Formal Verification

B Meenakshi

author photo
author intro

This is a short tutorial on formal methods/techniques for specifying and verifying complex software and hardware systems. A few examples of successful industrial use of these are also presented.

Computers are ubiquitous these days and are used to control various safety critical systems like aircrafts, satellites, medical instruments, etc. Software that is developed to operate these systems is complex, consisting of many modules, each with thousands of lines of code. Any errors in the functioning of these programs results in serious loss of life and money. An example of such a failure is that of the Ariane 5 rocket, which exploded in June 1996, less than forty seconds after it was launched. The reason for this explosion was a software error in the computer that was responsible for calculating the rocket's horizontal velocity. An exception was caused during the conversion of a 64-bit floating point number into a 16-bit signed integer. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This caused the computer to fail and the backup computer too failed due to the same reason. Consequently, no instruction regarding altitude was transmitted to the onboard computer and this resulted in the rocket failing.

Keywords

This example (and many more) clearly indicates the need for developing reliable hardware and software systems. How is this ensured during the development of a system? This article describes one method to ensure correctness. In the sequel, we use the term 'system' to refer to a software program or hardware system.

A typical software development life cycle is depicted in

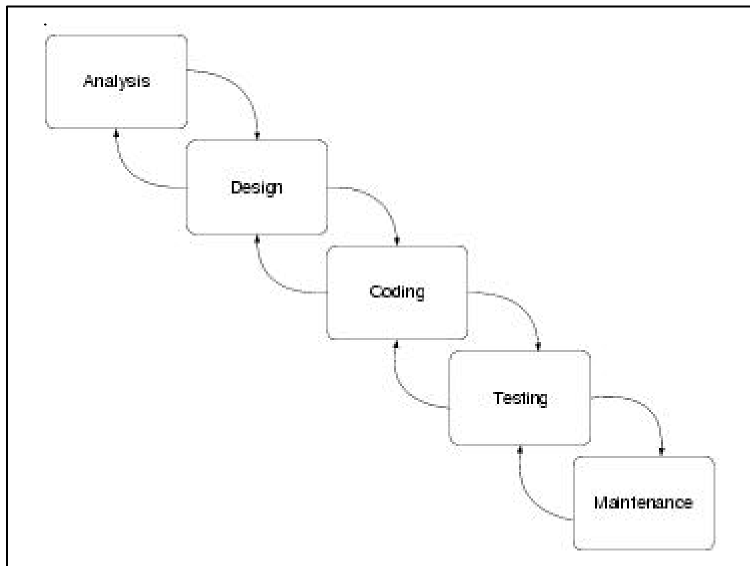


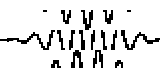
Figure 1. Waterfall model of software development life cycle.

Figure 1. Requirements describing properties of the software to be developed are specified first. This is followed by design of the software system which involves developing a high-level (followed by a detailed) layout of the various components of the system. The next step is the coding phase which usually also includes testing of the individual modules that are coded. Coding is followed by testing of the components and successful testing leads to the software being deployed and maintained for use. Each of these phases critically influences the correctness of the system being developed.

1. Testing and Simulation

Testing and *simulation* are well-known and traditional techniques used to check that the software written is correct with respect to its functionality. Vigorous testing and code review techniques are available to test the software written in almost any programming language. These techniques are considered highly effective and very few coding bugs of any significance escape detection.

However, very few defects in the final product are due



to bugs contributed during coding. For example, of 197 critical faults detected during the testing phase of the Voyager and Galileo spacecraft, just 3 were coding errors [1]. About 50% of the faults were traced to requirements and 25% to design and the rest due to other errors. This is a typical example of a prevalent problem – majority of faults in software arise in requirements and design and very few occur due to coding.

Another problem with the techniques of testing and simulation is that they do not cover 'all' possible *behaviours* of the system being developed. Behaviour of a system refers to a way in which the values of the variables in the program change when it is executed with different inputs. The *state space* of a system is the set of all possible *states* or modes the system can be in. A *state* of a system typically talks about the values of all the variables involved, the *location* where each component of the system resides in, etc.

