ESDS final project:
Model-checking embedded systems with probabilistic nondeterminism

Stephen Tarzia

March 17, 2007

Abstract

The broad goal of this project was to start exploring the practical limitations of model checking in the context of embedded systems. Specifically, random behavior in a hard-disk power-management policy and its environment was modeled in TLA+, a language with no explicit support for probabilistic nondeterminism. Variables were added to the system state to represent the probability of reaching the current state. States violating the safety condition contribute their probability to the aggregate failure probability. This method was found to be useful for moderately-sized systems; in larger systems, aggravated state space explosion and limited probability precision are obstacles.

Contents

1 Introduction

Embedded systems are good candidates for model-checking for several reasons. First of all, a correctness proof is often required. In some devices, failure can have severe safety or security consequences. Also, hardware and firmware designs are impossible to "patch" after production, so there is an emphasis on getting them right the first time. Manual correctness proof can require a high level of mathematical sophistication and an understanding of every design decision. Thus, an automated verification technique accessible to practicing engineers is desired.

In cases where strict correctness is impossible or prohibitively expensive, a design with a small failure probability may be desired. Therefore, a system verification methodology should allow for probabilistic actions and properties.

1.1 System modelling with TLA+

TLA+ is a system specification language [2]. Formally, a system is a set of behaviors, a behavior is a sequence of states, and a state is an assignment of values to a set of variables. The primary advantage of specification is that it allows one to use a model-checker to simulate all possible behaviors of the system and thus identify any behaviors that reach unexpected states. TLC is the TLA+ model-checker.

A TLA+ code defines a set of variables, a set of actions that modify the variables, and a set of predicates on the variables. A TLC configuration file completes a system specification by indicating the semantics of the TLA+ code. The configuration file identifies the initial state predicate, the next state action, and the temporal predicate. It also has model-checker directives that specify specific values to be assigned to constants and predicates to be tested by the model-checker.

1.2 When to model-check

Model checking is attractive for systems whose complexity lies in *scheduling*. A scheduler, sometimes called an *adversary*, is a nondeterministic process that decides the order of actions in a system. For complex systems, it is difficult to generalize for every possible schedule in a manual proof; a model-checker can approach this with an exhaustive search. However, nondeterministic scheduling leads to an exponential number of behaviors, thus restricting the feasibility of *any* exhaustive approach. There is a "sweet spot" in problem complexity for which model-checking is very useful; these are systems that are too complex for manual proof but small enough to allow exhaustive behavior space exploration by a computer (thousands to millions of behaviors).

1.3 Classic challenges

Beyond the language barrier, several challenges exist in model-checking embedded systems. The *state space explosion* problem is, perhaps, the most fundamental. Most real systems exhibit wide data-value ranges and nondeterministic event ordering; both of these factors contribute to state space explosion.

2 Related work and background

Segala and Lynch [3] introduced probabilistic I/O automata (PIOA) as a model for randomized systems. They distinguish between *true* and *probabilistic* nondeterminism. True nondeterminism is decided by an unpredictable *schedule* or *adversary*. Probabilistic nondeterminism is decided by a simple Bernoulli scheme. They introduce *probabilistic executions*. These are the probabilistic analogue of a behavior; it is a tree of states where branching represents probabilistic choice. A given schedule produces a single probabilistic execution. They describe several methods of manual PIOA analysis.

PRISM [5, 4, 6] is a symbolic model-checker with explicit support for probabilistic nondeterminism and continuous time. PRISM supports three probabilistic models as shown in Table 1. The fourth model, PTA, is only supported by an expensive reduction to discrete time. It should be noted that the continuous time model used is somewhat weak. Each transition has a random delay; however, the distribution is restricted to exponential with a custom *rate*. This is analogous to the Bernoulli restriction for probabilistic nondeterminism; it is unclear to me whether this is suitably general.

Table 1: Characteristics of PRISM's supported probabilistic models

	continuous time	pure nondeterminism
discrete time Markov chains	NO	NO
Markov decision processes	NO	YES
continuous time Markov chains	YES	NO
probabilistic timed automata	YES	YES

Benini *et al.* [7] modeled the dynamic power management problem as a Markov decision process and solved it optimally in polynomial time as a *linear programming* optimization problem. This project borrows their problem formulation and will use their results as a benchmark. Since we cannot design a better policy, a model-checking solution is sought simply to develop a general-purpose probabilistic model-checking methodology.

