End-to-End Deployment of Formal Methodology
- a Case Study on Multiple Reader/Writer Program

Moonjoo Kim, Inhye Kang
SECUi.COM R&D center
Seoul, Korea
{moonjoo,inhye}@secui.com

Abstract

The rapid increase in the significance of software systems has made software assurance a critical requirement in the information age. Formal verification of system design and testing system implementation with a variety of inputs have been used for this purpose. Each of these methodology, however, has its own weaknesses. First drawback of formal method is difficulty of writing formal specification correctly. Furthermore, verifying a system design does not guarantee the correctness of an implementation. Thus, testing of system implementation is unavoidable. Traditional testing, however, does not provide formal treatment on correct execution because testing criteria is informally described in many cases.

In order to build correct systems, it is required to deploy formal methodology end-to-end on entire software life cycles, not only on design stage. We demonstrate end-to-end deployment of formal methodology through the case study of multiple reader/writer system.

1 Introduction

In the past two decades, much research has concentrated on the methods for analysis and validation of software systems used in safety critical areas including avionics and automobiles. Many successful industrial case studies have been conducted in the area of formal verification [3]. Complete formal verification, however, has not yet become a prevalent analysis method. Reasons for this are as follows. First, complete verification of real-life systems remains infeasible. The growth of software size and complexity seems to exceed advances in verification technology. Second, formal verification assumes that given formal requirement specification is correct. However, building up formal requirement specification from informal specification is error-prone process without systematic guideline. Third, verification results apply not to system implementations, but to formal models of these systems. That is, even if a design has been formally verified, it still does not ensure the correctness of a particular implementation of the design. This is because an implementation often is much more detailed, and also may not strictly follow the formal design. So, there are possibilities for introduction of errors into an implementation of the design that has been verified. One way that people have traditionally tried to overcome this gap between design and implementation has been to test an implementation on a pre-determined set of input sequences. This approach, however, fails to provide accurate result about the correctness of the implementation because test coverage is not complete and test criteria is often ad-hoc.

Therefore, in order to build correct systems, it is necessary to deploy formal methodology end-to-end in entire software life cycles, not only on design stage. We concentrate on the following three stages of software life cycles in this paper: requirement specification, design verification, and implementation testing. At the requirement specification stage, which is the first stage in software development processes, a user describes his/her requirements on systems. Requirements are described in natural language with diagrams initially. A human user is responsible for building correct formal requirement specification from this informal requirement specification [6]. In spite of the importance of requirement engineering, this stage is often neglected by engineers and produces poor and incorrect requirement specifications, which affects the rest of software development steps and increases the software development cost severely. Therefore, formal requirement specification should be built through systematic and rigorous procedure. Given formal requirement specification, a user describes the design of system in formal language. More detailed system design a user describes, more information a user can obtain through formal verification. System model, however, should be simple enough to be handled by formal verification tools. Thus, a user has to create a system design at the right abstract level depending on requirement specification. Lastly, after the implementation stage, a user needs to test implementation of system. Although ad-hoc test criteria are prevailing, formal requirement specification can be used as a test oracle to check the correctness of implementation [10, 7, 2].

In this paper, we demonstrate these processes concretely using the well-known example of multiple
reader/writer system. Section 2 describes reader/writer (RW) system and requirement. Section 3 shows the process of building up formal requirement specification of RW system. Section 4 describes formal model of RW system and verification result. Section 5 shows the Java implementation of RW excerpted from [9] and testing of the implementation using the formal requirement specification built in Section 3. Finally, Section 6 concludes the paper and suggests future work.

2 Multiple Reader/Writer System

Multiple reader/writer (RW) system is an exemplary system of handling concurrency and mutual exclusion. Figure 1 shows the RW system. There are one common data area, multiple readers, and multiple writers. Readers read data from the common data area, to which writers write data.

![Multiple reader/writer problem](image)

Figure 1: Multiple reader/writer problem

Following rules are given to operate the RW system efficiently and consistently.

1. **Concurrency**
   multiple readers can read data from the common data area at the same time.

2. **Exclusive writing**
   - readers cannot read data from the common data area while a writer is writing into the area.
   - no more than one writer can write data into the common data area at the same time.

3. **High priority of writer**
   a waiting writer blocks readers from starting of read operations.

Concurrence is a rule for maximizing throughput of readers. Exclusive writing is to keep the RW system consistent. High priority of writer is for readers to read “fresh” data instead of old one.

For a reader to collaborate with other readers/writers for keeping these rules, a reader sends three signals. A reader sends a signal when it accesses to the area. Also, when a reader begins/ends reading, it sends a signal indicating beginning/ending of reading. Similarly, a writer sends signals for accessing to the area, beginning of writing, and ending of writing.

