Outline

1. Introduction
2. From programs to Kripke structures
3. Temporal Logic Model Checking
4. SPIN
INFO: Tips for Windows NT Driver Developers — Things to Avoid

Following are some tips for creating Windows NT device drivers. The tips presented apply to all technologies. You can also use this as a checklist for troubleshooting driver problems.
1. Never return STATUS_PENDING from a dispatch routine without marking the I/O request packet (IRP) pending (IoMarkIrpPending).

2. Never call KeSynchronizeExecution from an interrupt service routine (ISR). It will deadlock your system.

3. Never set DeviceObject->Flags to both DO_BUFFERED_IO and DO_DIRECT_IO. It can confuse the system and eventually lead to fatal error. Also, never set METHOD_BUFFERED, METHOD_NEITHER, METHOD_IN_DIRECT or METHOD_OUT_DIRECT in DeviceObject->Flags, because these values are only used in defining IOCTLs.

4. Never allocate dispatcher objects from a paged pool. If you do, it will cause occasional system bugchecks.

5. Never allocate memory from paged pool, or access memory in paged pool, while running at IRQL >= DISPATCH_LEVEL. It is a fatal error.

6. Never wait on a kernel dispatcher object for a nonzero interval at IRQL >= DISPATCH_LEVEL. It is a fatal error.

7. Never call any function that causes the calling thread to wait directly or indirectly while executing at IRQL >= DISPATCH_LEVEL. It is a fatal error.

8. Never lower the interrupt request level (IRQL) below the level at which your top-level routine has been invoked.

9. Never call KeLowerIrql() if you haven’t called KeRaiseIrql().

10. Never stall a processor (KeStallExecutionProcessor) longer than 50 microseconds.

11. Never hold any spin locks longer than necessary. For better overall system performance, do not hold any system-wide spin locks longer than 25 microseconds.

12. Never call KeAcquireSpinLock and KeReleaseSpinLock, or KeAcquireSpinLockAtDpcLevel and KeReleaseSpinLockFromDpcLevel, while running at IRQL greater than DISPATCH_LEVEL.

13. Never release a spin lock that was acquired with KeAcquireSpinLock by calling KeReleaseSpinLockFromDpcLevel, because the original IRQL will not be restored.
Never call KeAcquireSpinLock and KeReleaseSpinLock or any other routine that uses an executive spin lock from an ISR or SynchCritSection routine(s).

Never forget to clear DO_DEVICE_INITIALIZING flag when you create a device object in a routine other than DriverEntry.

Never queue a deferred procedure call (DPC) object (using KeInsertQueueDpc) with multiple threads on different processors simultaneously. It can lead to fatal error.

Never deallocate a periodic timer from a CustomerTimerDPC routine. You can deallocate nonperiodic timers from a DPC routine.

Never pass the same DPC pointer to KeSetTimer, or KeSetTimerEx (CustomTimerDpc) and KeInsertQueueDpc (CustomDpc), because it causes race conditions.

Never call IoStartNextPacket while holding a spin lock. It can deadlock your system.

Never call IoCompleteRequest while holding a spin lock. It can deadlock your system.

Never call IoCompleteRequest without setting the completion routine to NULL if your driver sets the completion routine.

Never forget to set the I/O status block in the IRP before calling IoCompleteRequest.

Never call IoMarkPending after queuing an IRP or sending it to another driver (IoCallDriver). The IRP may be completed before the driver calls IoMarkPending and a bugcheck might occur. For drivers with completion routines, the completion routines must call IoMarkPending if Irp->PendingReturned is set.

Never touch an IRP after you have called IoCompleteRequest on it.

Never call IoCancelIrp on an IRP that is not owned by your driver unless you know that the IRP has not been completed yet.

Never call IoCancelIrp for the IRP that your dispatch routine is working on until your dispatch routine returns to caller.

Never call IoMakeAssociatedIrp to create IRPs for lower drivers from an intermediate driver. The IRP you get in your intermediate driver could be an associated IRP, and you cannot associate other IRPs to an already associated IRP.
Never call IoMakeAssociatedIrp on an IRP that is set up to perform buffered I/O.

Never simply dereference virtual pointers to device I/O registers and access them. Always use correct hardware abstraction layer (HAL) functions to access a device.

Never access IRP or device object fields from an ISR that may be modified from DISPATCH_LEVEL. On a symmetric multiprocessor system this can cause data corruption.

Never modify data while running at high-IRQL if that data may be written by low-IRQL code. Use the KeSynchronizeExecution routine.

