Formal techniques for software and hardware verification

LECTURES:

Vladimir Anatolyevich Zakharov,
Vladislav Vasilyevich Podymov

http://mk.cs.msu.ru/

PROGRAM OF THE COURSE

1. Introduction. Hardware and software verification problems

PROGRAM OF THE COURSE

1. Introduction. Hardware and software verification problems
2. Deductive approach to software verification

PROGRAM OF THE COURSE

1. Introduction. Hardware and software verification problems
2. Deductive approach to software verification
3. Formal models of concurrent programs and circuits

PROGRAM OF THE COURSE

1. Introduction. Hardware and software verification problems
2. Deductive approach to software verification
3. Formal models of concurrent programs and circuits
4. Temporal logics

PROGRAM OF THE COURSE

1. Introduction. Hardware and software verification problems
2. Deductive approach to software verification
3. Formal models of concurrent programs and circuits
4. Temporal logics
5. Model checking for Computational Tree Logic CTL: enumerative and symbolic approaches

PROGRAM OF THE COURSE

1. Introduction. Hardware and software verification problems
2. Deductive approach to software verification
3. Formal models of concurrent programs and circuits
4. Temporal logics
5. Model checking for Computational Tree Logic CTL: enumerative and symbolic approaches
6. Model checking for Linear Temporal Logic LTL: automata-theoretic approach

PROGRAM OF THE COURSE

7. Efficiency improvement methods: abstraction, simulation, bounded model checking

PROGRAM OF THE COURSE

7. Efficiency improvement methods: abstraction, simulation, bounded model checking
8. Formal models of real-time computing systems

PROGRAM OF THE COURSE

7. Efficiency improvement methods: abstraction, simulation, bounded model checking
8. Formal models of real-time computing systems
9. Model checking of real-time computing systems

PROGRAM OF THE COURSE

The slides of the lectures are be loaded on the web-page

http://mk.cs.msu.ru

Literature

- Э.М. Кларк, О. Грамберг, Д. Пелед. Верификация моделей программ: Model Checking. Изд-во МЦНМО, 2002.
- Ю.Г. Карпов. Model Checking: верификация параллельных и распределенных программных систем. Изд-во БХВ-Петербург, 2010.
- K. R. Apt, E.-R. Olderog. Verification of sequential and concurrent programs, Springer, 1997.
- B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen. Systems and Software Verification: Model-Checking Techniques and Tools. Springer, 2001.
- **Baier C., Katoen J.-P. Principles of model checking, MIT Press, 2008.**
- **Clarke E., Henzinger T.A., Veith H., Bloem R. Handbook of model checking, Springer, 2018.**

Lecture 1.

Hardware and software verification problems

1. Why do we need formal verification of information processing systems?
2. Basic approaches to formal verification of hardware and software systems.
3. Principles of model checking.
4. Historical overview
5. Achievements of formal verification techniques.

Why do we need formal verification of information processing systems?

It is customary to regard the tasks of program specification and verification as a kind of auxiliary activity, as an inevitable "headache" which accompanies the main activity of a software engineer - the development of program code.

However, in the early 80s of the XX century, software developers discovered that more than half of the costs associated with the manufacturing of a software product or microcircuit fall precisely at the stage of verification, debugging, and elimination of errors, and this share is steadily increasing.

Among the most significant reasons explaining this phenomenon are the following.

Why do we need formal verification of information processing systems?

1. Increased cost of damage due to a missed program error

Hardware and software systems are widely used in many applications where any failure is considered unacceptable such as

- weapon control,
- power stations and energy systems,
- banking and e-commerce,
- telecommunications,
- transport control,
- medical equipment,

and this list is quite long.

Why do we need formal verification of information processing systems?

Example 1.

On July 22, 1962, the Mariner-1 spacecraft was launched. Soon after the start the antenna of the rocket lost contact with the guidance system on Earth and, therefore, the on-board computer took over the flight control. But it turned out that some procedure in the flight control program contained an error. As a result, the navigation system lost orientation, and the spacecraft was detonated 293 seconds after the launch.

A NASA report sent to Congress in 1963 mentions: «The omission of the hyphen in the data edit caused the computer to issue a series of unnecessary course correction signals that knocked the ship off course and led to its destruction.»

Why do we need formal verification of information processing systems?

Example 2.