Usually, many programs have a large number of states, one for every possible value that a variable can take. Take, for example, a program which has an integer variable x. Clearly, x can take as many values as allowed by the memory available, each representing a different state. If memory limitations are abstracted away, the program can be considered to have infinitely many number of states. Testing and simulation work by explicit substitution of values to variables and it is very tedious to cover all possible behaviours for such a program. Consequently, certain critical behaviours might be left out while testing and these might correspond to the system failing.

2. Formal Verification

Formal verification is a technique that developers can use to construct systems that operate reliably. It is complementary to testing and should be used in conjunction

with testing to increase reliability of the system being developed. It is worth noting that formal verification is not a way to ensure that the system being developed is 100% correct; it enhances reliability of the system by ensuring that it meets certain functional requirements, particularly at the earlier stages of design.

Formal verification is based on *formal methods* which are mathematically based languages, techniques and tools for *specifying* and *verifying* systems [2].

- Specifying a system means writing down the requirements about the system in a mathematical language.

- Verification is the next step to specification and involves *proving* formally that the system meets its requirement.

There are various *tools* that aid in specification and verification. These tools provide notations and algorithms for a system developer to formally specify and/or verify a system.

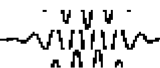
3. Specification

Specification is the process of describing a system and its desired properties. Properties or requirements of a system being developed usually completely characterize the system and are generally written in a common language. These include:

Functional or behavioural properties: The system should never hang, every request from a client should be followed by a response, etc.

Structural properties: The system should possess a main server and a backup server, the colour of the delivered product should be grey, the size of the product should not exceed a specified dimension, etc.

Timing properties: The status on user screen should be



Specification language	URL
CASL	http://www.cofi.info/CASL.html
CSP	http://vl.fmnet.info/csp/
JML	http://www.cs.iastate.edu/~leavens/JML/
Maude	http://maude.cs.uiuc.edu/overview.html
Murphy	http://sprout.stanford.edu/dill/murphi.html
SDL	http://www.sdl-forum.org/
Z	http://vl.zuser.org/

Table 1. Some specification languages.

periodically updated every 5 seconds, the receipt of a message should be acknowledged within 10 seconds etc.

Formal specification involves specifying the behavioural and timing properties in a mathematical language. This has its advantages. An exercise of precise specification helps to uncover ambiguities and incompleteness present in the requirements of a system and formal specification is the first step to formal verification.

Various specification languages are available to specify properties of systems. A few are listed below. *Table* 1 lists some more specification languages along with the URLs of their websites where the reader can get more information.

- Z, CASL (Common Algebraic Specification Language), B, etc. are languages to specify properties of sequential systems.

- Communicating Sequential Processes (CSP) [4], statecharts [5], SDL (Specification and Description Language), etc. are used to specify properties of concurrent or distributed systems (i.e. systems consisting of many interacting components).

- Temporal logics [3] are among the most popular specification languages used to specify properties of finite

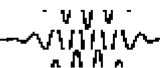
state systems. We discuss temporal logics through an example below.

Consider the simple program given below.

```
byte x;
(1) x := 13;
x := x/2+1;
(2) if x <= 0
        goto (1);
else
        x := x/2;
        goto (2);
```

Most interesting properties would deal with the achievable and non-achievable values of the variable `x` during an execution. Consider, for instance, properties like p: *the value of* `x` *is odd* and q: *the value of* `x` *is* 13. Assuming that the initial value of `x` (not initialized by the program) is 0, the sequence of *truth values* for p when the program is executed is given by *false, true, true, true, true, true, true, true, false, false,* ... and that for q is *false, true, false, false, false, false,* .... Notice that this sequence is infinite as the program does not stop.

