SPEZIFIKATION UND VERIFIKATION:
MODEL CHECKING
MO 10 - 12 UHR, C - 221
RÜDIGER VALK

Inhalt

• Kapitel 0: Geschichte und Bedeutung des Model-Checking
• Kapitel 1: Die temporalen Logiken CTL und LTL
• Kapitel 2: Algorithmen für CTL und LTL
• Kapitel 3: Tools für CTL und LTL
• Kapitel 4: Binäre Entscheidungsbäume
• Kapitel 5: Symbolisches Model-Checking
• Kapitel 6: Model-Checking durch Auffalten
The development of model checking methods is one of the towering achievements of the late 20th century in terms of debugging concurrent systems. This development came about in the face of the pressing need faced by the computing community in the late 1970’s for effective program debugging methods. The correctness of programs is the central problem in computing, because there are often very designs that are correct, and vastly more that are incorrect. In some sense, defining what ‘correct’ means is half the problem – proving correctness being the other half of the problem.

The quest for high performance and flexibility in terms of usage (e.g., in mobile computing applications) require systems (software or hardware) to be designed using multiple computers, processes, threads, and/or function units, thus making system behavior highly concurrent and non-intuitive. With the rapid progress in computing, especially with the availability of inexpensive microprocessors, the computer user community found itself in the late 1970’s in a position where it had plenty of inexpensive hardware but close to no practical debugging methods for concurrent systems. We will now examine the chain of events that led to the introduction of model checking in this setting.
The enterprise of *sequential program verification* pioneered, among others, by Floyd [40], Hoare [55], and Dijkstra [37] was soon followed by the quest for *parallel program correctness*, pioneered, among others, by Owicki and Gries [93], and Lamport [73].
“Based on [our] software developer and user surveys, the [US] national costs of an inadequate infrastructure for software testing is estimated to range from $22.2 to $59.5 billion. Over half these costs are borne by users...”

Quote For The Day

When the time is ripe for certain things, these things appear in different places in the manner of violets coming to light in early spring.

(Wolfgang Bolyai to his son Johann in urging him to claim the invention of non-Euclidean geometry without delay.)

Quote from Clarke & Emerson 81

“The task of proof construction is in general quite tedious and a good deal of ingenuity may be required to organize the proof in a manageable fashion.

We argue that proof construction is unnecessary in the case of finite state concurrent systems and can be replaced by a model-theoretic approach which will mechanically determine if the system meets a specification expressed in propositional temporal logic.

The global state graph of the concurrent systems can be viewed as a finite Kripke structure and an efficient algorithm can be given to determine whether a structure is a model of a particular formula (i.e. to determine if the program meets its specification)”.
The Model Checking Problem

The Model Checking Problem (CE81):

Let $M$ be a Kripke structure (i.e., state-transition graph).

Let $f$ be a formula of temporal logic (i.e., the specification).

Find all states $s$ of $M$ such that $M, s \models f$.

Advantages of Model Checking

- No proofs!!
- Fast (compared to other rigorous methods such)
- Diagnostic counterexamples
- No problem with partial specifications
- Logics can easily express man concurrency properties
Main Disadvantages

- Proving a program helps you understand it. Bogus!
- Temporal logic specifications are ugly. Depends on who is writing them.
- Writing specifications is hard. True, but perhaps partially a matter of education.
- State explosion is a major problem. Absolutely true, but we are making progress!

Petri Net Tools

Tadeo Murata:

"I started working on Petri nets from mid-1970, and attended the 1st International Workshop on Petri Nets held in 1980 and thereafter. But I do not recall any papers discussing formal verification using Petri nets (PNs) BEFORE 1981. Also, I doubt there were any PN reachability tools before 1981. MetaSoft Company was selling earlier PN drawing tools and may have had a primitive one before 1981."

Kurt Jensen:

"Like Tad I do not think there is any work on Petri net TOOLS prior to 1981. The first Meta Software tool was made in the mid 80's and was merely a drawing tool for low level Petri nets.