On September 2 September 1988, the wrong command was sent to the Soviet interplanetary station Phobos-1. When processing an exceptional situation and launching an interrupt handling, the on-board computer interpreted this command as disabling the stabilization and orientation system.

As a result, the station got out of control and got lost in space.

Why do we need formal verification of information processing systems?

Example 3.

On June 4 1996 European space rocket Ariane-5 exploded less than forty seconds after the launch.
This accident was due to an error in the program of the computer responsible for the trajectory calculations. During startup, an exception was thrown when a large 64-bit floating point number has been converted to 16-bit integer. This transformation was not protected by an exception handling code, which led to a computer failure. This same error was reproduced by duplicate computer.
As a result, incorrect altitude data was transmitted to the on-board computer, causing the missile to be destroyed.

Damage cost — 370 000 000 $.

Why do we need formal verification of information processing systems?

Example 4.

In 1999, a NASA subcontractor supplied software that used imperial units (inches) while the instrumentation was calibrated to metric (centimeters).

This caused the crash of the satellite sent to Mars.

Damage cost — 327 000 000 $.

Why do we need formal verification of information processing systems?

Example 5.

On January 15 January 2005, the Huygens space probe separated from the automatic interplanetary station Cassini which after a 7-year flight reached the orbit of Titan — the satellite of Saturn — and began to descend into the atmosphere of the satellite and transmit information about the atmosphere and surface of Titan. This is the only landing accomplished in the outer Solar System.

An error in the on-board program (data transfer procedure) of the probe led to the fact that only 350 pictures were received instead of the 700 planned.

Probe development costs amounted to 2,000,000,000 $.

Why do we need formal verification of information processing systems?

Example 6.

Medical device for radiation therapy Therac-25 was involved in at least six accidents between 1985 and 1987.

Because of numerous errors, it sometimes gave its patients radiation doses that were 100 times greater than normal, resulting in death or serious injury.

Example of an error: the software set a flag variable by incrementing it, rather than by setting it to a fixed value. Occasionally an arithmetic overflow occurred, causing the flag to return to zero and the software to bypass safety checks.

These accidents highlighted the dangers of software control of safety-critical systems, and they have become a standard case study in health informatics and software engineering.

Why do we need formal verification of information processing systems?

Example 7.

In 2000, in Panama, several patients overdosed on radiation from radiation therapy. It was found that the control program set the radiation power depending on the order in which some data were received through the parallel channels (the "race" condition effect).

Overdose has caused the death of at least five patients.

Why do we need formal verification of information processing systems?

Example 8.

The software of the control system of the MIM-104 Patriot anti-aircraft missile system contained an error, due to which the timers in different computers within this complex tended to desynchronize 0.3 seconds per 100 hours of continuous operation.

As a result, on February 25, 1991, such a complex was unable to intercept an Iraqi tactical missile, which led to the death of 28 US military personnel.

Why do we need formal verification of information processing systems?

Example 9.

On March 23, 2003, the control system of the MIM-104 Patriot anti-aircraft missile system mistakenly identified the British Tornado bomber as an approaching enemy missile and sent a command to launch the missile.

As a result, the British plane was shot down; both of his pilots were killed.

During Operation Desert Storm, 24 % of casualties occurred due to "friendly fire" .

The conclusions of numerous independent commissions were unambiguous — the main reason was errors in the software and hardware of automatic weapons control systems.

Why do we need formal verification of information processing systems?

Example 10.

On December 20, 1995, 159 people were killed in the crash of a Boeing 757 (flight Miami - Cali).

Investigation revealed that the cause of the disaster was a mistake in one symbol of the flight control software.

Boeing paid over $ 300 million to the relatives of the victims of this plane crash.

Why do we need formal verification of information processing systems?

Example 11.

In 1998, in the United States, the commissioning of a new air transport service system was delayed for half a year due to incessant failures in the software of this system.

According to experts, the damage amounted to several hundred million dollars.

Why do we need formal verification of information processing systems?

Example 12.

On August 1, 2012 the financial company Knight Capital Group lost $ 440 million in 45 minutes.

There had been substantial code refactorings in trading software over the years without thorough regression testing.

Once an updated version of the software was incorrectly installed on the servers, which caused the old and new code to work alternately for some time.

Why do we need formal verification of information processing systems?

Example 13.