Never acquire one of the driver’s own spin locks (if you have any) in your DispatchCleanup routine, before acquiring the system-wide cancel spin lock (IoAcquireCancelSpinLock). Following a consistent lock acquisition hierarchy throughout your driver is essential to avoiding potential deadlocks.

Never call IoAcquireCancelSpinLock in your cancel routine because it is always called with the system cancel spin lock held on its behalf.

Never forget to call IoReleaseCancelSpinLock before returning from a cancel routine.

Never use IRQL-based synchronization because this works only on single processor systems. Raising IRQL on one processor does not mask interrupts on other processors.

Never use RtlCopyMemory for overlapped memory address ranges. Use RtlMoveMemory.

Never assume page sizes are constant, even for a given CPU. Use PAGE_SIZE and other page related constants defined in header files to maintain portability.

Never access any registry keys other than Registry\Machine\Hardware and Registry\Machine\System from DriverEntry routine of a driver loaded in Boot\System Initialization phase.

Never create an Enum key for loading a driver under a driver’s registry key (Registry\Machine\System\CurrentControlSet\Services). The system creates this key dynamically.

Never attempt to initialize a physical device without claiming the necessary bus-relative I/O ports, memory ranges, interrupt, or direct memory access (DMA) channel/port hardware resources in the registry first.

Never call IoRegisterDriverReinitialization from your DriverEntry routine unless it returns STATUS_SUCCESS.
42 Never call KeSetEvent with the Wait parameter set to TRUE from a pageable thread or pageable driver routine that runs at IRQL PASSIVE_LEVEL. This type of call causes a fatal page fault if your routine happens to be paged out between the calls to KeSetEvent and KeWait..Object(s).

43 Never call KeReleaseSemaphore with the Wait parameter set to TRUE from a pageable thread or pageable driver routine that runs at IRQL PASSIVE_LEVEL. If your routine happens to be paged out between the calls to KeReleaseSemaphore and KeWait..Object(s), this type of a call causes a fatal page fault.

44 Never call KeReleaseMutex with the Wait parameter set to TRUE from a pageable thread or pageable driver routine that runs at IRQL PASSIVE_LEVEL. If your routine happens to be paged out between the calls to KeReleaseMutex and KeWait..Object(s), this type of a call causes a fatal page fault.

45 Never call KeBugCheckEx or KeBugCheck from a retail Windows NT driver to bring down the system, unless the error encountered is a critical error which would corrupt system memory or eventually cause the system to bugcheck. Always try to handle error conditions gracefully.

46 Never assume that an IoTimer routine will be called precisely on a one-second boundary because the intervals at which any particular IoTimer routine is called ultimately depends on resolution of the system clock.

47 Never call Win32s application programming interfaces (API) from a kernel-mode device driver.

48 Never use recursive functions that can cause the stack to overflow because the calling thread’s kernel-mode stack does not grow dynamically while it is running in kernel-mode.

49 Never use interrupt object pointers (PKINTERRUPT) to identify interrupts in an ISR that handles more than one interrupt, because the address of the interrupt object you get in the ISR will not always be the same as the one you got from IoConnectInterrupt. You should only use the ServiceContext value that you specify in IoConnectInterrupt to identify the current interrupting device.

50 Never unload a driver without clearing CustomTimerDpc (KeCancelTimer). If the DPC is fired after the driver is unloaded, it could hit non-existent-code and cause the system to bugcheck.

51 Never unload a driver until all the IRPs that have the I/O CompletionRoutine of the driver set in it are completed. If the IRP gets completed by the lower driver after your driver is unloaded, the system could try to execute the non-existent code and cause the system to crash.
Never enable device interrupt until your driver is ready to handle it. You should enable only after your driver is completely initialized, and it is safe for the system to touch the driver’s internal structures in ISR and DPC.

Never call outside of your driver while holding a spinlock because it can cause deadlock.

Never return any status other than STATUS_MORE_PROCESSING_REQUIRED from your I/O CompletionRoutine for an IRP created by your driver with IoBuildAsynchronousFsdRequest/IoAllocateIrp because the IRP is not prepared for completion related post-processing by the I/O manager. Such an IRP should be freed explicitly (IoFreeIrp) by the driver. If the IRP is not meant for reuse, it can be freed in the CompletionRoutine before returning status STATUS_MORE_PROCESSING_REQUIRED.

