Model Checking for Self-Stabilizing Algorithm in C

Yiyan LIN
December 08 2010

Outline

- Preliminary
- Model Checking
- Temporal Logic
- Concurrent Modeling
- Self-Stabilizing
- Motivation
- Case Studies

Model Checking

Given a model of a system, test automatically whether this model meets a given specification.

- Process of Model Checking
  - **Modeling**: Convert a system to a formalism accepted by a model checking tool.
  - **Specification**: State the properties that the system must satisfy. For hardware and software systems, temporal logic is commonly used to describe specification, which can assert the behavior evolves over time.
  - **Verification**: Use model checking as a verification method to establish whether the description of a system satisfies the specification.

Temporal Logic

Describe the ordering of events in time without introducing time explicitly.

- \( F \ p \) – sometimes \( p \)
- \( G \ p \) – always \( p \)
- \( X \ p \) – next time \( p \)
- \( p \ U \ q \) – \( p \) until \( q \)
Concurrent Modeling

Non-determinism + Fairness

E. Allen Emerson

- **Non-determinism:**
  A current program starting in a given state may follow any one of a number of different computation paths in the tree corresponding to the different sequences of nondeterministic choices the program might take.

- **Fairness:**
  Each process be executed infinitely often.

**Example:**
A concurrent program \( P_1, P_2 \) with two processes, we would expect the corresponding sequence of interleaving to be the form:
\[ P_1, P_2, P_1, P_2, \ldots \]
on the other hand, we would not expect a sequence:
\[ P_1, P_1, P_1, P_2, P_2, \ldots \]
Unfair to process \( P_2 \).

Self Stabilization

A system is self-stabilizing if and only if: convergence and closure holds.

- **Convergence**
  Starting from any state, it is guaranteed that the system will eventually reach a correct state.
  Convergence property holds, if only if formula \( \text{F legitimate} \) holds.

- **Closure**
  Given that the system is in a correct state, it is guaranteed to stay in a correct state, provided that no fault happens.
  Closure property holds, if only if formula \( \text{legitimate} \rightarrow \text{A legitimate} \) holds.

Motivation

Some previous work:

- Tsuchiya, etc. Symbolic Model Checking for Self-Stabilizing Algorithms (2001)
  Utilize SMV for model checking self-stabilizing programs
  Weakness: Verification is only feasible for small number of processes

  Observe the cost of verification under weak fairness is more than 1000 times that of the cost without fairness.
  Two approaches for improving scalability:
  (1) decomposition
  (2) utilize weak stabilization.

Motivation

- To demonstrate the feasibility of utilizing Copper, a software model checker for concurrent C programs, to model check self-stabilizing algorithm and to gain more practical experience.

- To improve the cost of verification by applying approaches (1) verify algorithm without fairness (2) decomposition
**Copper**

- C file as Input to Copper

```c
void phil1()
{
    int eating;
    eating = 0;
    while(1) {
        pick_left_1();
        pick_right_1();
        eating = 1;
        if(eating != 1)
            assert(0);
        eating = 0;
        put_left_1();
        put_right_1();
    }
}

void phil2()
{
    int eating;
    eating = 0;
    while(1) {
        pick_left_2();
        pick_right_2();
        eating = 1;
        if(eating != 1)
            assert(0);
        eating = 0;
        put_left_2();
        put_right_2();
    }
}
```

- Define specification in Copper

```c
program phil1,phil2
{
    specification abs_1,{P0::eating == 0,P1::eating == 0},DpSpec1;
}
```

```c
ltl DpSpec1 { #G [(P0::eating == 0) || (P1::eating = = 0)]; }
```

**Case Study**

K-State Token Ring

- Two Actions

```c
process P0:
    if(p0.token == p1.token)
        p0.token=(p0.token+1) % (n+1);
```

```c
process P1:
    if(p0.token != p1.token)
        p0.token= p1.token;
```

- Define Program Under Fairness

```c
program token_proc0,
    token_proc1,
    token_proc2 {
    specification abs_1,{1,1,1},SPEC;
}
```

```c
ltl SPEC {#F legitimate & (legitimate => #X(legitimate));}
```

**Fairness is not necessary to guarantee self-stabilizing**

- Define Program Under Unfairness

```c
program phil1,phil2
{
    specification abs_1,{P0::eating == 0,P1::eating = = u},DpSpec1;
}
```

```c
ltl DpSpec1 { #G [(P0::eating == 0) || (P1::eating = = 0)]; }
```

