Contemporary Engineering Sciences, Vol. 3, 2010, no. 6, 249 - 257

Dynamic Live Binary Tree Distributed Mutual Exclusion Protocol

Walid J. Ghandour

Electrical and Computer Engineering Department
American University of Beirut, Beirut, Lebanon
wjg00@aub.edu.lb

Nadine J. Ghandour

Lebanese University, Lebanon
nadine.ghandour@liu.edu.lb

Abstract

In this paper we introduce, simulate and formally verify Dynamic Live binary Tree Distributed mutual exclusion protocol (DLTD). DLTD has full binary tree structure where the internal nodes are arbiter cells and the external nodes are objects that compete for the token to get access to a shared resource. The protocol guarantees liveness and features dynamic prioritization.

Keywords: Mutual exclusion protocol, liveness, formal verification

1 Introduction

Dynamic Live binary Tree Distributed mutual exclusion protocol (DLTD) has full binary tree structure where the internal nodes are arbiter cells and the external nodes are objects that compete for the token to get access to a shared resource. The protocol guarantees liveness and features dynamic prioritization.

The remaining of the paper is organized as follows. Section 2 provides a detailed description of the protocol. Section 3 discusses the implementation of the protocol using Verilog and its simulation and formal verification using VIS [1]. Section 4 presents the benefits of the protocol. Finally, we conclude in Section 5.

2 Description of the Protocol

We use two enumeration types, “boolean” and “handShakeType”, in the design to represent wires and registers used in the arbiter cell and wires that connects components with each others.

typdef enum {myTRUE, myFALSE} boolean
typdef enum {idle, request, lock, release} handShakeType

2.1 Arbiter Cell

Each arbiter cell can have up to two children. It has five inputs and three outputs. It controls the propagation of the token. It guarantees liveness by returning the token to its parent after granting the token to both children at most once. This allows the token to propagate to the root of the tree. This prevents terminal nodes in the network from starving (starvation is the situation where a terminal node makes a request for the token and never receive it).

2.1.1 Arbiter cell inputs

- clk: Clock
- topCell: of type boolean, set to myTRUE to designate the root cell
- urLeft: of type handShakeType, input from the left child
- urRight: of type handShakeType, input from the right child
- xa: of type boolean, input from the parent. The parent uses this signal to grant the token to the cell

urLeft and urRight designate the states of the left and right children. A child can be in one of the following four states: “idle, request, lock, release”.

2.1.2 Arbiter cell outputs

- uaLeft: of type boolean, output to the left child. It is set to myTRUE when the cell grants the token to its left child.
- uaRight: of type boolean, output to the right child. It is set to myTRUE when the cell grants the token to its right child.

- xr: of type handShakeType, output to the parent of the cell. The cell uses this signal to communicate with its parent cell.

2.1.3 Arbiter cell registers

The arbiter cell has the following five registers: doneLeft, doneRight, haveToken, previousLeft and previousRight.

2.1.3.1 haveToken

haveToken register is used to determine if the cell has the token. Its initial value is set to “TopCell” which is set to myFALSE to all cells except the root cell. Having TopCell set to myTRUE, the root cell initially holds the token.

2.1.3.2 doneLeft and doneRight

doneLeft and doneRight registers are both initialized to myFALSE. doneLeft (doneRight) is set to myTRUE when the left (right) child returns the token to the cell. They are reset to myFALSE when the cell returns the token to its parent. The cell will not grant the token to a child whose done register is set to myTRUE.

2.1.3.3 previousLeft and previousRight

These registers are used to keep track to which child the token was granted the last time. By initializing previousLeft to myFALSE and previousRight to myTRUE, the left child will have the first chance to hold the token in case both left and right children make a request at the same time. When the cell grants the token to a child, the previous register of this child is set to myTRUE and that of the other child is set to myFALSE.

2.1.4 Arbiter cell wires

The arbiter cell has the following four wires: ackChild, childHaveToken, childReleasedToken and returnToParent.

2.1.4.1 ackChild

ackChild is set to myTRUE when the cell grants the token to one of its children.

2.1.4.2 Children status

childHaveToken is set to myTRUE if any of the children has the token.

childReleasedToken is set to myTRUE if any of the children is in release state (has just released the token to the cell). The cell will have the token at the next cycle.

If any of haveToken, childHaveToken or childReleasedToken is set to myTRUE, the cell is in lock state with respect to its parent.