3 TLA+ model for dynamic power management

TLA+ and TLC were chosen primarily because they were familiar. The primary contribution of this project is the modeling of probabilistic systems in a probability-agnostic language. TLA+ is expressive enough to make this possible. It is done by adding a probability variable for each instance of probabilistic nondeterminism in the system (these are *p1* and *p2* in the disk model). These variables are initialized to 1 and multiplied by a transition's probability factor when taken. The overall probability of reaching a state is then the product of each probability variable.

3.1 `Fractions` TLA+ module

TLC does not support real numbers. It only supports natural and integer numbers using the 32-bit java `int` datatype. To represent probabilities, high precision fractional numbers are needed. I tried to overcome this obstacle by defining a fixpoint approximation in the `Fractions` TLA+ module. A fraction on the range $[0, 1]$ is approximately represented by a *pair* of integers in the range $[0, 10^9]$. The two components, *upper* and *lower*, represent the upper and lower bound values. For example, the pair of integers $(450000000, 550000000)$ represents 50% plus or minus 5% and would be printed as "45.0000000%...55.0000000%". The quirks of the implementation are hidden from the user; in TLA+ code, fractions are defined in the familiar way, `a/b`. A set of fractional arithmetic operations are defined which accurately account for error introduced by the fixpoint approximation. The bounds are strict, so our probability analysis will be strictly correct (within some given error range). However, when the behavior space is very large the probability of each behavior tends to be very small and error margin becomes significant.

See Appendix B for the module code listing.

3.2 `Probability` TLA+ module

The `Probability` module extends `Fractions` to include all of the reusable probabilistic model-checking code. Most importantly,, it defines a function, `RecordFailure`, for aggregating and reporting probability values. Here I rely on two special `TLC` module functions, `TLCSet` and `TLCGet`, which operate on an array of global variables. See Appendix A for the module code listing.

Figure 1 gives part of a TLC trace for a specification using the `Probability` module. Each line reports on a discovered failure state; the probability of that state and the current aggregate failure probability are printed as ranges. When using the `Probability` module, the failure safety condition is not specified in the TLC configuration file. This is because we want to explore all states to calculate the total failure probability. Instead, the safety condition is used in the TLA+ specification to trigger `RecordFailure` and halt that behavior; see the *Next* action in Appendix C as an example.

```
"Violation: prob = 0.0012328%...0.0012329%,  total = 23.1280246%...23.1280459%"
"Violation: prob = 0.0007495%...0.0007496%,  total = 23.1287741%...23.1287955%"
"Violation: prob = 0.0009015%...0.0009016%,  total = 23.1296756%...23.1296971%"
"Violation: prob = 0.0000030%...0.0000031%,  total = 23.1296786%...23.1297002%"
"Violation: prob = 0.0000002%...0.0000003%,  total = 23.1296788%...23.1297005%"
"Violation: prob = 0.0000017%...0.0000018%,  total = 23.1296805%...23.1297023%"
"Violation: prob = 0.0000021%...0.0000022%,  total = 23.1296826%...23.1297045%"
"Violation: prob = 0.0000002%...0.0000003%,  total = 23.1296828%...23.1297048%"
"Violation: prob = 0.0000003%...0.0000004%,  total = 23.1296831%...23.1297052%"
"Violation: prob = 0.0000000%...0.0000001%,  total = 23.1296831%...23.1297053%"
"Violation: prob = 0.0000000%...0.0000001%,  total = 23.1296831%...23.1297054%"
"Violation: prob = 0.0000139%...0.0000140%,  total = 23.1296970%...23.1297194%"
"Violation: prob = 0.0000167%...0.0000168%,  total = 23.1297137%...23.1297362%"
"Violation: prob = 0.0000030%...0.0000031%,  total = 23.1297167%...23.1297393%"
```