High-level Petri nets were invented in the late 70's. The first two publications appeared in TCS in 1979 and 1980. It is only after this that people really started the construction of tools. The first simulator for high-level nets and the first state space tools for these were made in the late 80's and the early 90's."
Bochmann and Protocol Verification

Gregor Bochmann: For a workshop organized by André Danthine, I prepared the paper "Finite State Description of Protocols" in which I presented a method for the verification of communication protocols using the systematic exploration of the global state space of the system (sometimes called reachability analysis). This paper was later published in Computer Networks (1978) and was much cited.

At the same time, Colin West had developed some automated tools for doing essentially the same as what I was proposing, but I learned about his activities only later.

The Importance of Model Checking

Gregor Bochmann continued: "The need for exploring the reachable state space of the global system is the basic requirement in protocol verification.

Here model checking has not provided anything new.

However, temporal logic has brought a more elegant way to talk about liveness and eventuality; in the protocol verification community we were talking about reachable deadlock states (easy to characterize) or undesirable loops (difficult to characterize)."
“If I press the eject button, I am guaranteed to be safely ejected from a burning airplane in less than 5 seconds.”

versus

“If I am lucky to be in a plane that was debugged by an expert reader of a program who happened to spot a bug, then I might get ejected in a reasonable amount of time.”

In one thread of work that was evolving in the late 1970’s, some scientists, notably Pnueli [97], had the vision of focusing on concurrency. In a nutshell, by focusing on control and not data, it becomes possible to model a system in terms of finite-state machines, and then employ decision procedures to check for its reactive properties. Even after such simplifications, system control tends to be highly non-intuitive, and hence simply not amenable to any reasonable social processes. Automated analysis of finite-state models can, on the other hand, automatically hunt bugs and report them back. Pnueli’s vision lead to Manna, Pnueli, and many others developing temporal logic proof systems [79, 80].
The breakthrough towards algorithmic methods for reasoning about concurrent systems (as opposed to the initial proof theoretic methods), was introduced in the work of Clarke and Emerson [18], Queille and Sifakis [99], and Clarke, Emerson, and Sistla [19]. This line of work also received multiple fundamental contributions, notably from Vardi and Wolper who introduced an automata theoretic approach to automatic program verification [120], and a team of researchers at AT&T Bell Laboratories, notably by Holzmann, Peled, Yannakakis, and Kurshan [58, 51, 72, 59], who developed various ways to build finite-state machine models and formally analyze them. Known as model checking, these methods relied on (i) creating a finite state model of the concurrent system being verified, and (ii) showing that this model possesses desired temporal properties (expressed in temporal logic). Graph traversal algorithms were employed in lieu of deductive methods, thus turning the whole exercise of verification largely into one of building system models as graphs, and performing traversals on these graphs without encountering state explosion.

Holzmann and Protocol Verification

Holzmann:

“My first paper-method (never implemented) was from 1978-1979 -- as part of my PhD thesis work in Delft.

My first fully implemented system was indeed the 'pan' verifier (a first on-the-fly verification system), which found its first real bug in switching software (based on a model that I built in the predecessor language to Spin's Promela) at AT&T on November 21, 1980.”
J.P. Queille and J. Sifakis, Specification and Verification of Concurrent Systems in CESAR,

- Technical Report 254 June 1981,
- International Symposium on Programming, Turin, April, 1982
- Springer Lecture Notes in Computer Science 137, published in 1982

SPECIFICATION AND VERIFICATION OF CONCURRENT SYSTEMS IN CESAR

J.P. Queille and J. Sifakis
Laboratoire IMAG, BP 53X
38041 Grenoble Cedex, France

Abstract:

The aim of this paper is to illustrate by an example, the alternating bit protocol, the use of CESAR, an interactive system for aiding the design of distributed applications.

CESAR allows the progressive validation of the algorithmic description of a system of communicating sequential processes with respect to a given set of specifications. The algorithmic description is done in a high level language inspired from CSP and specifications are a set of formulas of a branching time logic, the temporal operators of which can be computed iteratively as fixed points of monotonic predicate transformers. The verification of a system consists in obtaining by automatic translation of its description program an Interpreted Petri Net representing it and evaluating each formula of the specifications.
process SENDER
  ( output X : exp ;
    input A : ack ) ;
X : data ;
Y : boolean := 0 ;
begin
  loop
    (X := (A, Y) ;
     do
       if ~A then
         Y := true ;
       end
       if A then
         X := Y ;
       else
         X := ~Y
       end
     loop
   end loop
end SENDER ;
Given $L$ and a transition system $S=(Q,\rightarrow)$ we define an interpretation of $L$ as a function $\llbracket \cdot \rrbracket$ associating to each formula of $L$ a truth-valued function of the system state in the following manner:

- $\forall q \in Q \cdot \llbracket \top \rrbracket(q) = \top$
- $\forall q \in Q \cdot \llbracket \bot \rrbracket(q) = \bot$
- $\forall q \in Q \cdot \llbracket \text{true} \rrbracket(q) = \top$ if $q = \top$ and $\llbracket \text{false} \rrbracket(q) = \bot$ if $q = \bot$
- $\forall f_1, f_2 \in L \cdot \llbracket f_1 \lor f_2 \rrbracket(q) = \top$ if $\llbracket f_1 \rrbracket(q) = \top$ and $\llbracket f_2 \rrbracket(q) = \bot$ or $\llbracket f_1 \rrbracket(q) = \bot$ and $\llbracket f_2 \rrbracket(q) = \top$ or $\llbracket f_1 \rrbracket(q) = \bot$ and $\llbracket f_2 \rrbracket(q) = \bot$
- $\forall f \in L \cdot \llbracket \text{POT}(f) \rrbracket(q) \equiv \forall s \in E \forall k \in \mathbb{N} \cdot \llbracket q \}\llbracket q = s(k) \rrbracket \land \llbracket f(s(k)) \rrbracket$
- $\forall f \in L \cdot \llbracket \text{INEV}(f) \rrbracket(q) \equiv \forall s \in E \forall k \in \mathbb{N} \cdot \llbracket q \}\llbracket q = s(k) \rrbracket \land \llbracket f(s(k)) \rrbracket$

Obviously, $\llbracket \text{POT}(f) \rrbracket(q)$ represents the set of the states $q$ of $S$ such that there exists an execution sequence starting from $q$ containing a state satisfying $\llbracket f \rrbracket$. We say that $\llbracket \text{POT}(f) \rrbracket(q)$ is the set of the states from which some state of $\llbracket f \rrbracket$ is potentially reachable. In the same way, $\llbracket \text{INEV}(f) \rrbracket(q)$ is the set of the states from which $\llbracket f \rrbracket$ is inevitably reachable in the sense that every execution sequence starting from a state of this set contains a state satisfying $\llbracket f \rrbracket$. 

ACM TURING AWARD HONORS FOUNDERS OF AUTOMATIC VERIFICATION TECHNOLOGY

Researchers Created Model Checking Technique for Hardware and Software Designers

NEW YORK, February 4, 2008 – ACM, the Association for Computing Machinery, has named Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis the winners of the 2007 A.M. Turing Award, widely considered the most prestigious award in computing, for their original and continuing research in a quality assurance process known as Model Checking. Their innovations transformed this approach from a theoretical technique to a highly effective verification technology that enables computer hardware and software engineers to find errors efficiently in complex system designs. This transformation has resulted in increased assurance that the systems perform as intended by the designers. The Turing Award, named for British mathematician Alan M. Turing, carries a $250,000 prize, with financial support provided by Intel Corporation and Google Inc. Clarke of Carnegie Mellon University, and Emerson and Sifakis, of the University of the Texas at Austin, working together, and Sifakis, working independently for the Centre National de la Recherche Scientifique at the University of Grenoble in France, developed this fully automated approach that is now the most widely used verification method in the hardware and software industries.

ACM President Stuart Feldman said the work of Clarke, Emerson and Sifakis has had a major impact on designers and manufacturers of semiconductor chips. “These industries face a technology explosion in which products of unprecedented complexity have to be manufactured as expected by companies that hope to survive. This verification advance enabled these industries to shorten time to market and decrease product costs,” Feldman said. 

Model Checking as a standard procedure for quality assurance has enabled designers and manufacturers to address verification problems that span both hardware and software. It has also helped them to gain mathematical confidence that complex computer systems meet their specifications, and it has provided added

25
Two Big Breakthroughs!

Significant progress was made on the State Explosion Problem around 1990:

- **Symbolic Model Checking**
  - Coudert, Berthet, and Madre 89
  - Burch, Clarke, McMillan, Dill, and Hwang 90;
  - Ken McMillan’s thesis 92

- **The Partial Order Reduction**
  - Valmari 90
  - Godefroid 90
  - Peled 94

Dealing with Very Complex Systems

Special techniques are needed when symbolic methods and the partial order reduction don't work.

