Defence Science Journal, Vol. 59, No. 3, May 2009, pp. 314-317
© 2009, DESIDOC

SHORT COMMUNICATION

Software Reliability through Theorem Proving

S.G.K. Murthy and K. Raja Sekharam

Defence Research & Development Laboratory, Hyderabad – 500 058

ABSTRACT

Improving software reliability of mission-critical systems is widely recognised as one of the major challenges. Early detection of errors in software requirements, designs and implementation, need rigorous verification and validation techniques. Several techniques comprising static and dynamic testing approaches are used to improve reliability of mission critical software; however it is hard to balance development time and budget with software reliability. Particularly using dynamic testing techniques, it is hard to ensure software reliability, as exhaustive testing is not possible. On the other hand, formal verification techniques utilise mathematical logic to prove correctness of the software based on given specifications, which in turn improves the reliability of the software. Theorem proving is a powerful formal verification technique that enhances the software reliability for mission-critical aerospace applications. This paper discusses the issues related to software reliability and theorem proving used to enhance software reliability through formal verification technique, based on the experiences with STeP tool, using the conventional and internationally accepted methodologies, models, theorem proving techniques available in the tool without proposing a new model.

Keywords: Software reliability, formal methods, software testing, software model checking, software theorem proving, stanford temporal theorem prover (STeP), autopilot software.

1. INTRODUCTION

As software systems are becoming complex and mission-critical, ensuring software reliability is the prime concern. Particularly in mission critical software systems, unreliable software results in high costs for end users and developers[2]. These are caused due to specification related as well as run-time related errors. Various types of testing techniques like peer reviews, code inspection and dynamic testing are used but these are effective in detecting presence of bugs never their absence. Run-time errors are a class of faults that are caused during execution and finding these using dynamic testing is a hard task. Sometimes run-time errors, like division by zero, causes the crashing of system. Array out of bounds access, invalid arithmetic operations, and unreachable code, are some of the run time problems.

Software reliability comprises various aspects that include good software engineering practices ranging from requirements elicitation, design patterns, architectural robustness, use of coding standards, selection of appropriate hardware and operating system to execute the software, use of appropriate case tools, comprehensive quality checks through peer and independent reviews. The development needs to be augmented with stringent monitoring and control methodologies throughout the development period as suggested and advocated by various quality models like International Standard Organisation (ISO), Capability Maturity Model (CMM), and Capability Maturity Model Integrated (CMMI). Strict adherence to documented process by these techniques does enhance the reliability and maintainability of the software. In the case of mission-critical and safety- critical software, where the path coverage, branch coverage, decision coverage for all input conditions, the system needs to be verified with mathematical-intensive techniques like formal methodologies to ensure a high reliability.

Formal verification techniques utilise mathematical logic to prove correctness of the software based on given specifications, which in turn improves the reliability of the software[3]. Formal verification of hardware and software systems gained more importance after the Pentium bug in 1994[4] that costed millions of dollars. Unlike normal testing, formal verification considers predicates as input assertions values, so that program is verified for all possible conditions without executing the software. Theorem proving is a branch of formal verification technique used to prove the properties of the software programs.

The formal verification tools comprise models, methodologies, algorithms based on first-order or higher-order logic truly represented in mathematical form. The details of implementation, the actual process for validating the assertions are not widely published in the public domain in the case of all the tools referred – STeP, PVS, HOL. Based on literature survey and the simplicity of expressing the code to be verified, STeP tool has been chosen to verify the properties. The autopilot software has been verified to prove the properties of the software with algorithm that is implemented.

Received 2 July 2008

This paper discusses the issues related to software reliability of mission-critical software through formal verification techniques and verification of properties of software using theorem prover to identify requirement as well as run time errors.

2. SOFTWARE RELIABILITY

Reliability is defined as the probability of failure-free operation for a specified time, in a specified environment, under a specific condition[1]. It means the software in an onboard application will be 99.9 per cent reliable during a period implies that failure may occur in one case out of 1000. Usually software faults are discovered either through program inspection or from the software failures. The presence of certain un known errors in the software are caused by a set of inputs, generated a set of erroneous outputs typically both. This situation is depicted in Fig. 1.

As reliability is the prime concern for mission-critical software, number of techniques like peer reviews, code inspection, and dynamic testing, etc are used throughout the development to improve software reliability.