Figure 1: Snippet of TLC Probabilistic model-checking output

3.3   Disk DPM system

Following Benini *et al.* [7], a hard-disk dynamic power management system was
modeled. As a simplification, the disk can either be *asleep* or *spinning*; tran-
sitions between these states occur based on the DPM *policy*. A disk requester
is either *active*, constantly submitting disk requests, or *idle*; transitions be-
tween these states happen with some fixed probability, based on experimental
measurements [7]. Requests that occur while the disk is asleep increase the
length of the *request queue*. In this case, the DPM policy is hard-coded into
the model (the disk sleeps with 10% probability if the request queue is empty).
Disk power consumption and workload probability parameters are specified in
the TLC configuration file (see Appendix C.1). Time is discretized; each step
represents 1 ms. Each request can be handled by the spinning disk in one time
step. The disk is initially asleep, and spinup requires one time step. The failure
condition was was average power consumption over a given value; execution
duration is a configurable parameter.

3.4   TLC model-checking results

3.5   Environment

The test machine had an Intel Core 2 6300 CPU @ 1.86 GHz with 1 GB RAM
running a Linux 2.6.18 kernel. The Sun Java 2 VM version 1.5.0_09 was used.
No significant memory swapping occurred in the model-checking. TLC was
invoked with the following syntax:
```
java -Xmx3500m tlc.TLC -config [config_file] [tla_file]
```

Table 2: DPM model-checking results

duration (ms)	checking runtime	failure lower bound	failure upper bound	total states	unique states	unique states w/out prob
5	0m1.183s	99.9954576%	99.9954613%	83	49	43
10	0m1.213s	99.9954583%	99.9954659%	182	108	93
15	0m25.511s	39.1746218%	39.1809292%	296844	145455	5397
20	1m21.359s	39.3785315%	39.3996157%	979865	459474	13407
25	10m41.591s	4.6369086%	4.7174982%	10197855	3700198	44969
30	26m48.737s	4.6018760%	4.7212036%	17201304	5996770	82755

3.6 Analysis

Table 2 gives the performance and calculated probability ranges for six different system execution durations. Duration corresponds to the number of time steps considered; this is the depth of the state tree.

The failure probability results are as expected. In very short executions the first spinup will surpass our energy budget; the 0.005% success rate represents the probability of no disk requests being made. The strong correlations in probability between the successive pairs of experiments is a bit suspicious; I expected a smoother function, so this should be investigated.

The largest probability error encountered is 0.12%, which shows that the fraction approximation used is sufficient. Generally, systems with more states will introduce greater error but we can seen that checking such systems would likely be intractable.

3.7 Cost of modelling probability

Adding probability variables to the system state has the potential to greatly increase the number of distinct states. A *system* state reachable through many different action sequences will be represented by many different *model* states, each with a distinct values for probability variables. Initial results indicated only a small increase in state space when adding probability variable. This was caused by the tendency of the probability values to uniformly underflow to zero with the limited precision initially available. Actually, the effect is significant. After adding probabilities, checking time for the 30 ms case was increased from 7 seconds to almost 27 minutes. The rightmost column in Table 2 shows the number of unique states in the original, probability-free, system.

4 Conclusions

The results show that probabilistic model-checking with TLC is feasible for moderately sized systems. However, the specification constructs needed are

somewhat complex. Some progress was made by encapsulating much of the common probability code in the `Probability` module. I believe that a single, well commented, example such as Appendix C may be sufficient to teach the use of my method.

It is always difficult to check the semantics of a specification. A positive result from the model-checker is often due to behavior-limiting specification errors. It is difficult to detect such errors. Probabilistic systems have additional parameters that much be manually checked for sanity; for example, transition probabilities must add to one.

5   Future Work

In addition to randomized workloads and algorithms, probabilistic models can be used to check embedded system subject to some types of attacks [8, 11]. Other approaches to this problem leave some room for improvement [10]. The effectiveness of the PRISM model-checker could also be evaluated.

Low-probability branches could be pruned to increase running time. This must be done carefully since small probabilities can add up arbitrarily large.

References

[1] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models, Validation, and Synthesis. *Proc. IEEE*, Mar. 1997.