Four basic techniques are

- Compositional reasoning,
- Abstraction,
- Symmetry reduction, and
- Induction and parameterized verification
State explosion—having to deal with an exponential number of states—is an unfortunate reality of model checking methods because finite-state models of concurrent systems tend to *interleave* in an exponential number of ways with respect to the number of components in the system.
Effective methods to combat state explosion became the hot topic of research — but meanwhile model checking methods were being applied to a number of real systems, with success, finding deep-seated bugs in them! In [14, 13], Bryant published many seminal results pertaining to binary decision diagrams (BDD) and following his popularization of BDDs in the area of hardware verification, McMillan [83] wrote his very influential dissertation on symbolic model checking. This is one line of work that truly made model checking feasible for certain “well structured,” very large state spaces, found in hardware modeling. The industry now employs BDDs in symbolic trajectory evaluation methods (e.g., [2]).

Symbolic Model Checking
An approach to the state explosion problem
Kenneth L. McMillan
May, 1992
CMU-CS-92-131

Contents

1 Introduction 11
  1.1 Background 12
  1.2 Temporal logic 12
  1.3 Axiomatic model checking 14
  1.4 Remarks on the thesis 16
  1.5 Related research 18
  1.6 Conclusion 20
  1.7 Other symbolic methods 20

2 Symbolic model checking 23
  2.1 Temporal logic 25
    2.1.1 Linear time 25
    2.1.2 Branching time 27
    2.1.3 Time complexity 27
  2.2 The temporal logic LTL 29
  2.3 Axiomatic model checking of LTL 30
    2.3.1 First-order characterization of LTL 30
    2.3.2 Symbolic CTL model checking 35
      2.3.2.1 Quantifier-free formulae 35
      2.3.2.2 Representing sets and relations 36
      2.3.2.3 CTL formulas 38
      2.3.2.4 Binary Decision Diagrams 40
  2.4 Extensions 40
    2.4.1 Fuzzy sets 40
    2.4.2 Stochastic state machines 49
    2.4.3 Nondeterministic state machines 54
    2.4.4 Concurrent state machines 54
    2.4.5 Complex synthesis 54
    2.4.6 Complex with and OR-ODDs 62

32
Model checking has truly caught on in the area of hardware verification, and promises to make inroads into software verification—the area of “software model checking” being very actively researched at the time of writing this very sentence. In particular, Boolean satisfiability (SAT) methods are being widely researched, as already discussed in Section 18.3. In modern reasoning systems, SAT and BDD methods are being used in conjunction with first-order (e.g., [92, 110]) reasoning systems, for example in tools such as BLAST [53]. In addition, higher-order logic (e.g., [3, 47, 94]) based reasoning systems also employ BDD, SAT, and even model checking methods as automated proof assistants within them. As examples of concrete outcomes, we can mention two success stories:
Model checking in the design of modern microprocessors: All modern microprocessors are designed to be able to communicate with other microprocessors through shared memory. Unfortunately since only one processor can be writing at any memory location at a given time, and since “shared memory” exists in the form of multiple levels of caches, with further levels of caches being far slower to access than nearly levels of caches, extremely complex protocols are employed to allow processors to share memory. Even one bug in one of these protocols can render a microprocessor useless, requiring a redesign that can cost several 100s of millions of dollars. No modern microprocessor is sold today without its cache coherence protocol being debugged through model checking.

