Model Checking

B. Feigin

March 23, 2005

1 Introduction

We start with a motivational observation that the predominant strategies for software system verification — unit and regression testing, coverage analysis and so on — are largely heuristic in nature. Though fertile when the defect density is high, they often fail to uncover the more intricate bugs [1]. The system requirements themselves are usually expressed in natural language resulting in ambiguities [9].

Traditional techniques are not sufficient to ensure that the zero-defect requirement encountered in environments where impact of failure is disproportionately high (such as health- and safety-critical systems and financial applications) is satisfied. As the old maxim goes, *there's always one more bug*[1].

Formal methods — namely *deductive verification* and *model checking* — built on rigorous mathematical foundations aim to provide rock-solid correctness guarantees called for in such circumstances. Application of these methods, by necessity, also promotes strictness and precision in specification of system requirements.

Deductive verification (or *theorem proving*) entails axiomatization of the system domain using detailed knowledge about the system and establishment of inference (*deduction*) rules [6]. One then manually constructs correctness proofs using these axioms and rules — a process requiring much skill and time.

[1]Affectionately known as *Lubarsky's Law of Cybernetic Entomology*

In contrast, model checking, which is based on exhaustive exploration of possible the states of the system, aims to be fully automatic, requiring little operator intervention. Additionally, in order to use a model checker, the operator need not possess detailed knowledge of and extensive experience in mathematical logic that are necessary to perform deductive verification. These benefits lower the entry barrier for adoption of model checking as a standard quality assurance technique.

In the rest of the essay, we go on to take a more detailed look at model checking (as it applies to software systems), the one major limitation preventing more extensive application of the method and some possible remedies, and finish with a look at some of the currently available software tools.

2 Model Checking

In short, model checking is a collection of techniques for automated formal verification of finite-state concurrent systems. We start off with a bird's eye view of the process (Figure 1) and proceed to refine it in the following paragraphs.

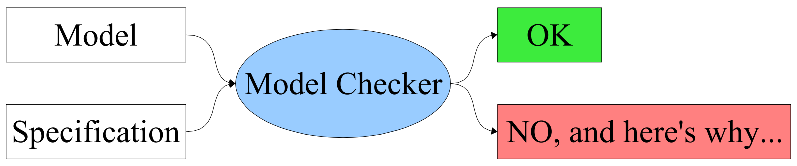

Figure 1: Model checking — a high-level view

Given a model (an abstraction of the system) and a specification of the properties that are required to hold (relating to absence of deadlocks, liveness, invariants etc.), the model checker verifies whether the former satisfies the latter. A counterexample is produced upon discovery of a violation.

The model is defined using some suitable formalism, for example a process algebra (see Figure 5). Rather than manually constructing the model, it may be more appropriate to derive it from other descriptions of the system. For example, at design stage, UML diagrams may be annotated with such information [4] or it may be extracted from source [8] or executable [5] code post-implementation.

Abstraction is usually necessary to make verification tractable (see Section 4 below). For example, when analysing a communications protocol for deadlocks, the actual data passed around is of no significance [8]. It is important, however, that relevant details are not thrown away — if the behaviour of the model does not correspond closely to the actual system, the results of analysis are worthless.

For reasons which will become clear later, model checkers cope best with systems whose *state* information is compact and can be easily manipulated. This naturally favours control-oriented systems which do not perform intricate transformations on complex data structures [6]. Such systems are known as *reactive systems* and are characterised by continuous interaction with their environment. Examples include hardware controllers and various security and communications protocols.

We now proceed to formalise the concepts of *model* and *specification*.

3 Transition Systems and Temporal Logic

We reason about reactive systems in terms of their *state*. It is therefore convenient to introduce the notion of a *state transition system* to describe the evolution of such a system.

A *Kripke transition system* [3] T over a set of *atomic propositions* $\mathtt{AP}$ is a four-tuple $(S, \mathtt{Act}, \rightarrow, I)$ where S is the set of states, $\mathtt{Act}$ the set of actions (e.g. program statements), $\rightarrow \subseteq S \times \mathtt{Act} \times S$ is the transition relation and $I : S \rightarrow 2^{\mathtt{AP}}$ an *interpretation* (i.e. $I(s)$ for some $s \in S$ is the set of propositions which are true in s, e.g. $a = 1$)

The structure is readily visualised as a graph (see Figure 2). T can be rooted with an initial state $s_0 \in S$ and unfolded into an infinite *execution tree*.

We express assertions about system behaviour using *temporal logics* which extend propositional logic with the notion of time without explicitly introducing it as a quantitative measure. Formulae are constructed from atomic propositions, boolean connectives and temporal operators. We differentiate between *linear-* and *branching-time* logics.

