The Theory of Deadlock Avoidance via Discrete Control

By Wang, Lafortune, Kelly, Kudlur, Mahalke

The Authors start with explaining how mutexes solve the problem of data races in multi-processing, but a side effect is the possibility of deadlocks. Debugging deadlocks and concurrency issues is hard for a programmer. The authors claim they have devised a solution where a programmer writes code worrying about data races, and can conveniently ignore the possibility of deadlock. The tool they have invented instuments the binary executable to prevent states that can cause additional deadlocks.

The authors have used petri nets to model multiprogramming systems, as it is easy to do so. Petri nets of programs are derived from their CFG (Control Flow Graph). Due to some special properties, the petri nets representing program execution have special properties that enabled the authors to tweak popular results from `Discrete Control Theory` to achieve `Maximal Permissiveness`.

In Section 2, the authors present an high level overview of the working of the tool. (Step 1) Per function CFGs are extracted. (Step 2) Translate per functions CFGs to a Petri Net model. This petri net model will have each deadlock in the original program. (Step 3) Add some control logic to prevent deadlocks. The output of this step is a petri net that refelcts a program with all its deadlocks. (Step 4) Instrument the code to reflect the additional control logic added in the petri net in step 3. The authors then discuss why their method has a low performance overhead.

A small introduction to Petri Nets is provided to the reader. A couple of examples are provided to the user that explain common programming paradigms like branching, loops, function calls etc. The authors then explain how to convert the CFG of a multiprogramming program to a Petri Net. They also discuss how to prune the Petri Net representation, by pruning CFG nodes which do not use any locks. They also explain how recursion is modelled as a for loop. The authors also discuss how to model mutex locks extensively. The authors also introduce us to the concept of Petri Net Invariants, and how to use the theorem by Iordache and Antsaklis to make sure the Petri Net always satisfies that invariant condition. This theorem is central to making the changes in the net representing the CFG to prevent deadlocks.

The authors explain the next concept of Petri Net Siphons. The next theorem that is introduced states that `A totally deadlocked petri net contains atleast one **empty** siphon`. Continuing with the flow, the next theorem makes a correspondence between deadlock avoidance in a program and its corresponding Petri Net. It states that `The problem of deadlock avoidance in a concurrent program is equivalent to the problem of avoidance of empty siphons in its ordinary

petri nets model`. This means that if we solve the deadlock avoidance problem for a petri net, we can solve it for the corresponding program.

The author proceeds to find the invariant to avoid the deadlock problem. The invariant is that the sum of tokens in the places in the minimal siphon state should not be 0, but greater than or equal to 1. This can be perfectly captured by an inequality of the type that is solved by the theorem proved by Iordache and Antsaklis. The result is we know how to add a control state for each siphon. The author mentions the possibility of the control places creating a siphon, but that can be solved by repeating the process. The deadlock free petri net now generated has maximally permissible behaviour, according to the same theorem.

The author mentions that they in practice did not encounter any non unit weight control arcs added by this method. These experiments were conducted on both, real world programs as well as randomly generated programs.

The authors move to how the control places will be implemented. They show that the control places can be implemented with locks and count variable. The next theorem basically proves that threads can never get deadlocked while acquiring the control ‘locks’. Formally, it is stated as `With the implementation of control places and global ordering of all control locks (i.e control locks are acquired only in a particular order, if required), at least one thread will get all required locks from the control places and succeed in firing the corresponding transition`
This theorem paves the way to implementing the control places. Nothing other than locks and arrays for maintaining count are required.

The authors move on to the model extensions. They states how semaphores, reader writer locks, can be modelled using Petri Nets. They explain partial controllability and observability, and why it should be handled. They talk about other shortcomings with this method and possible solutions. A whole section after this is devoted to practical implementation strategies. The paper ends with experimental results, followed by Related Work.