2.1.4.3 returnToParent

returnToParent is set to myTRUE when both doneLeft and doneRight are both set to myTRUE and the cell is not the root cell. This means the token was granted to both children. The cell should return the token to its parent.

2.1.5 Dynamic prioritization

If the cell has the token and the value of returnToParent is equal to myFALSE, the cell grants the token to the child which previous register is set to myFALSE in case both children are in request state at the same time. If only one child is in request state, the child in request state is granted the token regardless of the value of the previous registers.
When the cell grants the token to a child, the previous register of this child is set to myTRUE and that of the other child is set to myFALSE.

If one of the children makes a request while the other one is in idle state and the arbiter cell does not have the token and it has not passed the token to the child that makes the request during this cycle, we set the previous register of the child that makes the request to myFALSE and that of the second child to myTRUE. This guarantees that the child that makes the request first, will have the priority of getting the token before the other child. If both children make a request at the same time or both of them become in request state, no action is taken.

2.1.6 Rules for passing the token to the left (right) child

- The value of returnToParent is myFALSE
- The cell has the token (haveToken == myTRUE)
- There is a request from the left (right) child
- No request from the right (left) child or it is left (right) child turn now (i.e. previousLeft set to myFALSE and previousRight set to myTRUE (previousRight set to myFALSE and previousLeft set to myTRUE))
- doneLeft (doneRight) is set to myFALSE

The cell grants the token to the left (right) child if all the above conditions hold. uaLeft (uaRight) is set to myTRUE. ackChild is then set to myTRUE .

The last step above is used to guarantee liveness. An arbiter cell grants the token at most once to each child before it returns it to its parent. Hence, after an arbiter cell returns the token to its parent, it will not get it again before it reaches the root arbiter cell.

2.1.7 Communication with the parent cell

The value of xr is set to:

- lock: if one of the children has the token or one of the children is in release state (it has released the token in this cycle, the token is in its way to the cell. The cell will have the token at the next cycle).
- release: the cell has the token and at least one of the following is true:
  - returnToParent is set to myTRUE (the cell is not the root cell and both doneLeft and doneRight are set to myTRUE).
  - There is no request from any of the children and the cell is not the root cell.
- lock: the cell has the token and the release condition above does not hold.
- request: the cell does not have the token and there is a request from at least one of the children.
- idle: if none of the above is true.

The above conditions should be checked sequentially in the same order as they appear, following if, else if, else logic.

2.1.8 Does the arbiter have the token?

- xa == myTRUE: the cell is granted the token by its parent. haveToken is set to myTRUE.
- xr == release: the cell releases the token to its parent. haveToken is set to myFALSE.
- ackChild == myTRUE: the cell grants the token to one of its children. haveToken is set to myFALSE.
- urLeft == release || urRight == release: one of the children has released the token. haveToken is set to myTRUE.

2.1.9 Is there a token in the tree?

The root cell continuously checks if there is a token in the tree. It inserts a new token if there is no one in the tree.

There is not a token in the tree if the following two conditions are true:

- Neither the root cell nor any of its children has the token.
- No one of its children is in release state.

If there is not a token in the tree, the root cell inserts a new one by setting its haveToken register to myTRUE.

2.2 Terminal Nodes

The terminal nodes are assumed to guarantee fairness. No terminal node should hold the token for ever.

Terminal nodes need to satisfy the following fairness constraints:

!(Pi.procState=lock);

i is in {0,1,…,n-1}; n is the number of terminal nodes.
Pi designates the ith terminal node.

3 Simulation and Formal Verification

We implemented DLTD using Verilog and simulated and formally verified it using VIS [1].

3.1 Simulation

We use VIS [1] to prove that DLTD guarantee liveness for the example provided in figure 1. We use (C, A, B) to represent an arbiter cell with its children. C is the cell, A the right child and B the left child. A and B can be either arbiter cells or terminal nodes.
Figure 1 represents a complete binary tree of seven arbiter cells and eight terminal nodes: {(C0, C1, C2), (C1, C3, C4), (C2, C5, C6), (C3, P1, P2), (C4, P3, P4), (C5, P5, P6), (C6, P7, P8)}. P3 and P4 are assumed to be always in idle state. P1 is always active. There are no constraints on the state value of P2.