In 2009, the Japanese division of the Swiss bank UBS nearly spent $ 31,000,000,000 on the purchase of Capcom bonds. The purchase was stopped at the last minute.

The investigation showed that an employee of the company gave a command to buy bonds in the amount of $ 310,000, but due to a system error in the software, 5 zeros were added to the order.

Why do we need formal verification of information processing systems?

Example 14.

In 1994, the development of Intel Pentium processors missed a bug in the built-in division program — several elements of the auxiliary array were not initialized.

The error was only discovered after the processors went on sale; the company was forced to replace all defective processors.

The damage is estimated at hundreds of millions of dollars.

Why do we need formal verification of information processing systems?

Example 15.

Statistical studies show that more than 10% of projects for the development of microelectronic circuits are never brought to serial production, due to significant errors that are discovered during the exploitation of prototypes. The cost of developing a circuit and making its prototype can be several hundred thousand dollars.

Research carried out in 2002 by the National Institute of Standards and Technology showed that the economic losses due to software errors are estimated at $ 59.5 billion, and this damage can be reduced by a third through improved methods of software verification.

Why do we need formal verification of information processing systems?

2. Because the field of application of computer programs is rapidly expanding

As the participation of software systems in our lives increases, so does the burden of responsibility. for the correctness of their functioning. Security already cannot be restored by simply disabling the malfunctioning system: disconnecting the device is even more dangerous. We depend on computing devices to function properly.

Therefore, the development of methods helping to increase our confidence in the correctness such information processing systems becomes even more urgent.
The very understanding of the verification task is changing: a program can be recognized as reliable not because errors were not found in it, but because the absence of errors was **proven** .

Why do we need formal verification of information processing systems?

3. Software systems are getting incredibly complex

OS Microsoft Windows 3.1 (1992 г.) contains 4 million lines of code.

OS Microsoft Windows 98 (1998 г.) contains 18 million lines of code.

OS Microsoft Windows XP (2002 г.) contains 40 million lines of code.

Google Services (2015 г.) contain more than 2000 million lines of code.

Why do we need formal verification of information processing systems?

3. Software systems are getting incredibly complex

The development of programs (and even more so for microelectronic circuits) "becomes a multi-layer" process; at different stages of this process the program model or microelectronic circuit is refined, which ends with their implementation. Errors missed in the early stages cannot be eliminated at the design stages.

Therefore, a variety of verification methods and tools is needed that could be applied not only to the programs (sircuits) themselves, but also to their models, presented at different levels of abstraction.

Why do we need formal verification of information processing systems?

4. Verification problems are getting harder

Computing hardware is getting cheaper and the performance of computers is increasing.

New opportunities are emerging for the application of computers for the solution of more and more complex and responsible task.

The size of program code is increasing.

As a result, the probability of introducing errors into such codes both at the design stage of the program and during the implementation of the project is increasing as well.

Why do we need formal verification of information processing systems?

4. Verification gets harder

Modern software design and development technologies considerably facilitate the work of software engineer and improve the productivity of their labor, and it grows as quickly as the volume of software increases.

However, program verification runs the risk of remaining an extensive labor requiring a large number of skilled workers exclusively engaged in searching for errors in the program code. But even this method does not allow us to be sure of the reliability of the software product.

The main trends in program verification

The most simple and straightforward methods for checking the correctness of programs and microelectronic circuits — simulation and testing .

Simulation deals with abstract schematics and software prototype, while **testing** is applied to final software or hardware product.
In the case of electronic circuits, for example, the design of the developed circuit is subjected to simulation, while the microcircuit itself is tested. In both cases in these methods, certain signals are usually entered in given points of the circuit, and at other points take appropriate readings.

These methods can be very efficient for identifying many errors. But to check correctness of **ALL** possible interactions and to cover **ALL** conceivable runs by applying only modeling and testing is unlikely to succeed.

The main trends in program verification

Formal methods are intended for **proving** that a system under construction **HAS NO** certain errors.

The main approaches to formal verification of information processing systems (software programs and microelectronic circuits) are:

1. **equivalence checking** ,
2. **proof-theoretic approach** (deductive analysis),
3. **model checking** (verification of program models),
4. **runtime verification** (verification of computation traces).

The main trends in program verification

Equivalence checking

Every information processing system is characterized by its

- control scheme Σ (syntax),
- computing function F_Σ (semantics).