Model checking in the design of device drivers: Drivers for computer input/output devices such as Floppy Disk Controllers, USB Drivers, and Blue-tooth Drivers are extremely complex. Traditional debugging is unable to weed out hidden bugs unless massive amounts of debugging time are expended. Latent bugs can crash computers and/or become security holes. Projects such as the Microsoft Research SLAM project [9] have transitioned model checking into the real world by making the Static Driver Verifier [8] part of the Windows Driver Foundation [122]. With this, and other similar developments, device-driver writers now have the opportunity to model-check their protocols and find deep-seated bugs that have often escaped, and/or have taken huge amounts of time to locate using traditional debugging cycles.
Has the enterprise of model checking succeeded? What about social processes? We offer two quotes:

\textit{Model checking has recently rescued the reputation of formal methods [64].} (1997)

\textit{Don’t rely on social processes for verification [38].} (1999)

21.2.1 Why model checking?

The design of most reactive systems is an involved as well as exacting task. Hundreds of engineers are involved in planning, analyzing, as well as building and testing the various hardware and software components that go into these systems. Despite all this exacting work, at least two vexing problems remain:

- Reactive systems often exhibit nasty bugs only when field-tested. Unfortunately, at such late stages of product development, identifying the root cause of bugs as well as finding solutions or workarounds takes valuable product engineering time. A manufacturer caught in this situation can very easily lose their competitive advantage, as these late life-cycle bug fixes can cost them dearly—especially if they miss critical market windows.
• The risk of undetected bugs in products is very high, in the form of law-suits and recalls. Since software testing methods are seldom exhaustive, product managers have a very difficult time deciding when to begin selling products.

\[2\] Software is often like a bridge that does not fail when subject to 100 tons or 101 tons of weight, but suddenly collapse when 101.1 tons of weight are applied.

Formal methods based on model checking are geared towards eliminating the above difficulties associated with late cycle debugging. While model checking is not a panacea, it has established a track record of finding many deep bugs missed by weeks or months of testing. Specifically,

• model checking is best used when a reactive protocol is in its early conceptual design stages. This is also the most cost-effective point

• model checking can return answers — either successful verification outcomes or high level counterexamples — often in a matter of a few minutes to a few hours. In contrast, testing can wastefully explore vast expanses of the state-space over weeks or months of testing. Error location can also become nightmarishly hard during testing, as the state-space sizes are large, and because an astronomically large number of computation steps may be executed from when the actual erroneous steps were carried out until when the system crashes or other symptoms of “ill health” are manifested.
21.3 Büchi automata, and Verifying Safety and Liveness

Büchi automata are automata whose languages contain only infinite strings. The ability to model infinite strings is important because of the fact that all bugs can be described in the context of infinite executions. We now elaborate on these potentially unusual sounding, but rather simple, ideas. Broadly speaking, all errors (bugs) in reactive systems can be classified into two classes: safety (property) violations and liveness (property) violations.

- Safety violations are bugs that can be presented and explained to someone in the form of finite executions (finite sequence of states) ending in erroneous states. Some examples of systems that exhibit safety violations are the following:
  - two people who, following a faulty protocol, walk opposite in a narrow dark corridor and collide;
  - an elevator which, when requested to go to the 13th floor, proceeds to do so with its doors wide open;
  - a process P which acquires a lock L and dies, thus permanently blocking another process, say Q, from acquiring L.

All finite executions of the form s_1 \ldots s_k can be modeled as infinite executions that infinitely repeat the last state, namely s_1 \ldots (s_k)^\omega. Modeling finite executions as infinite executions allows one to employ Büchi automata.
Liveness violations are bugs that can be presented and explained to someone only in the form of an infinite execution in which a desired state never occurs. In practice, liveness violations are those that end in a bad “lasso” shaped cyclic execution path which does not contain the desired state. Examples of liveness violations are:
- two people who, following a faulty protocol, engage in a perpetual ‘dance,’ trying to pass each other in a narrow well-lit corridor;
- an elevator that permanently oscillates between the 12th and 14th floors when requested to go to the 13th floor;
- A process \( P \) which acquires a lock \( L \) precisely before another process \( Q \) tries to acquire it, and releases the lock precisely after \( Q \) has decided to back off and retry; this sequence repeats.

---

A **liveness property** is a property stating that “something good will eventually happen.”

---

Liveness properties are best described using the **AF** or **F** combinators:

| AF done | F done |
| AG (req ⇒ AF grant) | G (req ⇒ F grant) |
| AG AF tick | G F tick |

---

All infinite executions in finite state systems can be cast into the form \( s_1 s_2 \ldots (s_j \ldots s_k)^\omega \) where the **reachable bad cycle** \( (s_j \ldots s_k)^\omega \) is called the “lasso.” Liveness verification of finite state systems reduces to finding one of these reachable lassos.