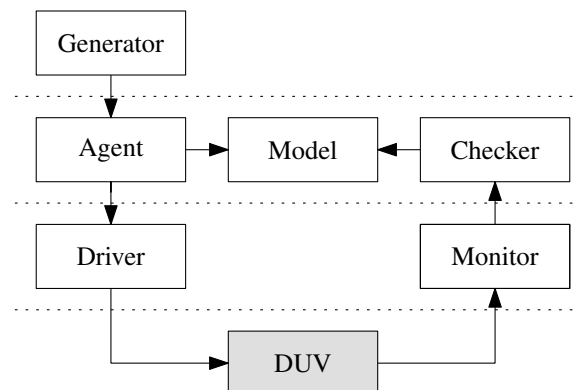


1 P(S + RO)L

Sie haben im Laufe dieses Semesters bereits eine vollständige Verifikationsumgebung für Ihren PROL16 entworfen. Während diese (hoffentlich) im Stande ist, den Großteil der Fehler im DUV zu finden, so unterliegt sie nach wie vor zwei Einschränkungen:

- Das Finden von Corner Cases ist nicht mehr von Ihrer Phantasie, sondern von der Güte des Random Number Generators und der Anzahl der Testfälle abhängig. Trotzdem führt der Constrained-Random-Test zwar *viele*, aber eben nicht *alle möglichen* Testfälle durch – die Functional Coverage kann unvollständig sein, Corner Cases können immer noch unentdeckt bleiben.
- Das DUV wird in der Verifikationsumgebung mehr oder weniger als Black Box betrachtet. Mehr oder weniger, da der Zustand aus dem Inneren des DUVs ausgelesen wird: dies ist jedoch nur notwendig weil das DUV sonst keine nennenswerten Ausgänge bietet. Sollte die Verifikationsumgebung einen Fehler finden, so lässt sie dennoch nur wenig Rückschlüsse auf die Ursache des Fehlers zu.



Assertions in DUV können in beiden Punkten helfen: im zweiten trivialerweise, da sie im Inneren des DUVs arbeiten und damit die Ursache des Fehlers ohne weiteres Zutun eingrenzen. Im ersten Punkt kann formale Verifikation ansetzen, für die eine formale Spezifikation des gewünschten Verhaltens notwendig ist.

Ziel dieser Übung ist es also, Ihren PROL16 mit PSL-Assertions auszustatten. Sinnvoll ist dies einerseits bei der ALU, da dort wahrscheinlich der Großteil der Berechnungsfehler

entsteht, und in der Core Control, da hier die meisten Steuerungsfehler entstehen werden. Natürlich könnte beispielsweise auch der Datenpfad mit Assertions versehen werden, für diese Übung sollen aber ALU und Core Control ausreichend sein.

Hinweise zur ALU: PSL benötigt einen Takt, um die Eigenschaften zum richtigen Zeitpunkt auswerten zu können. Ein rein kombinatorischer Entwurf wie die ALU ist daher schwer beschreibbar: wann ist das Ergebnis richtig? Wie hoch ist die benötigte Verzögerung? Da die ALU im PROL16 natürlich trotz allem einem Takt unterworfen ist, empfiehlt es sich für diese Übung, diesen Takt einfach zusätzlich in die ALU zu führen, auch wenn er dort nur für die Verifikation notwendig ist.

Hinweise zur Core Control: Sie werden bemerken, dass die Assertions für viele Befehle immer wieder ähnliche „Bauteile“ benötigt. Diese können Sie durch Sequences (die ggf. nur einen Zeitschritt dauern) mit Parametern einfach wiederverwenden; so ist beispielsweise die folgende Sequence im gegebenen Fall möglicherweise hilfreich:

```
1 sequence opcode_start (boolean is_opc) is {  
2     prev(mem_rd_stb_o = '1')  
3     and prev(clk_en_op_code_o = '1')  
4     and is_opc  
5 };
```

Was sollen die Assertions prüfen? Diese Frage ist in den meisten Entwürfen eine der am schwierigsten zu beantwortenden. Im gegebenen Fall bietet sich an, in der ALU des Ergebnis aller Operationen sowie die Berechnung des Carry- und des Zero-Flags zu prüfen. In der Core Control sollte die eigentliche Aufgabe der Core Control, nämlich das Setzen von bestimmten Steuerleitungen zu bestimmten Zeitpunkten, abhängig vom aktuellen Befehl, durch Assertions erfasst werden. Zusätzlich liegt es nahe, auch hier die Behandlung des Carry- und des Zero-Flags zu prüfen. Begründen Sie in jedem Fall, warum sie welchen Teil des Entwurfs getestet haben!

