Control Flow Model 1

```c
main(...)
{
    setuid(1000);
    log(...);
    system(...);
    setuid(0);
    log(...);
}
```

A model can be built based on this program that will look as shown on the right.

**Problem:** If an IDS uses this model, then it allows a loop such that `system()` can be called with uid = 0.

**Problem:** This kind of loop does not exist in the source program, i.e. the program does not allow a user to run `system()` as root, but the model does allow it.

**Problem:** It is not possible to replicate the `log(...)` function each time in the model owing to space complexity issues.

**Solution:** To use a PDA [Push Down Automaton]
PDA-based Model

A sample edge in a PDA looks like:

\[ A \xrightarrow{\alpha/F/G} B \]

This means the following:

1. On input $\alpha$
2. Pop ‘F’ from the stack top
3. Push ‘G’ onto the stack

At any given time, the IDS now needs to keep track of the state and stack to check program behaviour. The new model now looks as shown.

Note: The model now solves the problem of the loop and prevents root from calling $\text{system}()$. However, the model still faces issues regarding:

1. If conditions - the model will have to do ‘state bifurcating’ where it needs to keep track of 2 copies of each pair of ambiguous transitions. This can quickly grow exponentially.
2. $\text{system}$ arguments
3. Functions get called a lot, which may cause the graph to become ‘dense’.

Solution: The IDS needs more information about function calls, so that it need not indulge in bifurcation of states. → Dyke Model.
Dyke Model

Tell the IDS what to do with the stack. This is done by modification of the source program by adding a null function.

The modified program will look like:

```c
main(...)
{
    setuid(1000);
    null(2);
    log(...);
    null(-2);
    system(...);
    setuid(0);
    null(5);
    log(...);
    null(-5);
}
```

The Non-Determinism is now effectively removed.