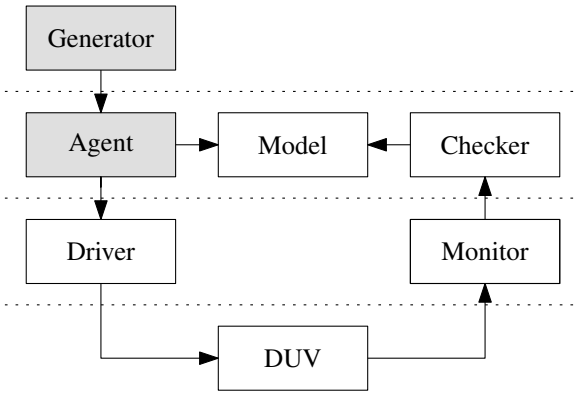
1 Constrained Random Stimulus

Das *Coverage* -Ziel wird Entwicklern meist durch das Management vorgegeben und liegt dadurch in fast allen Fällen sehr nah an 100%. Durch einen gerichteten Test ist dies bei komplexen Entwürfen meist kaum möglich. Eine Abhilfe für dieses Problem besteht in der automatisierten Generierung der Testfälle.

Stellen Sie sicher, dass die (*Functional* !) *Coverage* -Regeln aus der letzten Übung richtig in Ihre PROL16-Verifikationsumgebung aus den letzten Übungen integriert sind. Erweitern Sie außerdem die Klasse `Prol16Opcode` um *Constraints* , sodass nur gültige Opcodes und Registeradressen generiert werden.

Passen Sie Ihre Verifikationsumgebung nun so an, dass anstatt des gerichteten Tests Befehle randomisiert und dann auf dem Modell und dem DUV ausgeführt werden:



Simulieren Sie so eine gewisse Anzahl von Befehlen und dokumentieren Sie die erreichte *Functional Coverage* ! Erhöhen Sie die Anzahl der simulierten Befehle solange, bis Sie die maximal mögliche *Coverage* erreichen – wieviele Befehle werden dafür (ungefähr) benötigt? Erstellen Sie ein Diagramm, in dem Sie die erreichte Coverage über die Anzahl der simulierten Befehle darstellen. Nehmen Sie dazu mindestens zehn Messpunkte auf. Erklären Sie
□ den Kurvenverlauf!

□ Beantworten Sie die folgenden Fragen:

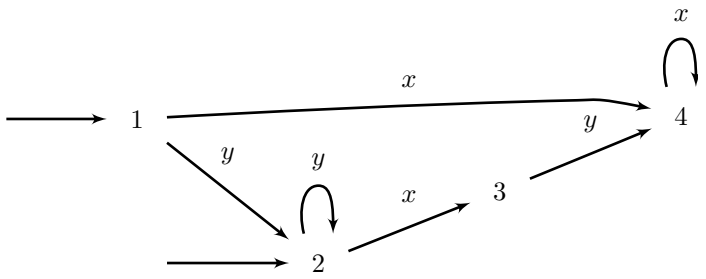
- Sind die randomisierten Werte vorhersehbar? Begründen Sie Ihre Antwort! Welchen Vorteil haben/hätten vorhersehbare Werte?
- Welcher mathematischen Funktion folgt die erreichte *Functional Coverage* in Abhängigkeit der Anzahl der durchgeführten Tests? Begründen Sie diesen Verlauf!
- Wie würden Sie ihre *Constraints* der Klasse `Prol16Opode` wählen, wenn:
  - 50% der generierten Registeradressen 0 sein sollen?
  - die beiden Registeradressen für **NOP** immer 0 sein müssen?
- Was ist der Unterschied zwischen `rand` und `randc`?
- Wieviele Befehle werden mindestens benötigt, um das folgende Covarage-Ziel zu erreichen?
  - Jeder Befehl soll mit jeder Kombination von Carry und Zero ausgeführt werden und jede dieser Ausführungen soll jede mögliche Kombination von Carry und Zero erzeugen. Beachten Sie dabei, dass nicht jeder Befehl jedes Flag beeinflusst – **AND** setzt bspw. das Carry-Flag nie, diese Kombinationen sollen also nicht berücksichtigt werden. Die Befehle **LOAD**, **STORE** und **SLEEP** sollen ebenfalls nicht berücksichtigt werden.

  Sie können dabei vereinfachend annehmen, dass vor jedem Befehl alle Registerinhalte und Flags beliebig verändert werden können.

  Wieviele Befehle benötigt Ihr Test aus dieser Übung? Erklären Sie, warum die Redundanz auftritt!

2 Modellierung

1. Gegeben sei das im Folgenden abgebildete LTS. Geben Sie die formale Darstellung des LTS als 4-Tupel, inklusive seiner Komponenten, an!



2. Konstruieren Sie eine Kripke-Struktur zu dem oben gegebenen LTS!

> *"The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at and repair."*
>
> Douglas Adams