Temporal logics give us a way of formalizing properties about possible and impossible sequences of boolean values of properties like p and q. The *formulas* of this logic talk about how the behaviour of the system evolves over time. In the context of this example, temporal logic formulas can specify properties like p is invariantly true, p always implies eventually q, etc. Temporal logic possesses *temporal operators* to formally represent notions like 'invariance', 'eventuality', etc. These operators are *interpreted* over (infinite) runs of a system and talk about properties of such runs. We will illustrate a method of checking if this program meets the above specifications in a later section.



Notable examples

There have been various successful attempts to formally specify systems. We list a few from [6].

• This example is from the avionics industry. Most commercial aircraft being operated in the US need to possess TCAS (Traffic Collision Avoidance System) system. The manufacturer of this system formally specified its set of requirements using a variant of the statecharts specification language. The effort unearthed many ambiguities in the original set of requirements. Algorithms were developed to automatically check TCAS specifications for completeness and consistency and going a step further, provably correct code can be automatically generated from TCAS specifications now.

• IBM Hursley Laboratories and Oxford University collaborated in the 1980s to formalize part of IBM's Customer Information Control System (an on-line transaction processing system) specification using the specification language Z. Through this exercise, IBM estimated a 9% reduction in the total development cost through reduction in the number of errors discovered and by detecting them early. This work received the Queen's award for technological achievement.

Verification

Formal verification is the next step to formal specification and involves checking/proving that a system satisfies a given property. Formal verification relies on building a mathematical model of the system and on formally specifying the requirements to be checked against the system. *Verification tools*, i.e. tools performing formal verification take two inputs: a system and its specification and check if the system *satisfies* the specification. Depending on how the modelling and the checking are done, there are two fundamental techniques in formal verification: *model checking* and *theorem proving*. We

discuss these techniques in detail below.

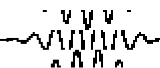
4.1 *Model Checking*

Model checking can be done only for systems that have a finite number of states. *Model checkers*, i.e., tools which perform model checking take two inputs: a finite state model of the system and a property specified formally. A model checker checks if the system *satisfies* the property and gives a 'yes' or 'no' answer. If the answer is 'no', i.e., if the system does not satisfy the property, model checkers also output a *counter example*, i.e., a run of the system which violates the property. The counter example can be analyzed to discover bugs in the system design.

As mentioned earlier, there are many systems which possess variables ranging over possibly infinite values. To make such systems amenable to model checking, the technique of *abstraction* is used to build a finite state representation of the system. Abstraction is a way of restricting the system in such a way that only those behaviours of the system that are relevant to the property being checked will be considered. Such a restricted system usually has only a finite number of states and we can then use model checking to verify that the system satisfies the property. Some model checkers possess features to suggest certain simple abstractions of a given system model.

Figure 2 describes the process of model checking. The book [7] provides a comprehensive presentation of the theory and practice of model checking.

The system is usually modelled as a finite state automaton – which is a finite graph with the set of system states as vertices [8]. Edges of the automaton model transitions from one state to another. Computations of the system (its set of *behaviours*) are modeled by *runs* of the automaton. Runs tell us how the system moves from



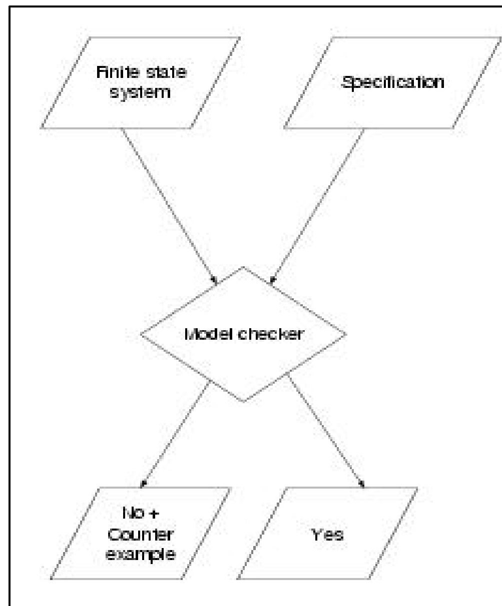


Figure 2. Overview of model checking.

one state to another through transitions which model the computations happening in the system.