Propositional Linear-Time Logic (PLTL) is the basic linear-time logic. Important operators in PLTL include: $\mathbf{X}\ \varphi$ ("next φ"), $\varphi\ \mathbf{U}\ \psi$ ("φ until

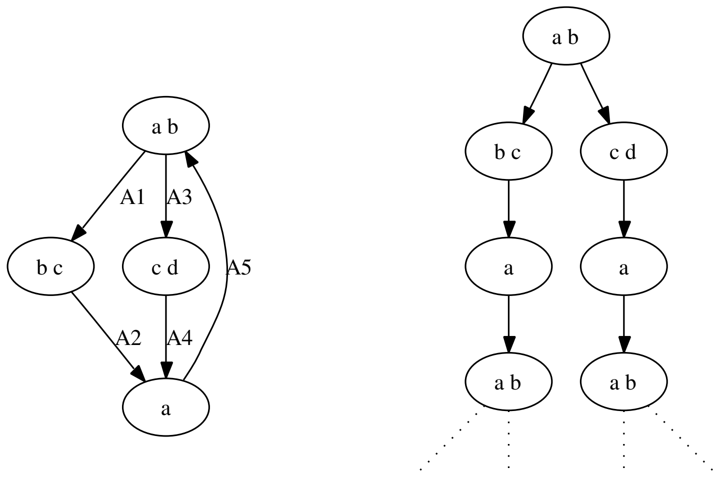

Figure 2: Transition system and top of execution tree.

ψ"), **F** φ ("eventually φ") and **G** φ ("always φ"). Formulae are interpreted over linear paths [2]. We are in general interested in verifying that *all* paths leading from the starting state(s) satisfy the formula.

As Figure 3 shows, the semantics are intuitive.

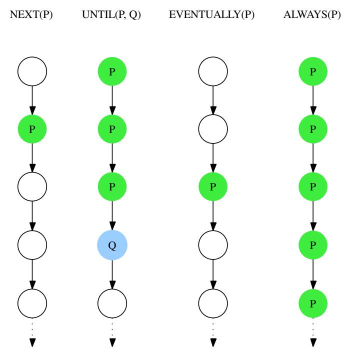

Figure 3: Interpretation of PLTL operators.

PLTL is used to express *correctness* properties of the system.

Computational Tree Logic (CTL) is a kind of branching-time logic. It gives selectivity by introducing existential and universal quantifiers — **E** and **A** respectively [3]. These alternate with PLTL operators. CTL formulae are interpreted at the states [2]: the PLTL properties must hold on some or all paths emerging from the state (as determined by the preceding quantifier).

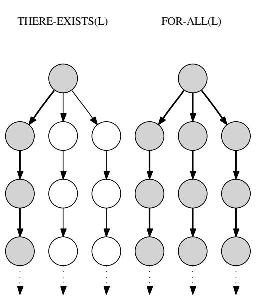

Figure 4: Quantification over paths in CTL formulae.

CTL allows one to express *possibility* properties [2].

As examples of temporal logic formulae, consider: $\mathbf{AG}\, \neg(owns_1 \wedge owns_2)$ [2] (where $owns_i$ signifies ownership of a resource by process i) which asserts the necessity for mutual exclusion on access to that resource and **AG** (**EF** *Restart*) [1] which means that from any state it should be possible to reach the *Restart* state.

Given a transition system T and a temporal logic formula φ, the job of the model checker is to decide whether $T \models \varphi$.

There are two main categories of model checking algorithms: *local* (usually used by PLTL model checkers) and *global* (used with CTL) [2].

We note that more "user-friendly" descriptions (such as specialised modelling languages) are used in practice in preference to these low-level formalisms. As a real-world example of the type of input model checkers take, consider Figure 5 which shows a description of the Dining Philosophers problem in terms of the *Finite State Processes (FSP)* process algebra.

The *Labelled Transition System Analyser* [7] — a model checking tool which accepts FSP as input — diligently reports that the philosophers of Figure 5 may well starve to death, by providing a sequence of transitions leading to a deadlock.

```
/** Concurrency: State Models and Java Programs
 *             Jeff Magee and Jeff Kramer
 *
 */

PHIL = (sitdown->right.get->left.get
          ->eat->left.put->right.put
          ->arise->PHIL).

FORK = (get -> put -> FORK).

||DINERS(N=5)=
   forall [i:0..N-1]
   (phil[i]:PHIL
   ||{phil[i].left,phil[((i-1)+N)%N].right}::FORK).

menu RUN = {phil[0..4].{sitdown,eat}}
```

Figure 5: Naïvely Dining Philosophers in FSP process algebra [7].

4 The State Explosion Problem

Applicability of model checkers based on explicit state enumeration is seriously limited by the *state explosion problem* — the number of states in most non-trivial systems is astronomical, in fact, exponential in the number of parallel components.