Equivalence checking problem : for every given pair of schemes Σ_1 and Σ_2 check that

$$F_{\Sigma_1} = F_{\Sigma_2}$$

holds.

Equivalence checking

When designing a System-On-a-Chip (SOC) , it is represented at different levels of abstraction. The Synopsis design route includes three main levels:

1. System level :
   - Stage 1: designing and verification of algorithmic model of SOC;
   - Stage 2: designing and verification of transaction model of SOC;
   - Stage 3: designing of so called "golden model" of SOC as technical specifications for the design of the circuit.

Equivalence checking

2. Logical , or gate , level, includes 5 stages:
   - Stage 1: designing of RTL (Register-Transfer Level) model of SOC in one of the hardware description languages (Verilog, VHDL);
   - Stage 2: logical specification of RTL-model by means of simulation;
   - Stage 3: physical verification through prototyping;
   - Stage 4: synthesis of SOC in the basis of gate circuits and automatic generation of test structures for validation control;
   - Stage 5: formal verification of netlists.

3. Topological level

Equivalence checking

Thus, during the design of VLSI, 5 descriptions of the system are consistently built:

- Algorithmic specification of the system (на языке C/C++);
- Macro-architectural description of the system (in System-C);
- RTL-specification of the system (in VHDL or Verilog);
- Netlists (in VHDL or Verilog in IEEE PDEF, DEF, LEF formats);
- VLSI topology description (GDSII format).

The basic principle of multi-level design:

functionality at a higher level of abstraction should be preserved at a lower level.

Equivalence checking

The following verification problem arises.

	Logical circuit Σ
Log2Top :	$\Downarrow$
	Topological implementation of the circuit T ,
	Topological implementation of the circuit T
Mod :	$\Downarrow$
	Improved topological implementation of the circuit T'
	Topological implementation of the circuit T'
Top2Log :	$\Downarrow$
	Logical circuit Σ' .

The problem: check that $F_\Sigma = F_{\Sigma'}$ holds.

For logic circuits, this problem is reduced to the satisfiability problem for Boolean formulas SAT.

Proof-theoretic approach (deductive analysis)

Proof-theoretic approach: axioms and inference rules are applied to prove the correctness requirements for system under development.

In the early stages of deductive analysis research, the focus was on ensuring the correct operation of critical systems, and such evidence was built exclusively by hand.

Over time, tools (provers) have appeared that make it possible to apply systematic search for various directions in constructing the correctness proof.

Proof-theoretic approach (deductive analysis)

The main principles of proof-theoretic approach are as follows:

1. Every program (circuit) π computes some input-output relation R_π ;

Proof-theoretic approach (deductive analysis)

The main principles of proof-theoretic approach are as follows:

1. Every program (circuit) π computes some input-output relation R_π ;
2. A code of a program (a description of a circuit) is formal description of the relation R_π in some programming language;

Proof-theoretic approach (deductive analysis)

The main principles of proof-theoretic approach are as follows:

1. Every program (circuit) π computes some input-output relation R_π ;
2. A code of a program (a description of a circuit) is formal description of the relation R_π in some programming language;
3. Correctness requirement (specification) Φ for a program is a description of an input-output relation to be implemented by a program (a circuit), but this specification is presented in some other language (or some other form);

Proof-theoretic approach (deductive analysis)

The main principles of proof-theoretic approach are as follows:

1. Every program (circuit) π computes some input-output relation R_π ;
2. A code of a program (a description of a circuit) is formal description of the relation R_π in some programming language;
3. Correctness requirement (specification) Φ for a program is a description of an input-output relation to be implemented by a program (a circuit), but this specification is presented in some other language (or some other form);
4. To check the correctness of a program π against a specification Φ is to prove that $R_\pi \Rightarrow \Phi$.

Proof-theoretic approach (deductive analysis)

The main principles of proof-theoretic approach are as follows:

1. Every program (circuit) π computes some input-output relation R_π ;
2. A code of a program (a description of a circuit) is formal description of the relation R_π in some programming language;
3. Correctness requirement (specification) Φ for a program is a description of an input-output relation to be implemented by a program (a circuit), but this specification is presented in some other language (or some other form);
4. To check the correctness of a program π against a specification Φ is to prove that $R_\pi \Rightarrow \Phi$.