Never allocate an IRP with IoBuildSynchronousFsdRequest/IoBuildDeviceIoControlRequest in an Arbitrary thread context because the IRP remains associated with the thread (Irp->ThreadListEntry) until it is freed.

Never call IoInitializeIrp on an IRP that has been allocated with IoAllocateIrp with ChargeQuota parameter set to TRUE. When you allocate an IRP with ChargeQuota set to TRUE, the I/O manager keeps the information about the pool from which it allocated the memory for the IRP in the IRP’s internal flag.

When you call IoInitializeIrp on such an IRP, the allocation pool information is lost as this function blindly zeros the entire IRP. This leads to memory corruption when you free the IRP. Also, never reuse an IRP that comes from the IO manager. If you want to reuse an IRP, you should allocate your own by using IoAllocateIrp.

Never specify WaitMode as UserMode in KeWaitForSingleObject/KeWaitForMultipleObjects if the Object is allocated in the calling thread’s stack. The corollary of this is that if the Object being waited on is created in the function stack, you must specify KernelMode as the WaitMode to prevent the thread stack from being paged out.

Never acquire resources such as ERESOURCES and FastMutex(Unsafe) in the context of a user-mode thread without protecting the code in a critical section.

Because the acquisition of these resources does not raise the IRQL to APC_LEVEL, if the thread is suspended (done by queuing an APC) after it has acquired the resource, it could cause deadlock and compromise system security. Therefore, you should acquire such resources either by explicitly raising the IRQL to APC_LEVEL or in a critical section by calling KeEnterCriticalRegion.
Software model checking

- Algorithmic analysis of programs to prove properties of their executions
- Target programs usually continuously running, concurrent
  - device drivers, protocols, operating systems, controller
  - non-deterministic
- Based on *temporal logic* [Pnueli 1977]

---

Basic idea

- Applies to finite state systems
- A property can be verified by checking that it holds in each state
  ⇒ Exhaustive search of the state space
  - State space explosion problem an issue, but many techniques have allowed model checking to scale to practical tasks
- Always produces a yes/no answer for the question of whether a property holds.
- A no-answer justified with a counter example
Temporal Logic Model Checking

- Model: finite state machine
- Property: statement written in *propositional temporal logic*

Independently developed by Clarke and Emerson 1981; Quielle and Sifakis 1982.
A model is characterized by
- **states**
  - a “snapshot” of the system. Values of all the variables of the system at some instance of time
- **transitions**
  - a possible transformation from one state to another
  - a transition is thus simply a pair of states

A *computation* is an infinite sequence of states, where each state is obtained from the previous state by some transition

Expressed as *Kripke structures*
- a kind of state transition graph
Practical view

1. System expressed as program text.
2. Extract (preferably mechanically) a specification of the system as formulas in first-order logic.
3. Given the formulas, extract a Kripke structure that models the system.
4. Check state space.
Kripke structure

Definition (Kripke structure)

Let $AP$ a set of atomic propositions. A *Kripke structure* $M$ over $AP$ is a four-tuple $M = (S, S_0, R, L)$ where

1. $S$ is a finite set of states.
2. $S_0 \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is a *total* transition relation.
4. $L : S \rightarrow 2^{AP}$ that maps each state to the set of atomic propositions that are true in that state.

- A relation $R \subseteq S \times S$ is total if $s \in S \implies \exists s' \in S. R(s, s')$
- A *path* in $M$ from a state $s$ is an infinite sequence of states $\pi = s_0s_1s_2\ldots$, such that $s_0 = s$ and $\forall i \geq 0. R(s_i, s_{i+1})$
States as formulas

- Assume $V = \{v_1, \ldots, v_n\}$ is the set of system variables
- State is then a *valuation* for $V$
  - a function $s : V \rightarrow D$ that associates a value (in some domain $D$) with each variable $v$ in $V$
- Example:
  - $V = \{v_1, v_2, v_3\}$
  - $s = \langle v_1 \leftarrow 2, v_2 \leftarrow 3, v_3 \leftarrow 5 \rangle$

A valuation gives rise to a formula, that is true in exactly that valuation.