3 Formal Requirement Specification

In this section, we formulate the requirements described in Section 2. We assume that there are two readers and one writer for the sake of simplicity. The formulation demonstrated in this section, however, can be easily extended to general n readers and m writers system. There are 3 properties the RW system must satisfy.

1. Concurrency (CON)
2. Exclusive Writing (EW)
3. High Priority of a Writer (HPW)

We build a formal requirement specification for these three properties.

We consider the following 9 events of the RW system to express CON, EW, and HPW.

\[ \Sigma = \{ir1, rs1, rel, ir2, rs2, re2, ww, ws, we\} \]

where \(ir1\) stands for incoming reader1 which accesses to the common data area, but not yet starts read operation, \(rs1\) for reader1’s starting of reading, \(rel\) for reader1’s ending of reading, \(ww\) stands for waiting writer which accesses to the common data area, but not yet starts write operation. \(ws\) and \(we\) are beginning/ending of write operation. In addition to CON, EW, and HPW, the system has to satisfy correct event ordering (CEO), i.e. \(i \cdot\) ith occurrence of \(ir1\) must precede the ith occurrence of \(rs1\) and \(rs1\) must precede \(re1\). Note that CON is described automatically without any dedicated specification when we allow any combination of the 9 events unless the combination violates other requirements.

We first define a valid execution path which does not violate any of CON, EW, HPW, and CEO. Then, we transform a set of valid execution paths into a finite state machine by merging states in the paths.
Defn 1 (An execution path) An execution tree is a labeled transition system \((S, T, \Sigma)\) where \(S\) is a set of states and \(T : S \times \Sigma \times S\) is a set of transition over \(S\) with a set of label \(\Sigma\). A state \(s\) consists of the following 6 integer variables

\[
\begin{aligned}
    s &\triangleq (n_{ir1}, n_{rs1}, n_{ir2}, n_{rs2}, n_{uw}, n_{ws}) \\
\end{aligned}
\]

An execution path \(\sigma = s_0s_1...s_n\) is a sequence of states in an execution tree. \(s_i\) denotes the \(i\)th state of \(\sigma\).

Defn 2 (Definition of a state) \(\#ir1(s_i) \triangleq 0\).

\(\#ir1(s_i) \triangleq \) a number of event \(\text{ir1}\) in an event trace \(\rho = l_0...l_{i-1}\) such that \(s_i \xrightarrow{l_i} s_{i+1}\) where \(i > 0\). Similarly defined are \(\#rs1, \#re1, \#ir2, \#rs2, \#re2, \#uw, \#ws,\) and \(\#we\).

State \(\sigma\) of an execution path \(\sigma\) consists of the following 6 variables

\[
\begin{aligned}
    n_{ir1}(s) &\triangleq \#ir1(s) - \#rs1(s) \\
    n_{rs1}(s) &\triangleq \#rs1(s) - \#re1(s) \\
    n_{ir2}(s) &\triangleq \#ir2(s) - \#rs2(s) \\
    n_{rs2}(s) &\triangleq \#rs2(s) - \#re2(s) \\
    n_{uw}(s) &\triangleq \#uw(s) - \#ws(s) \\
    n_{ws}(s) &\triangleq \#ws(s) - \#we(s) \\
\end{aligned}
\]

Initial state \(s_0 \triangleq (0, 0, 0, 0, 0, 0)\)

\(n_{ir1}(s)\) indicates whether there is “active” \(\text{ir1}\) in an execution path \(s_0...s\). We can think that \(i\)th occurrence of \(\text{rs1}\) “cancels” the \(i\)th occurrence of \(\text{ir1}\). \(n_{ir1}(s) = 1\) means that \(\text{ir1}\) occurs \(i\) times and \(\text{rs1}\) occurs \((i - 1)\) times up to state \(s_i\), which means that \(\text{ir1}\) is “active”.

Defn 3 (Valid execution path) An execution path \(\sigma = s_0s_1...s_n\) is valid if either

- \(\sigma = \text{ir1} \) or \(\text{ir2} \) or \(\text{uw}\)
- For \(i = n - 1, s_0...s_i\) is valid and \(s_{i+1}\) satisfies the following conditions

1. correct event ordering (CEO)

\[
\begin{aligned}
    (n_{ir1}(s_{i+1}) &\leq 0 \land n_{rs1}(s_{i+1}) = 0 \lor 1) \\
    (n_{uw}(s_{i+1}) &\leq 0 \land n_{rs2}(s_{i+1}) = 0 \lor 1) \\
    (n_{uw}(s_{i+1}) &\leq 0 \land n_{rs2}(s_{i+1}) = 0 \lor 1) \\
\end{aligned}
\]

2. exclusive writing (EW)

\[
(n_{uw}(s_{i+1}) = 1 \rightarrow (n_{rs1}(s_{i+1}) = 0 \land n_{rs2}(s_{i+1}) = 0))
\]

We can generate a valid execution tree starting from \(s_0\) consisting of only valid executions because the next states of a state \(s\) can be totally determined according to the 6 variables of \(s\). According to CEO requirement of Definition 3, a variable in a state \(s\) can be 0 or 1. Thus, there exist at most \(2^6 = 64\) states in the RW system. It means that many states containing the same 6 values in a valid execution tree can collapse into a single state. By repeating collapsing of states, we obtain a finite state machine representing valid execution tree. Figure 2 shows the finite state machine representing requirement CON, EW, HPW, and CEO.