- Two Actions

```c
process P0:
    if(p0.token == p1.token)
        p0.token=(p0.token+1) % (n+1);
```

```c
process P1:
    if(p0.token != p1.token)
        p0.token= p1.token;
```

- Define Program Under Fairness

```c
program token_proc,
    {
    specification abs_1,{1},SPEC;
}
```

```c
ltl SPEC {#F legitimate & (legitimate => #X(legitimate));}
```

<table>
<thead>
<tr>
<th>n=3</th>
<th>n=4</th>
<th>n=5</th>
<th>n=6</th>
</tr>
</thead>
<tbody>
<tr>
<td>fairness 0.47s</td>
<td>0.75s</td>
<td>2.03s</td>
<td>7.76s</td>
</tr>
<tr>
<td>without fairness 0.113s</td>
<td>0.36s</td>
<td>1.03s</td>
<td>2.35s</td>
</tr>
</tbody>
</table>

**Case Study**

Fairness is not necessary to guarantee self-stabilizing
Case Study
Raymond’s Tree Algorithm

- Unique token held in the tree
- A process can access the critical section, it must first acquire the token.
- There is no need for each node to be aware of the tree as a whole. It is sufficient that each node knows of the existence of its neighbors in the tree.
- A process without token should send request message to its holder. If the holder does not have the token, it must send the message to its holder.
- Each process only have one holder.

Figure 1: Holder Tree

Figure 1 illustrates a holder tree composed of 6 nodes. Node A holds the token, other nodes only knows their holders.

Case Study
Raymond’s Tree Algorithm

Fault Abnormality, etc. Constraint Based Automated Synthesis of Non-masking and Stabilizing Fault-Tolerance.

Constraints:

1. \( h : j \) node’s holder
2. \( P : j \) node’s parent, e.g. node E’s parent is D

Fault Transition:

- \( (h, j) \rightarrow (k, j) \) arbitrary value from its domain

\[
S_1 \land S_2 \land S_3
\]

\[
S_1 : (h, j) \rightarrow (k, j) \land (h, j) \leq k
\]

\[
S_2 : (h, j) \land (k, j) \land (h, k) \leq j
\]

\[
S_3 : (h, j) \land (k, j) \land (h, k) = j
\]

Case Study
Raymond’s Tree Algorithm

Constraints:

1. \( S_{1} \land S_{2} \land S_{3} \)
2. \( (h, j) \rightarrow (k, j) \land (h, j) \leq k \)
3. \( (h, j) \land (k, j) \land (h, k) \leq j \)
4. \( (h, j) \land (k, j) \land (h, k) = j \)

Case Study
Raymond’s Tree Algorithm

Figure 2: Holder Tree after faults occur violating S2

- \( F : E’s \) holder is D
- \( D : E’s \) parent
- \( E : Token held by E \)

Case Study
Raymond’s Tree Algorithm

Figure 2: Holder Tree after faults occur violating S3

- \( F : E’s \) holder is D
- \( E’s \) parent
- \( E : Token held by E \)
Case Study: Raymond's Tree Algorithm

Constraints:

(S1) ∀j: (h(j) = P(j)) ∨ (h(j) = j) ∨ (∃k: (P(k) = j) ∧ (h(j) = k))

- Requires that j’s holder can either be j’s parent, j itself, or one of j’s children.

(S2) ∀j: (h(j) = j) ∨ (h(j) = P(j)) ∨ (h(j) = k)

- Requires that the holder tree conforms to the parent tree.

(S3) ∃j: (h(j) = j) ∧ (h(j) = P(j)) ∧ (h(j) = k)

- Requires that there are no cycles in the holder relation.

Case Study: Raymond’s Tree Algorithm

Recovery Actions:

¬(S1) → h(j) = j \{h(j) = P(j) \land h(j) = k\}

¬(S2) → h(j) = P(j) \land h(j) = k

¬(S3) → h(j) = j \land h(j) = P(j) \land h(j) = k

Verification Approach:

Decompose the self-stabilizing program into parts which could be verified without fairness. The recovery actions for self-stabilizing could be verified without fairness.
Case Study: Raymond’s Tree Algorithm

Still working for the experiment, and the results need to be compared may include,

- Verification time for both token-passing component and recovery components under fairness manner.
- Verification time for recovery components with fairness.
- Verification time for recovery components without fairness.