$v_1 = 2 \land v_2 = 3 \land v_3 = 5$ gives rise to the formula

Convention: a formula represents the set of all valuations in which it is true.
States as formulas

- Assume \( V = \{v_1, \ldots, v_n\} \) is the set of system variables
- State is then a \textit{valuation} for \( V \)
  - a function \( s : V \rightarrow D \) that associates a value (in some domain \( D \)) with each variable \( v \) in \( V \)
- Example:
  - \( V = \{v_1, v_2, v_3\} \)
  - \( s = (v_1 \leftarrow 2, v_2 \leftarrow 3, v_3 \leftarrow 5) \)

- A valuation gives rise to a formula, that is true in exactly that valuation
  - \( v_1 = 2 \land v_2 = 3 \land v_3 = 5 \)
States as formulas

- Assume \( V = \{v_1, \ldots, v_n\} \) is the set of system variables
- State is then a *valuation* for \( V \)
  - a function \( s : V \rightarrow D \) that associates a value (in some domain \( D \)) with each variable \( v \) in \( V \)
- Example:
  - \( V = \{v_1, v_2, v_3\} \)
  - \( s = \langle v_1 \leftarrow 2, v_2 \leftarrow 3, v_3 \leftarrow 5 \rangle \)

- A valuation gives rise to a formula, that is true in exactly that valuation
  - \( v_1 = 2 \land v_2 = 3 \land v_3 = 5 \)

- Convention: a formula represents the set of *all* valuations in which it is true
  - \( \Rightarrow \) a formula defines a set of states
Transitions as formulas

• Idea: Two sets of variables
  • $V$ - variables in current state
  • $V'$ - variables in next state
  • For each variable $v \in V$, there is a corresponding variable $v' \in V'$

• Then, transition relation can be written as a first order formula, call it $R$
Transitions as formulas

- Idea: Two sets of variables
  - $V$ - variables in current state
  - $V'$ - variables in next state
  - For each variable $v \in V$, there is a corresponding variable $v' \in V'$

- Then, transition relation can be written as a first order formula, call it $R$

- Kripke structure that we get is then:
  - Set of states $S$ is the set of all valuations for $V$
  - For two states $s$ and $s'$, $R(s, s')$ holds if $R$ evaluates to true when each $v \in V$ is assigned the value $s(v)$ and each $v' \in V'$ the value $s(v')$. If for some state $s$ has no successor, $R$ is extended so that $R(s, s)$ holds (to make it total).
  - Labeling function $L : S \rightarrow 2^{AP}$ is defined so that $L(s)$ is the subset of all atomic propositions true in $s$
Example: from system to formulas

System description

- $V = \{x, y\}, D = \{0, 1\}$
- One transition:
  - $x := (x + y) \mod 2$
- Start state
  - $x = 1$ and $y = 1$

First-order formulas
Example: from system to formulas

System description

- $V = \{x, y\}, D = \{0, 1\}$
- One transition:
  - $x := (x + y) \mod 2$
- Start state
  - $x = 1$ and $y = 1$

First-order formulas

- Set of initial states
  - $S_0(x, y) \equiv x = 1 \land y = 1$
- Set of transitions
  - $\mathcal{R}(x, y, x', y') \equiv x' = (x + y) \mod 2 \land y' = y$
Example: from formulas to Kripke structure

First-order formulas

- Set of initial states
  \[ S_0(x, y) \equiv x = 1 \land y = 1 \]
- Set of transitions
  \[ \mathcal{R}(x, y, x', y') \equiv x' = (x + y) \mod 2 \land y' = y \]

Kripke structure \( M = (S, S_0, R, L) \)

- \( S = D \times D \) (value of \( x \) at first element, \( y \) at second)
- \( S_0 = \{(1, 1)\} \)
- \( R = \{(1, 1), (0, 1), (1, 1), (0, 1), (1, 0), (1, 0), (0, 0), (0, 0)\} \)

\[
\begin{align*}
L((0, 0)) &= \{x = 0, y = 0\} \\
L((0, 1)) &= \{x = 0, y = 1\} \\
L((1, 0)) &= \{x = 1, y = 0\} \\
L((1, 1)) &= \{x = 1, y = 1\}
\end{align*}
\]

- the only path starting from a state in \( S_0 \) is
  \[(1, 1) (0, 1) (1, 1) (0, 1) \ldots \]
Granularity of transitions

- Transitions should be **atomic**
  - no observable state of the system can result in executing a *part* of a transition
- Non-atomic transitions may lead to a Kripke structure that omits observable states
  - Then some errors may be impossible to find
- Too fine grained granularity also a problem
  - Possible to find spurious errors
  - Also more expensive
- Compare these transitions

\[
\alpha : \ x \leftarrow x + y \\
\beta : \ y \leftarrow y + x
\]

with these

\[
\alpha_0 : \text{load } R_1, x \\
\alpha_1 : \text{add } R_1, y \\
\alpha_2 : \text{store } R_1, x \\
\beta_0 : \text{load } R_2, y \\
\beta_1 : \text{add } R_2, x \\
\beta_2 : \text{store } R_2, y
\]
Outline