3. FORMAL VERIFICATION METHODS

Formal verification methods for software comprise a set of techniques for proving the correctness of software for a possible combination of input values. So, a deterministic software program is correct, if it satisfies the intended input/output relation $\{P\}\ S\ \{Q\}$, where P is called pre condition. Q is called post condition or output assertion and S is the software program. With the help of formal verification techniques it is possible to prove the formula $\{P\}\ S\ \{Q\}$. Generally correctness is interpreted as partial and total as follows:

- *Partial correctness*: If a program starts from a state satisfying P, runs the code and completes, then Q will be true.
- *Total correctness*: If a program starts from a state satisfying P and runs the code, then eventually it will complete with Q being true.

Model checking and theorem proving are two types of techniques frequently used alone or in action combination to verify the properties of software. Detailed description is given in sections 3.1 and 3.2 about verification techniques.

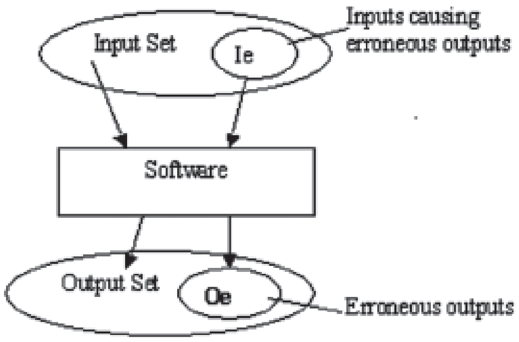


Figure 1. Input/output mapping.

3.1 MODEL CHECKING

Model checking is well known formal verification approach used in many applications. In this approach, an abstract model (for states) is constructed and used to verify the required properties. Verification of software by model checking is more suitable for control-intensive applications as it contains more if else statements[4]. Software verification using model checking involves translation of source into a model, so that it has a limitation on state space. As model checking is automatic, it has gained popularity in software verification industry; however software program states are not finite and contain complex data types, it is difficult to check the properties with the model checking technique.

3.2 THEOREM PROVING

As per Hoare, correctness of program, is achieved using pre-and post-conditions and inference such as assignment rule, composition rule, condition rule, while rule, for total and partial correctness. A simple inference is shown in Eqn.(1) for 'if' statement.

$$\frac{\{\ \phi_1\ \Lambda\ B\ \}\ C_1\ \{\ \phi_2\}\ \{\ \phi_1 \Lambda \neg B\}\ C_2\ \{\ \phi_2\}}{\{\phi_1\}\ \text{if B then}\ C_1\ \text{else}\ C_2\ \{\phi_2\}} \tag{1}$$

Further these ideas were extended by Dijkstra using the weakest pre condition concept[5]. As per Dijkstra, verification process starts from post-condition instead of pre-condition. The weakest pre-condition of a program S and a post condition Q is denoted by $Wp\ (S, Q)$ and it is a predicate that describes the set of all initial states that will guarantee termination of S in state satisfying Q, It is also represented as Hoare triple shown in Eqn.(2).

$$\{Wp\ (S,\ Q)\}\ S\ \{Q\} \tag{2}$$

For the verification of $\{P\}\ S\ \{Q\}$ using weakest precondition concept, it is necessary to find out $Wp\ (S, Q)$ and the property $P \subseteq Wp\ (S, Q)$ has to be proved.

To verify the run-time errors of a program, certain conditions are annotated in the code, using formal specification language. For verification of programs, usually properties to be proved are represented in first-order or temporal logic. In this paper, first-order logic based properties are considered to prove the given software.

Theorem proving is a technique, where the system to be proved and desired properties expressed in a mathematical form (i.e. first-order logic or temporal logic). Theorem proving technique considers set of axioms as pre or initial conditions and properties to be proved are post-conditions. In general pre-and post- conditions are asserted in the software using specialised tags. Utilising deductive and inductive proving logics, automatic theorem prover tools verify system properties. There are various public domain tools available for software theorem proving (Ex PVS, HOL, STeP). STeP tool is considered to verify the properties for the example in Section 4.1.

4. VERIFICATION OF PROPERTIES USING THEOREM PROVING

As theorem-proving technique needs mathematical logic related formulas, there is a necessity to pre- process the input information (software, specifications). Following steps are involved in verification process.

(i) Conversion of system specifications (i.e. input and outputs) into a formal specifications using 1st order or temporal logic[7].

(ii) Inserting specifications obtained in (a) as pre and post conditions in the code.