![Figure 2: A FSM for CON, EW, HPW, and CEO](image-url)

Figure 2: A FSM for CON, EW, HPW, and CEO

4 Formal Verification

In this section, we model the RW system using process algebra CCS [11]. We verify the RW system design against the requirement specification using Concurrent Workbench of New Century (CWNC) [4, 1].

4.1 Modeling of RW system

We model a RW system and requirement specifications in a process algebra CCS because CCS de-
Figure 3: Top-level structure of the RW system.

The description of a system provides concurrency and synchronization cleanly. A RW system consists of 8 processes as Figure 3. R1, R2, and W correspond to two readers and one writer. The other processes are for correct synchronization satisfying EW and HPW among two readers and writer. LOCK works as a lock ensuring mutual exclusion on the common data area among a writer and two readers. If a reader or a writer cannot proceed, it should sleep. SLEEP0 works for this purpose and shows a state that there is no sleeping writer or reader. AR0, AR1, and AR1 stand for states where a number of active reader is 0, a number of waiting writer is 0, and a number of active writer is 0. These processes move to AR1, WR1, and AR1 as a number of active reader, waiting writer, and active writer increases. All processes communicate with each other using synchronization events such as dec.3 and inc.3. Appendix A shows a complete RW system design usable with CWNC version 1.1.

4.2 Verification Results

There are various equivalence / preorder relations to define the relationship between a system and a requirement specification [5]. Most famous and well adopted equivalence / preorder is language equivalence / preorder for its intuitive meaning and simplicity. A correctness criteria using language equivalence / preorder is that the set of traces generated from a system design must be a subset of a set of traces generated from a requirement specification.

We verify our design S with a requirement specification S0 using CWNC version 1.1. Figure 4 shows the verification result. le -S may S S0 at line 1 of Figure 4 tests whether a set of traces generated S satisfies a subset of a set of traces generated S0 or not. le -S may S S0 S tests the other way.

Left column of Figure 4 shows the result of verifying L(S) S L(S0). Right column of Figure 4 shows the result of verifying L(S0) L(S). Line 12 shows that L(S) S L(S0) is TRUE but L(S0) L(S) is FALSE. Therefore, we verify that the system design S satisfies the requirement specification S0.

5 Testing using Formal Requirement Specification

We use a Java code of the RW system in the book “Concurrent Programming in Java” [9]. The Java code in the book has the same requirements as we have described in Section 3. Figure 5 shows the skeleton of the Java code. A reader performs three actions in its turn - beforeRead() at line 8 which tests whether a reader can start read operation or not, read() at line 9 which reads data, and afterRead() which cleans up. beforeRead() and afterRead() at line 10 are defined as synchronized. Thus, we can assume that they are atomic operations. Similar for beforeWrite(), write(), and afterWrite().

We instrument the Java code so that the beginning of beforeRead() generates ir, and beginning/ending of read() generates rs and re events. Similarly we instrumented beforeWrite(), write(). Generated events are fed into the CWNC through the customized filter interface. Using the step by step simulation facility of CWNC as a test oracle, we can test event traces generated from the RW system with regard to the formal requirement specification. We test the RW implementation 100 times. RW implementation generates an event trace of length 100 in each experiment. The implementation does not violate the requirement specification in the test.

6 Conclusion

We have demonstrated deployment of formal methodology on not only design phase, but also requirement specification phase and implementation testing phase through the case study of multiple reader/writer system. As we have seen in Section 3, building a correct formal requirement specification is not a naive
01: public abstract class RW {
02:     protected int activeReaders_ = 0;
03:     protected int activeWriters_ = 0;
04:     protected int waitingReaders_ = 0;
05:     protected int waitingWriters_ = 0;
06:     public void read(String id) {
07:         beforeRead(); // ir
08:         read_(id); // rs, re
09:         afterRead();
10:     }
11: }
12: public void write(String id) {
13:     beforeWrite(); // ww
14:     write_(id); // ws, we
15:     afterWrite();
16: }
17: }
18: ...
19: }

Figure 5: A skeleton Java code for the RW system (excerpted and modified from [9])

job, but requires thorough understanding of requirement as well as systematic description. Through the verification using language preorder, we prove that our design of the RW system satisfies all requirements of CON, EW, HPW, and CEO. We can use the same requirement specification to test the Java implementation of the RW system. The testing procedure is not quite satisfactory due to the complex user interface of CWNC and need of customized filter, we demonstrate a method of using formal requirement specification as a testing oracle.