Usually, a correctness requirement is composed of a pre-condition φ , which specifies the admissible input data, and post-condition ψ , which specifies an input-output relation.

Proof-theoretic approach (deductive analysis)

Deductive analysis had a significant impact on approaches to software development (for example, contributed to the introduction of the concept invariant).

However, this approach is labor-intensive and can only be done by experts, having knowledge of inference and considerable practical experience. Proof of the correctness of the individual protocol or electronic circuit can take days or even months.

Therefore, deductive analysis is used mainly for checking software systems, and usually is bounded to the analysis of software components which are the most susceptible to defects and failures when the necessary resources can be allocated to ensure that the system functions correctly.

Proof-theoretic approach (deductive analysis)

It is also important to realize that there is no algorithms able to recognize whether a arbitrary computer program ever terminates (written in one of the languages programming like C or Pascal). This well known fact puts a limitation on the the possibilities of automatic verification. In in particular, the correctness of a program cannot be proved pure automatically. For this reason, most provers are not fully automatic and operates in interaction with experts.

The advantage of deductive analysis is that it can be applied to systems with infinite number of states. This task can be automated up to certain limits. However, even if a property to be checked is valid, no estimation can be made on time and memory required to find the correctness proof.

Model Checking

Model checking (or verification of models against their specifications) is used mostly for verification of finite state systems. Usually a verification procedure is an exhaustive search in state space of the system.
If sufficient resources are available, this procedure always terminates and can be implemented by a rather efficient algorithm.

Although the finiteness limitation on the number of system states is very important, this approach is applicable to many classes of computing systems: controllers, drivers, communication protocols.

In some cases, systems with an infinite state space can be successfully checked by means model checking supplied with induction and abstraction techniques and refinement of models.

Model Checking

Since model checking techniques can be applied pure automatically, it has an advantage against proof-theoretic approach in those cases where it can be used.

However, some crucial applications will always remain which require deductive analysis for complete verification.

In this regard, a new promising line of verification can be launched: it provides an integration of deductive analysis and model checking in such a way that fragments of the system that have a finite number of states were checked automatically.

Model Checking

Principles of model checking

1. For any given information processing system (program or microelectronic circuit) build a model M , i.e. such a discrete structure which

Model Checking

Principles of model checking

1. For any given information processing system (program or microelectronic circuit) build a model M , i.e. such a discrete structure which
   - may be regarded as an interpretation for some formal logic, and

Model Checking

Principles of model checking

1. For any given information processing system (program or microelectronic circuit) build a model M, i.e. such a discrete structure which
   - may be regarded as an interpretation for some formal logic, and
   - specifies (at some abstraction level) a behaviour of the system under verification.

Model Checking

Principles of model checking

1. For any given information processing system (program or microelectronic circuit) build a model M , i.e. such a discrete structure which
   - may be regarded as an interpretation for some formal logic, and
   - specifies (at some abstraction level) a behaviour of the system under verification.
2. Build a formal specification of correctness requirements to the system as a logical formula φ .

Model Checking

Principles of model checking

1. For any given information processing system (program or microelectronic circuit) build a model M, i.e. such a discrete structure which
   - may be regarded as an interpretation for some formal logic, and
   - specifies (at some abstraction level) a behaviour of the system under verification.
2. Build a formal specification of correctness requirements to the system as a logical formula φ.
3. Check the satisfiability relation of the specification φ on the model M:

$$M \models \varphi$$

Model Checking

The Model Checking technique consists of several stages.

Model Checking

The Model Checking technique consists of several stages.

Modeling

The first task is to bring the designed system to such a form that would be acceptable for model checking tools.

This is often just a compilation task. In others cases (when a system is too much complicated) modeling may require abstraction to get rid of irrelevant details.

Model Checking

Specification

Before applying a verification procedure, it is necessary to formulate the requirements a designed system should comply with.

Typically, specifications are given as logical formulas. For hardware and software, as a rule, they use temporal logic which are rather expressive specification means, allowing to describe how the system behaves in time.

An important issue is completeness of specification. Model checking makes it possible to verify that the model of a designed system meets the specified correctness requirements, however, it is impossible to determine if a given specification covers all properties, which the system should have.