(iii) Conversion of system (software functions) into SPL code manually or using C2SPL converter [8].

(iv) Verifying the properties using STeP tool[9].

Fig 2. illustrates the verification process using STeP tool.With the help of theorem proving tool, it is possible to verify that the software meet the given specifications for all possible conditions (i.e., pre-conditions). Apart from that by proper annotation of required checks, same theorem-prover tool can also be used to identify run-time errors. In the following example [Section (4.1)], an example is given to check the unreachable code in a function.

4.1 Example

A simple industrial example is given to verify the run time properties without executing the code. The following function (*Example1*) accepts an input *x* which contains any value i.e. *x* >=0 or *x* < 0. *Example1* function has, 'if' and else statements, and based on the value of *x*, each if statement is executed. As mentioned in Sections 4, required specifications are inserted with special tags as pre and post-conditions in the source code, which are shown in bold letters in the given function. To verify the unreachability for the statements given in the if and else statements, post condition property has to be designed accordingly. In the given example, to verify the unreachability of the last 'else statement', a post condition (rcnt < 4) is asserted.

The files Example1.spl (source code) and Example1.spec (formal specifications) are generated using c2spl tool and in turn these two files are fed to STeP tool as inputs. By applying the options B-INV and WLPC[9] repeatedly, post condition property is proved, which is indicated by STeP tool, as "The proof is COMPLETE". The post condition property rcnt < 4 is true for all input conditions of x, implies that the statements under last else are unreachable. If it is reachable, the Post condition is not true. Fig. 3. describes this situation.

```
double Example1 (int x)
{
  int xarray[8];
   short index = 1;
    short n;
    double return_status = 0 ;
   int rcnt = 0;
 /*pre ( ( x >=  0 ∨ x < 0) ∧ xarray[0] = 0  xarray[1]
= 1 ∧ xarray[2] = 2 ∧ xarray[3] = 3 ∧ xarray[4] =
4 ∧ xarray[5] = 5 ∧ xarray[6] = 7 ∧ xarray[7] = 10
∧ n = 5) end*/
      if(x <= xarray[0])
       {
           rcnt = 1;
           return_status = 0.0;
       }
       else if(x >= xarray[n-1])
       {
           rcnt = 2;
           return_status = 1.0;
       }
        else if(x >= xarray[index])
        {
           rcnt = 3;
           return_status = 2.0;
        }
       else
        {
               rcnt = 4;
              return_status = 2.0;
        }
        return(return_status);
/*post (rcnt < 4) end*/
}
```

5. CONCLUSIONS

Software reliability for the mission-critical software is discussed. The role of Software theorem-proving technique helps to enhance software reliability is presented. With the help of STeP (Stanford temporal theorem prover) tool, a live industrial example is presented to prove the properties. For verifying run-time errors of software, it is required to annotate various flags and desired properties, which requires more manual efforts. Proving loop properties usually requires constructing loop-invariants and verifying loop-invariants with induction- related techniques, requires

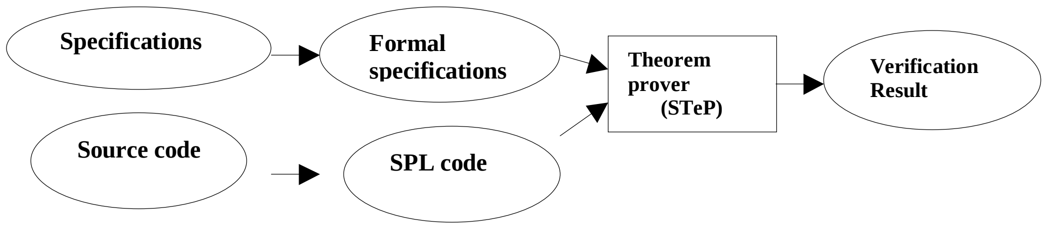
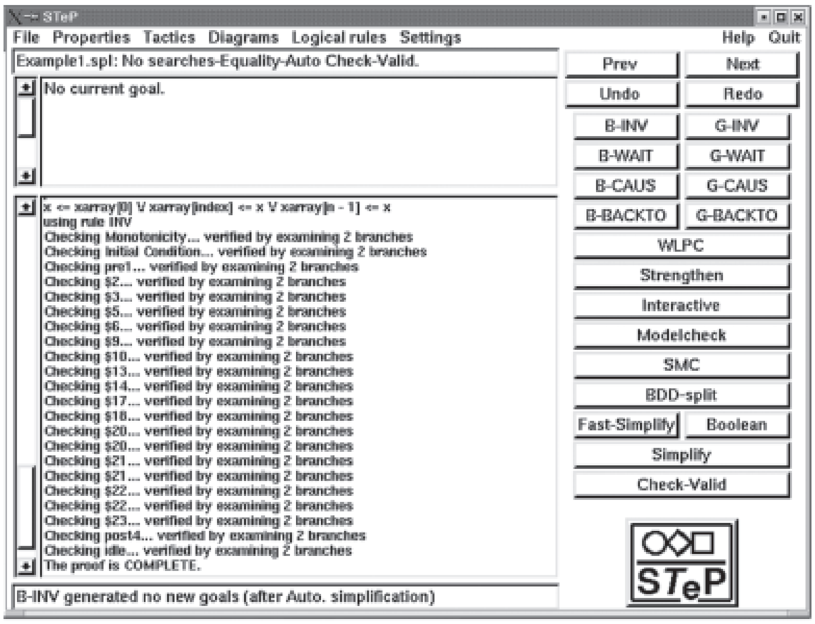