For example, *Figure* 3 gives the automaton corresponding to the program given in Section 3. The automaton is represented as a graph and has four states, each represented by a vertex. The initial state of the automaton (from where its computation begins) is marked by an incoming arrow into the state. Directed edges labelled by statements represent transitions of the automaton from one state to another. A *run* of the automaton represents one of its computations and is a sequence of states such that the first state is one of its initial states and every state in the sequence is related to its successor by a transition.

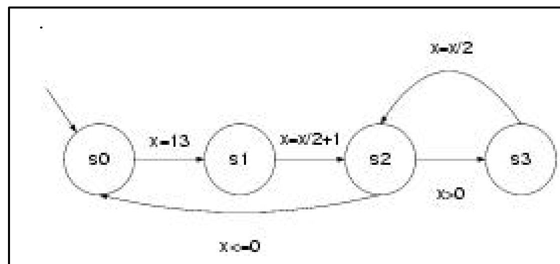


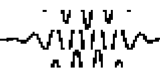
Figure 3. Model of a simple program.

As mentioned in the previous section, specification is modelled as a formula in temporal logic. A run of the considered system is said to be a *model* of the formula if it *represents* a behaviour described by the formula. We say that a system *satisfies* a temporal logic formula if every run of the system turns out to be a model of the formula. Model checkers possess algorithms to automatically check if a system satisfies a formula.

We illustrate one such algorithm for the example in *Figure* 3. Notice that the automaton given in the figure is *deterministic* – at every state, there is exactly one successor state as decided by value of the variable x. The sequence $s0, s1, s2, s3, s2, s3, s2, s3, s2, s0, \ldots$ is a run of the automaton. In order to talk about this run satisfying a property like p is invariantly true (where p is as defined in the previous section), we need to augment this run with the value variable x takes in each state. Such a run is then considered *consistent* with the original run only if all condition labels that appear in the transitions evaluate to true with the augmented value of x. One such augmented run is $(s0,0),(s1,13),(s2,7),(s3,7),(s2,3),$ $(s3,3),(s2,1),(s3,1),(s2,0),(s0,0),\ldots$.

We can extend this notion to define an augmented finite state automaton representing our program. In this example, the augmented finite state automaton would have nine *augmented* states – $(s0,0),(s1,13)$, $s2$ will be augmented with values $7,3,1,0$ of x to generate four states and $s3$ with values $7,3,1$. The truth or falsity of a formula like p is invariantly true is then dictated by the sequence of *truth values* for p at every augmented state – *false, true, true, true, true, true, true, true, false, false,* .... The requirement of p being invariantly true clearly does not hold as there are states in the augmented run where p is *false*.

In general, such a check is done by performing *reachability analysis* to check if a state violating a property is



reachable through a run from one of the initial states in the augmented automaton. These algorithms use standard depth first search routines on the automaton by considering it as a graph. There are properties which cannot be checked by performing a simple reachability analysis. Various other algorithms are available to model check such properties [7].

A few popular model checkers are briefly described.

• SPIN (Simple Promela INterpreter) is an open source model checking tool used for verifying distributed systems like communication protocols, network applications, multi-threaded code etc. System is specified in the language Promela as a *network of communicating automata* and property is given as a linear-time temporal logic formula. SPIN is one of the most widely used model checkers and won the ACM software system award in 2001.
URL: http://spinroot.com/spin/whatispin.html

• SMV (Symbolic Model Verifier) is a model checker used to verify hardware designs and is freely available for academic use. The system is modelled using *binary decision diagrams* and specification is given as branching time temporal logic formula.
URL: http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/

• UPPAAL is a model checker used for the verification of real-time systems and is also available free for academic use. System is modelled as a *network of timed automata*. The model checking algorithm can check if the system satisfies certain invariant properties.
URL: http://www.docs.uu.se/docs/rtmv/uppaal/

Notable examples

• AT & T applied model checking to the development of the International Telecommunication Union (ITU) ISDN User part protocol in the early 1990s. About 7500 lines of source code of the protocol written in SDL were

verified to satisfy 145 requirements. About 55% of the original design requirements were found to be logically inconsistent and a total of 112 errors were discovered.

