Formal Verification of Distributed Algorithms using Distributed PlusCal

Student: Heba Al-Kayed
Supervisors : Stephan Merz, Horatiu Cirstea

DEPEND
Erasmus Mundus Joint Masters Degree
MSc in Advanced Systems Dependability

Loria
Laboratoire lorrain de recherche en informatique et ses applications

Maynooth University National University of Ireland Maynooth
University of St Andrews
UNIVERSITÉ DE LORRAINE

Motivations

➢ **Formal Methods for Distributed Algorithms**

- Distributed Algorithms are prone to deadlocks and race conditions.
- Formal verification methods have been employed successfully to model the system and its properties and then verify its correctness.

➢ **Modeling Distributed Algorithms**

- TLA+[1] is a formal language used to describe algorithms, it is used to specify and verify complicated algorithms concisely.
- TLA+ relies on mathematical logic, PlusCal[2] was designed as an algorithm language with a more familiar syntax that can be translated into TLA+ specifications.
- PlusCal[2] lacks primitives for communication between processes which makes modeling distributed algorithms difficult.

References

[1]Lamport, Leslie. (2000). Specifying Concurrent Systems with TLA+.
[2] Lamport, Leslie. (2009). The PlusCal Algorithm Language. 5684. 36-60. 10.1007/978-3-642-03466-4_2.

Contributions

➢ We propose Distributed PlusCal, an extension of PlusCal that introduces constructs that can overcome some of the limitation of PlusCal.

➢ Distributed PlusCal offers primitives that aid in modeling distributed algorithms.

➢ We provide a backward compatible translator that translates from Distributed PlusCal and PlusCal to TLA+.

Distributed PlusCal

➢ Distributed PlusCal is a PlusCal extension that offers primitives that aid in modeling distributed algorithms such as

- Sub-processes
- Communication channels :
  - Unordered channels with the syntax

channel *⟨identifier⟩[⟨dimension1,…dimensionN⟩]*

  - FIFO channels with the syntax

fifo *⟨identifier⟩[⟨dimension1,…dimensionN⟩]*

  - Channels support the following operations
    1. send
    2. broadcast
    3. multicast
    4. receive
    5. clear

```
(***
--algorithm TwoPhaseCommit {

\* unordered channels
channels agt[Agent], coord;

  fair process (a \in Agent)
  variable aState = "unknown"; { \* sub-process

    a1: if (aState = "unknown") {
        with(st \in {"accept", "refuse"})
          aState := st;
          send(coord, [type |-> st, agent |-> self]);};};
    a2: await(aState \in {"commit", "abort"})
  } { \* sub-process
    a3:await (aState # "unknown");
       receive(agt[self], aState);
  }

  fair process (c = Coord)
  variables cState = "unknown",
            commits = {},
            msg = {}; { \* sub-process
   c1: await(cState \in {"commit", "abort"});
       broadcast(agt, [ag \in Agent|-> cState]);
  } { \* sub-process
     c2:while (cState \notin {"abort", "commit"}) {
         receive(coord, msg);
             if (msg.type = "refuse") {
                 cState := "abort";
             }
             else if (msg.type = "accept") {
                 commits := commits \cup {msg.agent};
                  if (commits = Agent) {
                     cState := "commit";}
              }
        }
    }
}***)
```