Model Checking

Verification

Ideally, verification is performed completely automatically. To this end, some well known algebraic and combinatorial techniques are used, which include algorithms for

- checking the satisfiability of Boolean formulas;
- checking the reachability of given nodes in finite graphs;
- checking the reachability of strongly connected components in directed graphs;
- building circuits (diagrams) that implement Boolean functions;
- checking the solvability of systems of linear inequalities;
- etc.

Model Checking

Model Refinement

In practice, verification often requires human assistance. One of the aspects of human activity is the analysis of verification results. If model checking results are negative, then the user is often granted a computation trace containing a detected error. It is used as a counterexample for the property being verified and helps the designer to understand where and why the error occurs. In this case, the analysis of the erroneous trace entails system modification and then model checking algorithm is applied once again.

There are methods that automatically refine a model of an analyzed programs in the case when the discovered counterexample (the trace of the supposedly erroneous computation) is a false positive, i.e. can not be implemented in the program.

Runtime Verification

Runtime Verification is a verification technique which occupies an intermediate position between testing and model checking.

It is not the program itself that is checked, but the traces of its computations (execution scenarios, observed behavior), which are generated as a result of test experiments or real program execution.

The correctness requirements are presented formally (say, in tems of regular expressions).

The computation trace is represented as a formal mode (say, as word in a certain alphabet).

Runtime Verification Procedure checks the correspondence between correctness requirements and computation traces (say, by checking if a given word contains in the language which is tha value of the regular expression).

Runtime Verification can be performed on-line or off-line.

Historical Overview

Modal Logics (Aristotel, 400 BC.)

Historical Overview

Modal Logics (Aristotel, 400 BC.)

Axiomatic theories for Modal Logics (C.I. Lewis, 1910)

Historical Overview

Modal Logics (Aristotel, 400 BC.)

Axiomatic theories for Modal Logics (C.I. Lewis, 1910)

Temporal Logics (A.N. Prior, 1957)

Historical Overview

Modal Logics (Aristotel, 400 BC.)

Axiomatic theories for Modal Logics (C.I. Lewis, 1910)

Temporal Logics (A.N. Prior, 1957)

Possible world semantics for Modal Logics(S. Kripke, 1959)

Historical Overview

Modal Logics (Aristotel, 400 BC.)

Axiomatic theories for Modal Logics (C.I. Lewis, 1910)

Temporal Logics (A.N. Prior, 1957)

Possible world semantics for Modal Logics(S. Kripke, 1959)

Using deductive methods to prove the correctness of programs (C.A. Hoare, M. Floyd 1968)

Historical Overview

Modal Logics (Aristotel, 400 BC.)

Axiomatic theories for Modal Logics (C.I. Lewis, 1910)

Temporal Logics (A.N. Prior, 1957)

Possible world semantics for Modal Logics(S. Kripke, 1959)

Using deductive methods to prove the correctness of programs (C.A. Hoare, M. Floyd 1968)

Dynamic Logics (V. Pratt, 1976)

Historical Overview

Modal Logics (Aristotel, 400 BC.)

Axiomatic theories for Modal Logics (C.I. Lewis, 1910)

Temporal Logics (A.N. Prior, 1957)

Possible world semantics for Modal Logics(S. Kripke, 1959)

Using deductive methods to prove the correctness of programs (C.A. Hoare, M. Floyd 1968)

Dynamic Logics (V. Pratt, 1976)

Using Temporal Logics to prove the correctness of message exchange in concurrent systems (A. Pnueli, 1977)

Historical Overview

Tableau model checking algorithm
for Computational Tree Logic CTL
(E. Clarke, E. Emerson, J. Sifakis, 1981)

Historical Overview

Tableau model checking algorithm
for Computational Tree Logic CTL
(E. Clarke, E. Emerson, J. Sifakis, 1981)

Reduction of Model Checking Problem
to Emptiness Problem for Buchi automata
(Systla A.P., Vardi M.Y., Wolper P., 1985)

Historical Overview

Tableau model checking algorithm
for Computational Tree Logic CTL
(E. Clarke, E. Emerson, J. Sifakis, 1981)

Reduction of Model Checking Problem
to Emptiness Problem for Buchi automata
(Systla A.P., Vardi M.Y., Wolper P., 1985)

Symbolic Model Checking,
SMV Verification Tool
(K. McMillan, 1991)