- In 1992, a research group in CMU found errors in the IEEE Futurebus+ standard 896.1−1991 using the model checker SMV – the first time model checking was used to find errors in an IEEE standard.

- In 2001, the model checker SPIN was used to verify legacy flight software from NASA's Deep Space One mission. The process not only unearthed a known error in the launch software but also discovered a second scenario under which a similar error could occur.

There are various other successful case studies from the industry [6]. In fact, model checking has become so popular that industry is building their own model checkers. The WWW virtual library on formal methods lists some of the formal methods companies [2].

4.2 *Theorem Proving*

Theorem proving is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. The logic is presented by a *formal system*, which defines a set of *axioms* and a set of *inference rules*. Theorem proving is the process of finding a *proof* of a property from the axioms of the system by using the axioms and rules and also, derived definitions and intermediate lemmas.

Theorem provers are tools that assist in theorem proving. They aid in constructing proofs of a system satisfying the property and are semi-automatic as against model checkers which are fully automatic. On the other hand, theorem proving can directly deal with infinite state spaces whereas model checking applies only to finite state systems. *Table* 2 lists some well-known theorem provers.

Suggested Reading

[1] R Lutz, Analyzing software requirements errors in safety-critical embedded systems, in *IEEE International Symposium on Requirements Engineering*, pp. 126-133, CA, January 1993.

[2] The WWW virtual library of formal methods: http://vl.fmnet.info/.

[3] M Huth and M Ryan, *Logic in Computer Science: Modelling and reasoning about systems* (Second edition), Cambridge University Press, 2004.

[4] C A R Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[5] D Harel, Statecharts: A visual formalism for complex systems, in *Science of Computer Programming*, Vol. 8, pp. 231-274, 1987.

[6] E M Clarke and J M Wing, Formal methods: State of the art and future directions, in *ACM Computing Surveys*, Vol. 28(4), pp. 626-643, 1996.

[7] E Clarke, O Grumberg and D Peled, *Model Checking*, MIT Press, 2000.

[8] D Kozen, *Automata and Computability*, Springer-Verlag, 1997.

Theorem prover	URL
PVS	http://pvs.csl.sri.com/
ACL2	http://www.cs.utexas.edu/users/moore/acl2/
Coq	http://coq.inria.fr/
Analytica	http://library.wolfram.com/infocenter/Articles/3152/

Table 2. Some theorem provers.

Notable examples

• The Pentium FDIV bug that caused Intel to take a $475 million charge against revenues with a problem in the lookup table of an SRT divider was successfully verified (after the occurrence of the bug!) by various communities. A formally verified treatment of the general theory of SRT division was given using the PVS theorem prover by a group from Stanford Research Institute.

• In the mid nineties, Motorola's Complex Arithmetic Processor used for Digital Signal Processing (DSP) was formally specified using the ACL2 theorem prover. The design was tracked as it was evolving and the binary micro code of various DSP algorithms were verified in the process.

Conclusion

Formal methods have demonstrated success in specifying and verifying safety-critical software, protocol standards and hardware designs. Due to the commercial pressure to produce high quality software, the role of formal methods is increasing in the system development process. The ideal situation would be when system developers would be trained to pick and use a formal verification tool of their choice during development to increase reliability. Lot of research is concentrated on making the tools and notations accessible to system developers who are non-experts in this area.

Address for Correspondence
B Meenakshi
Honeywell Technology Solutions Lab
151/1, Doraisanipalya
Bannerghatta Road
Bangalore 560076, India.
Email:
Meenakshi.Balasubramanian@honeywell.com