Our study, however, does not cover the whole software life cycle. A big process missed in this paper is implementation stage. There has been research for generating a code from formal design specification [12]. However, so far, the quality of implementation in terms of performance and readiness of integration with other components is not yet usable. Another process we did not cover in this paper is run-time monitoring. The system in real-field is hard to be error-free in spite of best development efforts. One complementary solution is to monitor the execution of target program continuously at runtime. A monitor can detect a fault of system execution before the fault causes system crash and help users to correct the system [8]. We believe that application of formal methodology to whole software life cycles can eventually reduce the development cost and increases the assurance of correct execution of critical systems.

References
A  CCS Description of the RW System S and the Requirement Specificaiton S0

*****************************************************
* RW system description of 2 Readers and 1 Writer *
*****************************************************
proc S = (R1,R2,W)\{A\}W0\{LOCK\}[SLEEP0]\{dec_WW, inc_WW, dec_AV, inc_AV, dec_AR, inc_AR, 
zero_WW, zero_AV, zero_AR, zero_WA, zero_AV, zero_AR, non_zero_WW, non_zero_AV, non_zero_AR, lock, unlock, 
zero_sleep, one_sleep, two_sleep, dec_sleep, inc_sleep, 
wake_up)\}

proc W0 = zero_WW,W00 + inc_WW,W01
proc W1 = dec_WW,W00 + non_zero_WW,W01
proc A0 = zero_AV,A00 + inc_AV,A01
proc A1 = dec_AV,A00 + inc_AV,A02 
  + non_zero_AR,A01
proc A2 = dec_AR,A01 + non_zero_AR,A02
proc SLEEP0 = zero_sleep.SLEEP0+ inc_sleep.SLEEP1
proc SLEEP1 = one_sleep.SLEEP1 + inc_sleep.SLEEP2 
  + dec_sleep.SLEEP0
proc SLEEP2 = two_sleep.SLEEP2 + dec_sleep.SLEEP1

proc R1 = 'lock.r1.
  ( 'zero_WW. 
    ( 'zero_AV. 'inc_AR. 'unlock.READ1 
    + 'non_zero_AV. 'inc_sleep. 'unlock.R1') 
  + 'non_zero_WW. 'inc_sleep. 'unlock.R1)
proc R1 = 'wake_up.'lock.
  ( 'zero_WW. 
    ( 'zero_AV. 'inc_AR. 'unlock.READ1 
    + 'non_zero_AV. 'inc_sleep. 'unlock.R1')
  + 'non_zero_WW. 'inc_sleep. 'unlock.R1)

proc R2 = 'lock.r2.
  ( 'zero_WW. 
    ( 'zero_AV. 'inc_AR. 'unlock.READ2 
    + 'non_zero_AV. 'inc_sleep. 'unlock.R2')
  + 'non_zero_WW. 'inc_sleep. 'unlock.R2)
proc R2 = 'wake_up.'lock.
  ( 'zero_WW. 
    ( 'zero_AV. 'inc_AR. 'unlock.READ2 
    + 'non_zero_AV. 'inc_sleep. 'unlock.R2')
  + 'non_zero_WW. 'inc_sleep. 'unlock.R2)

proc W = 'lock.WW.'inc_WW.
  ( 'zero_AR. 
    ( 'zero_AV. 'dec_WW. 'inc_AV. 'unlock.WRITE 
    + 'non_zero_AR. 'inc_sleep. 'unlock.W') 
  + 'non_zero_AV. 'inc_sleep. 'unlock.W')
proc W = 'wake_up.'lock.
  ( 'zero_AR. 
    ( 'zero_AV. 'dec_WW. 'inc_AV. 'unlock.WRITE 
    + 'non_zero_AR. 'inc_sleep. 'unlock.W')
  + 'non_zero_AV. 'inc_sleep. 'unlock.W')

proc READ1 = rsl.r1.'lock. 'dec_AR.
  ( 'zero_sleep. 'unlock.R1 
    + 'one_sleep. 'wake_up. 'dec_sleep. 'unlock.R1 
    + 'two_sleep. 'wake_up. 'dec_sleep. 'wake_up. 
    'dec_sleep. 'unlock.R1)
proc READ1 = rsl.r2.'lock. 'dec_AR.
  ( 'zero_sleep. 'unlock.R2)