Historical Overview

Tableau model checking algorithm
for Computational Tree Logic CTL
(E. Clarke, E. Emerson, J. Sifakis, 1981)

Reduction of Model Checking Problem
to Emptiness Problem for Buchi automata
(Systla A.P., Vardi M.Y., Wolper P., 1985)

Symbolic Model Checking,
SMV Verification Tool
(K. McMillan, 1991)

Verification Tool SPIN
(Holzmann G.J., 1980-90)

Historical Overview

Tableau model checking algorithm
for Computational Tree Logic CTL
(E. Clarke, E. Emerson, J. Sifakis, 1981)

Reduction of Model Checking Problem
to Emptiness Problem for Buchi automata
(Systla A.P., Vardi M.Y., Wolper P., 1985)

Symbolic Model Checking,
SMV Verification Tool
(K. McMillan, 1991)

Verification Tool SPIN
(Holzmann G.J., 1980-90)

Counter-example guided abstraction refinement, CEGAR
Verification Tools Blast, Slam
(T. Ball, S. Rajamani, 2000)

Historical Overview

Model of real-time computing systems: timed automata
(R. Alur, D.L. Dill, 1990)

Historical Overview

Model of real-time computing systems: timed automata
(R. Alur, D.L. Dill, 1990)

Verification algorithms for timed automata
(R. Alur, D.L. Dill, 1996)

Historical Overview

Model of real-time computing systems: timed automata
(R. Alur, D.L. Dill, 1990)

Verification algorithms for timed automata
(R. Alur, D.L. Dill, 1996)

Verification Tool UPPAAL
(K. Larsen, 1995)

Achievements of Formal Methods

One of the first examples of successful practical application of formal methods of program verification is the detection in 1993 of errors in the FutureBus + cache coherence protocol, which was adopted as an IEEE standard and defines the bus architecture for high-performance computers.

Verification of the protocol was made by means of model checking system SMV.

Achievements of Formal Methods

Other examples of successful use of formal verification tools include

1. Immos Ltd project (1988 г.) aimed at the development of microprocessors for transputers.

   Specification language Z.

   Programming language Occam.

   Verification method — proof-theoretic approach based on Hoare logic.

   Results: errors were found in rounding and calculating residuals in the floating point arithmetic block; development completed 3 months faster than the alternative development group that did not use formal verification
   Oxford university and Immos Ltd were awarded Queen's Award for Technological Achievement.

Achievements of Formal Methods

2. GEC Alsthom, MATRA Transport, and RATP (Paris public transport operator) project to computerize the control system of the Paris metro (RER) (1988 r.).

   Specification language B.

   Programming language Modula-2.

   Verification method — proof-theoretic approach based on Hoare logic.

   Results: the use of formal verification methods made it possible to avoid testing individual system modules and focus only on global testing. Later, full automation of one of the lines of the Paris metro was carried out in the same way.

Achievements of Formal Methods

3. National Westminster Bank and Platform Seven Co project on the development of an electronic payment system using smart-cards (8-bit microprocessor, 256 bytes of RAM and several kilobytes of ROM) (1992 г.).

   Specification language B.

   Verification method — proof-theoretic approach based on Hoare logic.

   Results: a number of possible vulnerabilities have been discovered and fixed; Proof of the safety standard requirements has been obtained (200 pages.).

Achievements of Formal Methods

4. Airbus project for design automation of software and electronic airborne equipment and ground transport (1982- currently).

   Specification language Esterel.

   Programming languages C, Verilog.

   Verification method — model checking, automatic code generation based on formal specifications.

   Results: reducing the number of errors in the code, 70% of the code is generated automatically, increasing the efficiency of making changes to the project.

Achievements of Formal Methods

5. Design of an automatic control system for a movable barrier to protect Rotterdam from floods (1992- currently).

   Specification languages Z, Promela.

   Programming language C.

   Verification method — model checking by means of SPIN.

   Results: detection of errors in specifications and code.

Achievements of Formal Methods

6. Full verification of the linux microkernel (seL4) by a group of software engineers from NICTA (Australia). (2009 – currently).

   Specification language Hascel.

   Programming language C, Assembler

   Verification method — proof-theoretic by means of Isabelle/HOL (interactive prover).

   Results: Proof that there are no errors of a certain kind in the microkernel code. "We can predict precisely how the kernel will behave in every possible situation."

END OF LECTURE 1.