Mini Project Model Checking
Universität Salzburg
SS 2009
LV-Leiter: Ana Sokolova

Schönegger Andreas

14. Juli 2009

Zusammenfassung

In my mini project i will try to check, if it is possible to build non blocking data structures with the hardware routine DCAS. My work is based on the paper "Two-Handed Emulation: How to build non-blocking implementations of complex data-structures using DCAS" from Michael Greenwald. I will give a short introduction into the idea and then show that it is enough to use the DCAS routine to implement a non blocking algorithm. In order to check the DCAS routine i use the model checker "Spin". For the Project i will check two properties. If the processes write the right values and second if all processes make progress.

Inhaltsverzeichnis

1 Problems and advantages of nonblocking

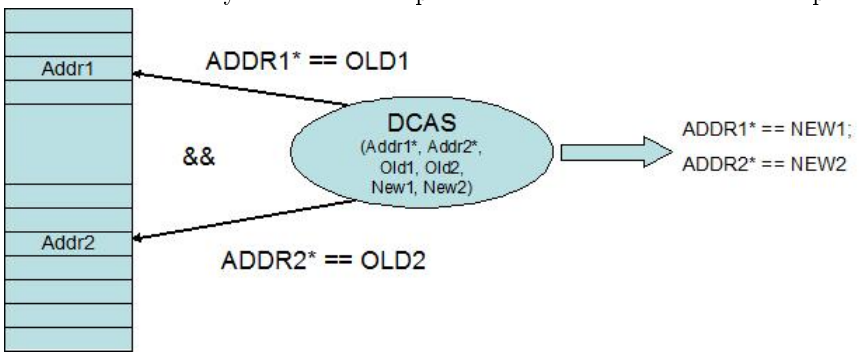
For the start i will begin with a short introduction into non blocking and why it is useful to make data structures non blocking. Therefore i will take the definition from the paper.

> "An implementation of a concurrent data structure is non-blocking if we guarantee that at least one process will make progress after a finite number of steps.
> By "make progress" we mean that it will complete a high-level operation;
> by "finite number of steps" we count the number of primitive operations in the underlying implementation."
> Michael Greenwald

Now we can look at the advantages of non blocking algorithm. When we build non-blocking data structures we can be sure that it is not possible to run in a deadlock. That´s easy to see, because there are no locks and therefore no process can be prevented from using a resource. The next advantage is that synchronisation and scheduling algorithms don´t interfere into the run of the program. As the definition grants us that at least one process will make progress it´s also not possible to run into a lifelock. But there are not only advantages of non blocking. The downside is that non blocking algorithm normally are much more complex and therefore harder to understand and not easy to debug.

2 Introduction into the Hardware routine DCAS

For the purpose to easier transform normal data structures into non blocking i will now introduce a new hardware routine. DCAS stands for double compare and swap and is an enhancement of the CAS (compare and swap) routine. Other then the DCAS is CAS already popular and in most of the systems integrated. Now to the functionality of DCAS. The pictuere below shows how DCAS operates.



As we can see in the picture DCAS got six parameters. Two addresses, two old values and two new values. The routine first compares the value of the addresses with the old values and if both of them matches it writes the two new

values into the addresses. The property we get is that either both values will be written or none.

3 Introduction into the Two-Handed Emulation

Now before i start with my implementation and tests in Spin i will summarize the two handed emulation in a view words. The two handed emulation is split into 3 steps.

- First step: If no other process have already registered a work →register yourself and put your data into the shared memory so that every process is able to complete the work.
- Secound step: Proceed the registered work until its finished.
- Third step: Bring the system into a status where a new work can be registered.

4 How i used the two-handed Emulation in spin

In order to hold the model as simple as possible. I decided not to rebuild a complex data structure. Instead I used the simplest action where a process can be interrupted: to write two variables. For the two handed emulation i need two kinds of memory. A shared memory and a private memory. I simulate them with global and local variables. The registerValue1, registerValue2 and registerID are the shared memory and are used for registering the current work. The variable state counts the current state of the system. The two finalValues are the variables we want to set.
The first part of the program is the initialising of all variables. newValues are the value of a spezific process and myID is the id of the process.

```
active[1] proctype P()
{
        byte newValue1;
        byte newValue2;
        byte myID;

        d_step
        {
                newValue1=currentValue;
                newValue2=currentValue;
                myID = idGiver;
                idGiver = idGiver + 1;
                currentValue =currentValue + 4;
        }

        byte lookedUPID = registerID;
```

After the initialisation we are coming to the working loop. Every process is trying to get his work registered and done. The registration is the first step of the two handed emulation and looks as follow. We first look if the system is in a state where a process can register himself for a work. If the system is in such a state then the CAS command is used to register the new values. The registeredValues, the registerId and the state are handled like a single object.

```
if
::state%3 == 0 ->   atomic
                    {
                    if
                    ::lookedUPID == registerID ->
                            registerValue1 = newValue1;
                            registerValue2 = newValue2;
                            registerID = myID;
                            state = state + 1;
                    fi;
                    }
```

Lets now go to the second step. If a system is in that state, the registered values will be processed. It doesn't matter what process is currently in control, he just works on the registered command.

```
::state%3 == 1 ->   byte tempValue = finalValue1;
                    atomic
                    {
                    if
                    ::lookedUPID == registerID && tempValue == finalValue1 ->
                            finalValue1 = registerValue1;
                    fi;
                    }
                    tempValue = finalValue2;

                    atomic
                    {
                    if
                    ::lookedUPID == registerID && tempValue == finalValue2 ->
                            finalValue2 = registerValue2;
                            state = state +1;
                    fi;
                    }
```

I use the last step for two things. On the one hand i use the third step to bring the system back to a state where a new proces can register himself like its used in the original system. On the other hand i use this state for my tests if the procedure was a success. Normally, every process can handle this step, but for my tests the step only can be completed by the registered process.

```
::state%3 == 2 ->   if
                    ::myID == registerID -> assert(finalValue1 == newValue1 && finalValue2 == newValue2);
                            donework[myID] = 1;
                            state = state +1;
                    fi;
```

5 Checked Properties

Now in this chapter i will summerise my final results.

5.1 Property 1: Check if the right values are written

To check my properties i use asserts.My first assert is equivalent to the LTL formula

$G((step 3\ finisched)(values\ written\ correctly)$. Values written correctly means that (finalValue1 == newValue1) && (finalValue2 == newValue2).

5.2 Property 2: Check if all processes finish their work

The secound assert looks if all processes have finished their Work. Its to asume that both properties are correct becouse they are logical and easy to see. Finally in my tests i got the proof of my asumed result. Both properties where correct.