1. Introduction

2. From programs to Kripke structures

3. Temporal Logic Model Checking

4. SPIN
Model checking answers the question whether

\[ M \models P \]

That is, does \( M \) satisfy \( P \)?
Big picture

- Model checking answers the question whether
  \[ M \models P \]

- That is, does \( M \) satisfy \( P \)?
  - \( M \) ?
  - \( P \) ?
  - \( \models ? \)
$M$ is a Kripke structure
- **states**: valuations to all variables
- **initial states**
- **edges**: transitions between states
- **atomic propositions**
Semantics of state machines

Unfold the state transitions graph to produce an infinite tree

- A trace or a path is an infinite sequence of states
- The semantics is the set of possible traces
Formula in some temporal logic
Temporal Logic

- We have seen atomic propositions describing properties of states.
- And logical connectives, such as $\land$, $\lor$, $\implies$, $\neg$ describing more complex properties of states.
- However, in reactive systems, the interest is also in describing properties of transitions between states, or sequences of states, or paths.
- Example properties:
  - *eventually* a particular state is reached
  - an error state is *never* entered
  - a property holds in *all* states reachable from an initial state
- Properties on sequences of states expressed using temporal operators.
- In temporal logics “traditional” logical connectives can be freely mixed temporal operators.
CTL* == Computational Tree Logic

- CTL* formulas describe properties of computational trees.
- Tree’s formed by starting from an initial state of a Kripke model, and unwinding the transition relation into an infinite tree.
- The paths in the tree are all the possible computations that start from the designated initial state.
- Formulas composed using:
  - path quantifiers
  - temporal operators
Path quantifiers

- Describe the branching structure
- Two quantifiers \( A \) and \( E \)
  - \( A \ p \)
    - All of the paths starting from a given state have property \( p \)
  - \( E \ p \)
    - Some path starting from a given state has property \( p \)
Temporal operators

- Describe properties of an individual path
- Five temporal operators
  - \( X \ p \)
    - \( p \) holds in the “neXt” state
  - \( F \ p \)
    - \( p \) holds “eventually” (in the Future)
  - \( G \ p \)
    - \( p \) holds “always” (Globally)
  - \( p \ U \ q \)
    - \( q \) holds for some state, and \( p \) holds in all states “Until” that state
  - \( p \ R \ q \)
    - \( q \) holds up to and including a state where \( p \) is true (if ever)
    - \( p \) “Releases” \( q \)
Two kinds of formulas

- State formulas
- Path formulas
State formulas

- If $p \in AP$, then $p$ is a state formula.
- If $f$ and $g$ are state formulas, then $\neg f$, $f \land g$, and $f \lor g$ are state formulas.
- If $f$ is a path formula, then $Ef$ and $Af$ are state formulas.
Path formulas

- If $f$ is a state formula, $f$ is also a path formula.
- If $f$ and $g$ are path formulas, then the following are path formulas:
  - $\neg f$
  - $f \land g$
  - $f \lor g$
  - $X f$, $F f$, $G f$
  - $f U g$, $f R g$
Path formulas

- If $f$ is a state formula, $f$ is also a path formula.
- If $f$ and $g$ are path formulas, then the following are path formulas:
  - $\neg f$
  - $f \land g$
  - $f \lor g$
  - $\mathbf{X} f, \mathbf{F} f, \mathbf{G} f$
  - $f \mathbf{U} g, f \mathbf{R} g$

- CTL* is the set of state formulas generated by the above rules.
Semantics of CTL*

- Defined with respect to some Kripke structure $M = \langle S, R, L \rangle$
- New notation:
  - $\pi_i$ denotes the suffix of path $\pi$ starting at state $s_i$
  - $M, s \models f$ means that $f$ holds at state $s$ in the Kripke structure $M$
  - $M, \pi \models f$ means that $f$ holds along path $\pi$ in the Kripke structure $M$
- In the following metavariable $f$ ranges over state formulas, $g$ over path formulas, $p$ over atomic propositions
Semantics of CTL*