Also, we noted above that transition systems are not usually constructed directly, but instead some higher level language is used. Now, [2] notes that the size of a transition system corresponding to such a description will be exponential in the length of the description.

We survey some of the proposed techniques to help alleviate the problem:

Partial Order Reduction This optimisation is based on the observation that for loosely interacting processes, it is not necessary to examine every possible interleaving of concurrent actions, provided the said actions *commute*. By eliminating redundancies, the size of the state space is reduced.

Abstraction The technique is based on eliminating any details which do not affect the particular property being checked. It involves constructing an *abstraction relation* such that when the abstract model satisfies a property, it is provably the case that the actual system also does [2].

Symmetry Reduction This technique pulls in the machinery of group theory in order to obtain a reduced system by exploiting symmetries within it. For example, in cases where system behaviour remains unaffected when some data values are permuted [2].

4.1 Symbolic Model Checking

A radically different approach is *symbolic model checking*. Here the states of the system are represented *implicitly*, hence the size of the state space ceases to be a limiting factor [6].

Binary Decision Diagrams (BDDs), an efficient representation of boolean formulae, are used. The temporal formulae can be checked directly on the BDDs avoiding explicit state construction altogether [6].

5 Sample Applications

Model checking techniques have been applied in a variety of hardware and software projects. We give a couple of the more interesting examples of the latter category.

Fluke IPC [8] The SPIN model checker was used to verify the the highly concurrent Interprocess Communications subsystem of the Fluke microkernel. The Promela model was derived directly from C source code for some parts and built from ground up for others (in order to take advantage of built-in Promela functionality). Two serious bugs were identified.

DEOS Avionics Operating System [5] The Java PathFinder model checker was used to check for a subtle error which has been previously discovered during manual code analysis. For this purpose, the relevant part of the system (a *slice*, in fact) was translated from C++ to Java. It is noted in particular that using partial order reductions helped to find the error much quicker.

6 Software

Currently available model checking packages include:

SPIN [2] One of the biggest successes in model checker construction. SPIN takes in a system model expressed in PROMELA (PROcess MEta LAnguage) and a PLTL specification. Uses so-called *on-the-fly* techniques in order to avoid preconstruction of states.

Java PathFinder (JPF) [3] Works directly on Java bytecode rather than a specialised modelling language. Geared towards deadlock detection and does not allow PLTL checking. JPF is an explicit-state model checker and as such uses various methods such as symmetry and partial-order reductions to tame the state space.

LTSA [4] The Labelled Transition System Analyser can be used to verify concurrent systems described using the FPS process algebra. Both system and required properties are represented as finite state machines.

7 Conclusions

We have given a brief overview of model checking. As a different breed of quality assurance tool, model checking facilitates reaching levels of reliability and integrity unachievable with conventional methods.

In conclusion, we observe that the biggest rewards are reaped by combining model checking with state space reduction techniques described above as well as static analysis methods such as program slicing [**?**].

References

[1] E. M. Clarke, O. Grumbler, D. A. Peled. *Model Checking.* MIT Press, 1999

[2] S. Merz. Model Checking: A Tutorial Overview. *Lecture Notes in Computer Science* 2067, pp. 3–38, 2001

[2]`http://www.spinroot.com`

[3]`http://ase.arc.nasa.gov/visser/jpf/`

[4]`http://www.doc.ic.ac.uk/~jnm/book/ltsa-v2/index.HTML`

[3] M. Müller-Olm, D. Schmidt, B. Steffen. Model-Checking: A Tutorial Introduction. *Lecture Notes in Computer Science* 1694, pp. 330–354, 1999

[4] N. Kaveh. Model Checking Distributed Objects. *Lecture Notes in Computer Science* 1999, pp. 116–128, 2001

[5] W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda. Model Checking Programs. *Automated Software Engineering Journal* 10(2), pp. 203–232, 2003

[6] P. Wolper. An Introduction to Model Checking. `http://www.montefiore.ulg.ac.be/~pw/papers/papers.html`, 1995

[7] J. Magee, J. Kramer. Web page of *Concurrency: State Models & Java Programs.* `http://www.doc.ic.ac.uk/~jnm/book/`

[8] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, G. Back. Formal Methods: A Practical Tool for OS Implementors. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '97)*, pp. 44–56, Las Vegas, NV, June 1997 `http://www.cs.utah.edu/flux/papers/index.html`

[9] J. Wu, G. Liu, V. Lane. *CIS 841 Web Book – Fall 1999, Chapter 4 Formal Verification.* `http://www.cis.ksu.edu/ hankley/d841/Fa99/chap4.html`

[10] G. K. Palshikar. An introduction to model checking. `http://www.embedded.com/showArticle.jhtml?articleID=17603352`, *Embedded.com*, 2004