We simulate two different types of terminal nodes. A terminal node of the first type must enter a one cycle idle state after releasing the token. A terminal node of the second type will either enter idle or request state after releasing the token based on the value of a non deterministic variable. For the first terminal node type, P1 immediately makes a request after it enters its mandatory one cycle idle state after releasing the token. For the second type, in case it enters an idle state after releasing the token, it enters request state on the following cycle.

We simulate the example in figure 1 for an implementation of DLTD using VIS [1] and using the two different terminal node types. DLTD guarantees liveness for this example using the two different types of terminal nodes described above. An arbiter cell returns the token to its parent after granting the token to both children at most once. This prevents the token from entering an infinite path (C1, C3, P1, and P2) where C1, C3 and P1 occur infinitely often. When one or more of P5, P6, P7 and P8 nodes make a request, they are guaranteed the token.

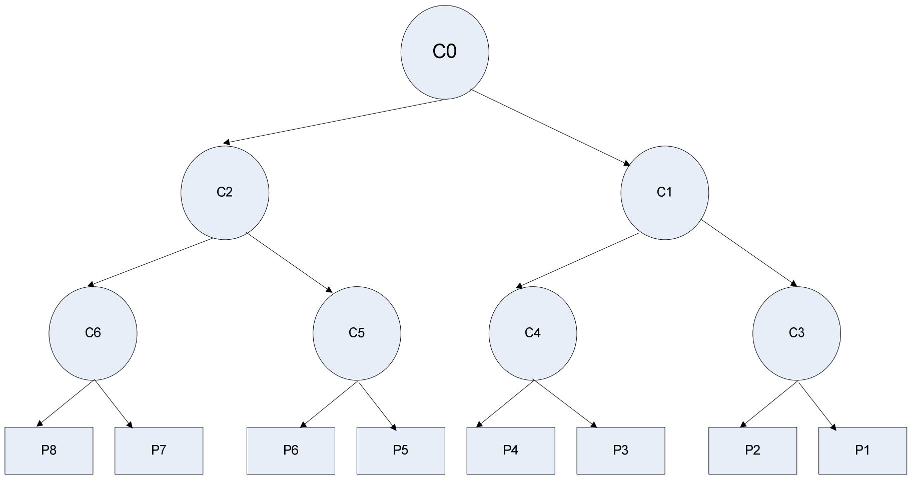

Fig 1. Complete binary tree of 14 nodes.

3.2 Formal Verification

DLTD has passed the following checking using VIS [1].

# Mutual exclusion properties
# Where i ≠ j, and Pi and Pj are any terminal nodes

*AG (! ((Pi.procState = lock) * (Pj.procState = lock)));*

# Liveness properties
#
# For liveness properties to hold, terminal nodes should satisfy the following fairness constraint

# !(Pi.procState=lock);

AG ((Pi.procState = request) -> AF (Pi.procState = lock));

We run model checking for four and eight terminal nodes. We use the two different types of terminal nodes described in section 3.1 above. DLTD has passed the checking for four and eight terminal nodes of both types.

4 Benefits of DLTD

Following are some of the benefits of DLTD:

- Dynamic prioritization.
- The path length from the root arbiter cell to any terminal node is equal to $\log_2(N+1)$ where N is the number of arbiter cells. This results in short propagation delays of the token and smaller average waiting time for external nodes relative to a token ring implementation. The maximum propagation delay is $2^* \log_2(N+1)$.
- An arbiter cell will not grant the token to a child unless the child is in request state. This prevents unnecessary delays and reduces the waiting time of external nodes in request state.
- Determinism: the maximum waiting time (worst case) of an external node can be calculated.
- Terminal nodes failure will not have any impact on the tree.

5 Conclusion

We have provided in this paper a detailed description of Dynamic Live binary Tree Distributed mutual exclusion protocol (DLTD). The protocol can be used to control asynchronous access to a shared resource. It features dynamic prioritization, liveness, determinate worst case waiting time, and smaller average waiting time relative to a token ring structure. The protocol was implemented in Verilog, simulated and formally verified using VIS [1].

References

[1] VIS: A system for Verification and Synthesis", The VIS Group, In the Proceedings of the 8th International Conference on Computer Aided Verification, p428-432, Springer Lecture Notes in Computer Science, #1102, Edited by R. Alur and T. Henzinger, New Brunswick, NJ, July 1996.

Received: May, 2010