2 Theorie

- Erklären Sie Vacuity! Warum ist dieses Konzept wichtig? Ist es auch für andere (temporale) Logiken wichtig ist, oder ist es nur in PSL notwendig?
- Geben Sie für die folgenden Assertions je mindestens einen Signalverlauf an, für den die Assertion *vacuously* hält!

```
- assert always (x -> y);  
- assert always (x -> next y);  
- assert always ({x; y} |-> {z});  
- assert always ({x[*]; y} |-> {z});
```

- Erklären Sie, warum der Operator **eventually**! (für eine Simulation) ein starker

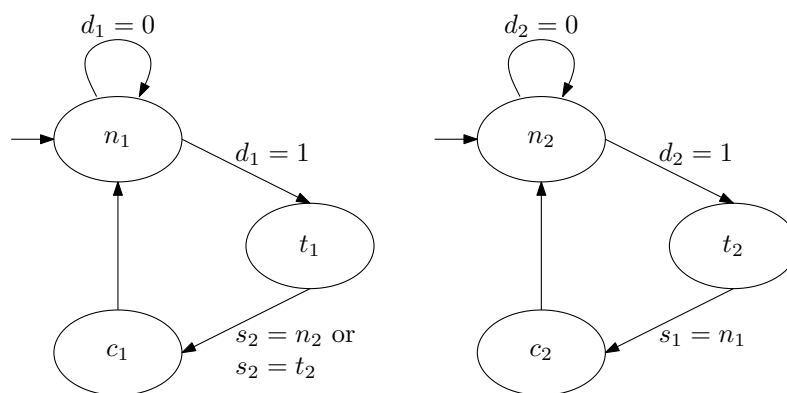
Operator sein muss, bzw. warum eine schwache Version davon in einer Simulation wenig Sinn ergeben würde!

- Beschreiben Sie den Algorithmus für Model Checking für Invarianten mit eigenen Worten!

3 Model Checking: Anwendung

3.1 Mutex

Ähnlich wie in der Vorlesung seien die beiden folgenden, kommunizierenden Automaten gegeben:



Diese Automaten sollen einen gegenseitigen Ausschluss modellieren. Die Zustände sind dabei mit n für “noncritical”, t für “trying” und c für “critical” bezeichnet. Die Übergänge $n_i \rightarrow n_i$ und $n_i \rightarrow t_i$ werden in Abhängigkeit der Eingänge d_i gewählt: solange $d_i = 0$ ist bleibt der Mutex im Zustand “noncritical”.

Hinweis: Sie können die beiden Automaten zu einem zusammenfassen, indem Sie, ähnlich wie in einer Produktautomatenkonstruktion, einen Automaten konstruieren, in dem die Zustandsmenge gleich dem kartesischen Produkt der Zustandsmengen der beiden Automaten ist.

Beweisen Sie die durch den Algorithmus zum Prüfen von Invarianten, dass die folgende Eigenschaft gilt:

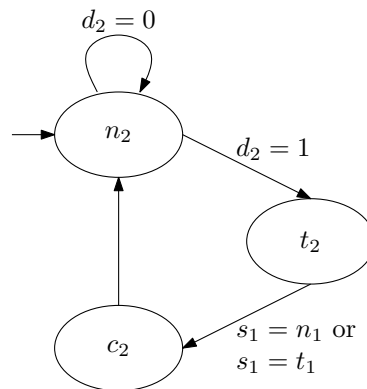
- `never (s1=c1 and s2=c2);`

Prüfen Sie zusätzlich durch Betrachten des Zustandsraums, ob die beiden folgenden Eigenschaften gelten:

- `always (s1=t1 -> eventually! s1=c1);`
- `always (s2=t2 -> eventually! s2=c2);`

3.2 Mutex, Implementierung 2

In einer Weiterentwicklung des Produkts wurde die Implementierung des Mutex, um einen besseren Durchsatz zu erreichen, leicht modifiziert. Der zweite Automat ist nun wie folgt implementiert:



Zeigen Sie, ob die obigen drei Eigenschaften immer noch gelten! Wenn nicht, geben Sie ein Gegenbeispiel an!

"If debugging is the process of removing bugs, then programming must be the process of putting them in."

Edsger W. Dijkstra