1. \( M, s \models p \iff p \in L(s). \)
2. \( M, s \models \neg f \iff M, s \not\models f. \)
3. \( M, s \models f_1 \lor f_2 \iff M, s \models f_1 \) or \( M, s \models f_2. \)
4. \( M, s \models f_1 \land f_2 \iff M, s \models f_1 \) and \( M, s \models f_2. \)
5. \( M, s \models E g \iff \) there is a path \( \pi \) from \( s \) such that \( M, \pi \models g. \)
6. \( M, s \models A g \iff \) for every path \( \pi \) starting from \( s \) \( M, \pi \models g. \)
7. \( M, \pi \models f \iff s \) is the first state of \( \pi \) and \( M, s \models f. \)
8. \( M, \pi \models \neg g \iff M, \pi \not\models g. \)
9. \( M, \pi \models g_1 \lor g_2 \iff M, \pi \models g_1 \) or \( M, \pi \models g_2. \)
10. \( M, \pi \models g_1 \land g_2 \iff M, \pi \models g_1 \) and \( M, \pi \models g_2. \)
11. \( M, \pi \models X g \iff M, \pi^1 \models g. \)
12. \( M, \pi \models F g \iff \) there exists a \( k \geq 0 \) such that \( M, \pi^k \models g. \)
13. \( M, \pi \models G g \iff \) for all \( i \geq 0, M, \pi^i \models g. \)
14. \( M, \pi \models g_1 U g_2 \iff \) there exists a \( k \geq 0 \) such that \( M, \pi^k \models g_2 \) and for all \( 0 \geq j < k, M, \pi^j \models g_1. \)
15. \( M, \pi \models g_1 R g_2 \iff \) for all \( j \geq 0, \) if for every \( i < j \) \( M, \pi^i \not\models g_1 \) then \( M, \pi^j \models g_2. \)
In fact, only $\lor$, $\neg$, $X$, $U$, $E$ strictly necessary

- $f \land g \equiv \neg(\neg f \lor \neg g)$
- $f \mathbin{R} g \equiv \neg(\neg f \mathbin{U} \neg g)$
- $F f \equiv True \mathbin{U} f$
- $G f \equiv \neg F \neg f$
- $A f \equiv \neg E \neg f$
Two restrictions on CTL*

Computation Tree Logic (CTL)
- If \( f \) and \( g \) are state formulas, then \( X\ f \), \( F\ f \), \( G\ f \), \( f \cup g \), and \( f \mathbin{R} g \) are path formulas.

Linear Temporal Logic (LTL)
- If \( p \in AP \), then \( p \) is a path formula.
- If \( f \) and \( g \) are path formulas, then the following are path formulas:
  - \( \neg f \)
  - \( f \land g \)
  - \( f \lor g \)
  - \( X\ f \), \( F\ f \), \( G\ f \)
  - \( f \cup g \), \( f \mathbin{R} g \)
CTL and LTL

- Two restrictions on CTL*
- Computation Tree Logic (CTL)
  - if $f$ and $g$ are state formulas, then $X f$, $F f$, $G f$, $f U g$, and $f R g$ are path formulas
- Linear Temporal Logic (LTL)
  - if $p \in AP$, then $p$ is a path formula
  - if $f$ and $g$ are path formulas, then the following are path formulas:
    - $\neg f$
    - $f \land g$
    - $f \lor g$
    - $X f$, $F f$, $G f$
    - $f U g$, $f R g$

- All these three logics have different expressive powers
- Different kinds of safety properties/errors expressible
LTL Formula examples

- $G (Req \implies F \text{ Ack})$
  - whenever $Request$ occurs, it will be eventually $Acknowledged$

- $G (DeviceEnabled)$
  - $DeviceEnabled$ always holds on every computation path

- $G (F \text{ Restart})$
  - Fairness: from any state one will eventually get to a $Restart$ state. That is, $Restart$ occurs infinitely often.

- $G (Reset \implies F \text{ Restart})$
  - Whenever the reset button is pressed on will eventually get to the $Restart$ state
• SPIN (= Simple Promela Interpreter)
• tool for analysing the logical consistency of concurrent systems, specifically of data communication protocols.
• state-of-the-art model checker, > 2000 users
• systems described in the modelling language called Promela
Promela

- Promela (= Protocol/Process Meta Language)
  - specification language to model finite-state systems
  - loosely based on CSP (*Communicating Sequential Processes*)
  - dynamic creation of concurrent processes
  - communication via message channels can be
    - synchronous (i.e. rendezvous), or
    - asynchronous (i.e. buffered)
- features from Dijkstra’s guarded command language
- features from the programming language C