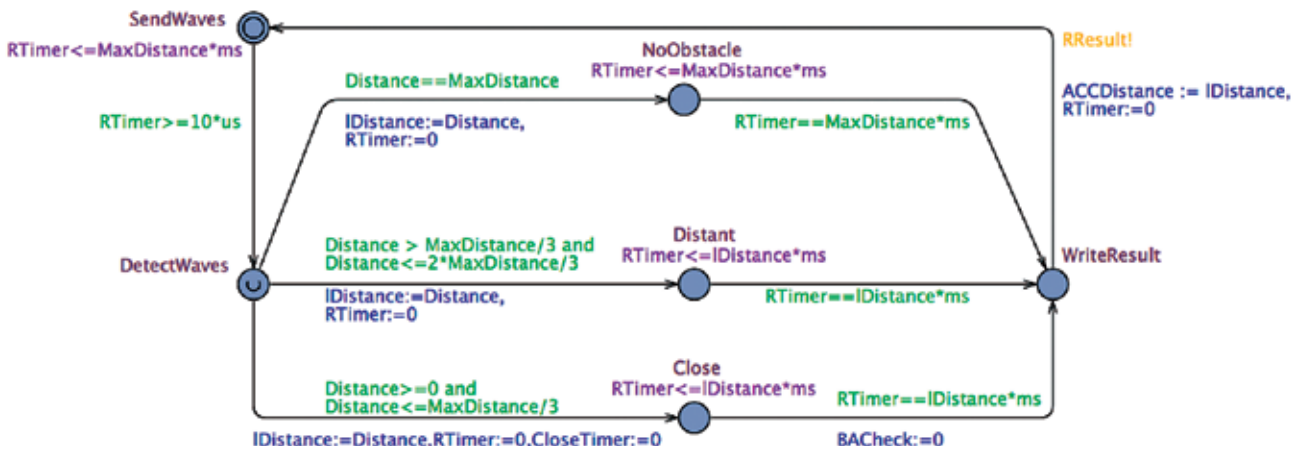


**Figure 3:** Model of a radar unit in UPPAAL

correction command every 5 ms using at most 3 ms.

The theory of timed automata [17] is used for formalizing the system. Firstly, timed automata are well-suited for describing real-time systems, and secondly, there exists tool support in, for example, UPPAAL. UPPAAL provides a graphical front-end for modeling timed automata and a model-checker for analyzing a model. **Figure 3** shows an example of a possible model of the radar unit in UPPAAL.

A model-checker is a tool that automatically checks whether or not a given model possesses a given property. The model has to be formally given – in many applications as an automaton. In the present case study timed automata are used. The investigated property also has to be given, formally, as a set of system states or runs. In general, properties are formulated in a suitable logic or as automata themselves. A fragment of the timed computation tree logic (TCTL) [18] is used in UPPAAL for specifying properties. The following two formulas shall serve as formalization examples in the TCTL fragment of UPPAAL: "A[](not deadlock)" formulates that deadlock freeness of the system. "Radar.Close –> (Brake. EmergencyBrake and (CloseTimer <= 30000))" formulates the property that in case an object is recognized as being close (Radar.Close) an emergency brake (Brake.EmergencyBrake) is initiated within 30 ms (CloseTimer <= 30000).

The model-checking procedure not just checks whether or not a given model satisfies a given formula, i.e., property, but also gives a counter-example when the property does not hold, that is, it provides a calculation path that leads to the violation of the property. This can be used to gain precise timing contraints by entering suitable formulas. Such time bounds can be fed back to the design process.

The Universität Stuttgart and the Daimler AG verified several security properties in the presented case study. For example, a possible violation of a time bound was diagnosed. The cause of which was found to be in the wrong assignment of communication priorities for the used components. Automatized verification methods possess a very high asymptotic computational complexity: in UPPAALs case we have that the model-checking problem of TCTL on timed automata is PSPACE-hard. The presented case study shows that, nevertheless, a method of such high computational complexity can be useful in the early phase of development. Design flaws and inconsistencies can be detected early. This alone justifies the employment of formal methods.

## 4 Conclusion

Formal methods have reached a degree of maturity that suggests their use outside classical application areas, like the design of hardware or space- and aviation systems. They are not a substitute for traditional methods, but they provide various approaches and opportunities to handle complex systems. In particular, the automotive industry can benefit from formal methods.

## References


[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, vol. 1051, pp. 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), vol. 28, nr. 3, pp. 308 – 320

[3] Spivey, J. M.: An introduction to Z and formal specifications: IEE/BCS Software Engineering Journal (1989), vol. 4, nr. 1, pp. 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), vol. 3, nr. 3, pp. 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, pp. 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), vol. 126, nr. 2, pp. 183 – 235

[18] Alur, R.; Courcoubetis, C.; Dill, D.: Model-Checking in Dense Real-Time: Information and Computation (1993), vol. 104, nr. 1, pp. 2 – 34 ((Bosch Firmenbezeichnung und Überschrift aus Figure 1 löschen, Danke))