Figure 2. Verification process using STeP.



```
Example1.spl: No searches-Equality-Auto Check-Valid.

No current goal.

x <= xarray[0] V xarray[index] <= x V xarray[n - 1] <= x
using rule INV
Checking Monotonicity... verified by examining 2 branches
Checking Initial Condition... verified by examining 2 branches
Checking pre1... verified by examining 2 branches
Checking $2... verified by examining 2 branches
Checking $3... verified by examining 2 branches
Checking $5... verified by examining 2 branches
Checking $6... verified by examining 2 branches
Checking $9... verified by examining 2 branches
Checking $10... verified by examining 2 branches
Checking $13... verified by examining 2 branches
Checking $14... verified by examining 2 branches
Checking $17... verified by examining 2 branches
Checking $18... verified by examining 2 branches
Checking $20... verified by examining 2 branches
Checking $20... verified by examining 2 branches
Checking $21... verified by examining 2 branches
Checking $21... verified by examining 2 branches
Checking $22... verified by examining 2 branches
Checking $22... verified by examining 2 branches
Checking $23... verified by examining 2 branches
Checking post4... verified by examining 2 branches
Checking idle... verified by examining 2 branches
The proof is COMPLETE.

B-INV generated no new goals (after Auto. simplification)
```

Figure 3. Verification of properties using STeP GUI.

expertise and user efforts. As verification process is not fully automatic, it is hard to apply theorem proving for larger software systems, however formal methods have certain advantages over normal testing, combined utilisation of both the techniques ensures absence of bugs, so that software reliability is achieved.

REFERENCES

1. Sommerville,Ian. Software engineering. Addison-Wesley Publisher Ltd, 1978.
2. Ponarsard, C. *et al.* Early verification and validation of mission-critical systems.CETIC Research Center,Charleroi, Belgium, 2004.
3. Collins, Michael. Formal methods , Carnegie Mellon University, 1998.
4. Ouimet, Martin. Formal software verification: model checking and theorem proving. Embedded Systems Laboratory , Technical Report MIT, USA, 2005.
5. Popov, Nikolaj. Verification using weakest precondition strategy. Research Institute of Symbolic Computation, Hagenberg, Austria.
6. Ben-Ari, M. Mathematical logic for computer science, Printice Hall International (UK) Ltd.,1993.
7. Sandholm Tuomas. First-order logic (FOL), Computer Science Department. Carnegie Mellon University.
8. Sharma, Babita. *et al.*, Assertion checking environment (ACE) for formal verification of C program. Reliability Engineering & System Safety, 2003.
9. Bjorner, Nikolaj. *et al.*, The Stanford temporal prover Users Manual, Stanford University, 1998.

Contributors

Mr S.G.K. Murthy did MSc(Mathematics). He is presently working as Scientist D at the Defence Research and Development Laboratory (DRDL), Hyderabad. His areas of interest are: Software theorem proving techniques. Multi-sensor data fusion, information security, and soft computing.

Mr K. Raja Sekharam did MSc(Computer Science). He is presently working as Scientist E and is Head, IV & V Group at Defence Research and Development Laboratory, Hyderabad. His areas of interest are: Software theorem proving techniques, software testing, software reliability, and embedded systems.