[2] L. Lamport. Specifying Systems. Boston: Addison-Wesley Longman Publishing Co., 2002.

[3] R. Segala. Modeling and Verification of Randomized Distributed Real-Time Systems. Doctoral Thesis, Massachusetts Institute of Technology, 1995.

[4] M. Kwiatkowska. Model Checking for Probability and Time: From Theory to Practice. In *Proc. 18th IEEE Symposium on Logic in Computer Science* (LICS'03), pages 351-360, June 2003.

[5] PRISM web page (`http://www.cs.bham.ac.ek/d̃xp/prism`)

[6] G. Norman, D. Parker, M. Kwiatkowska, S. Shukla, and R. Gupta. Using probabilistic model checking for dynamic power management. *Formal Aspects of Computing* 17, 2, pages 160–176, Aug 2005.

[7] L. Benini, A. Bogliolo, G. Paleologo, and G. De Micheli. Policy optimization for dynamic power management. In *Design Automation Conference*, pages 182–187, 1998.

[8] D. D. Hwang, P. Schaumont, K. Tiri, and I. Verbauwhede. Securing Embedded Systems. *IEEE Security and Privacy* 4, 2 (Mar. 2006), 40-49.

[9] Koopman, P., Embedded system security, *IEEE Computer*, vol.37, no.7pp. 95- 97, July 2004

[10] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated Generation and Analysis of Attack Graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy* (May 12 - 15, 2002). 273.

[11] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Ravis, Security as a New Dimension in Embedded System Design, *Proc. Design Automation Conf.*, Jun. 2004.

A Probability module code

$$\text{------ MODULE } Probability \text{ ------}$$

EXTENDS *Fractions*
LOCAL $T \triangleq$ INSTANCE *TLC*
LOCAL $S \triangleq$ INSTANCE *Sequences*
LOCAL $N \triangleq$ INSTANCE *Naturals*

Applies factor x to probability measure p
this state transition
$p \dot{=} x \triangleq$
$\qquad p' = p **x$

Pray that the user doesnt call T!TLCSet(41) or T!TLCSet(42)
$InitSum \triangleq T!TLCSet(41, 0) \wedge T!TLCSet(42, 0)$

$GetSum \triangleq T!TLCGet(41)\dots T!TLCGet(42)$

$AddToSum(a) \triangleq$
$\wedge T!TLCSet(41, T!TLCGet(41)N! + a.lower)$
$\wedge T!TLCSet(42, T!TLCGet(42)N! + a.upper)$

$RecordFailure(a) \triangleq$
$\qquad \wedge AddToSum(a)$
$\qquad \wedge T!PrintT(\text{"Violation: prob = "}S! \circ$
$\qquad\quad FracToString(a)S! \circ \text{", total = "}$
$\qquad\quad S! \circ FracToString(GetSum))$

B Fractions module code

$$\text{------ MODULE } Fractions \text{ ------}$$

LOCAL $N \triangleq$ INSTANCE *Naturals*
LOCAL $S \triangleq$ INSTANCE *Sequences*
LOCAL $T \triangleq$ INSTANCE *TLC*

9

Fraction has two fields: lower and upper bound
$Frac \stackrel{\Delta}{=} [lower : N!Nat, \ upper : N!Nat]$

$Truth(a) \stackrel{\Delta}{=}$ IF (a) THEN 1 ELSE 0

Definition of a fraction range
$a \ldots b \stackrel{\Delta}{=} [lower \mapsto a, \ upper \mapsto b]$

Definition of fraction addition
$a ++ b \stackrel{\Delta}{=} a.upperN! + b.upper \ldots a.lowerN! + b.lower$

Definition of fraction subtraction
$a -- b \stackrel{\Delta}{=} a.upperN! - b.lower \ldots a.lowerN! - b.upper$

Lower bound for integer division
$divisionLowerBound(a, \ b) \stackrel{\Delta}{=} aN! * (1000000000N! \div b)$

Construct a fraction from the division of two Naturals
$a/b \stackrel{\Delta}{=}$
  $divisionLowerBound(a, \ b) \ldots$
  $divisionLowerBound(a, \ b)N! + Truth(aN!\%bN! > 0)$

Define multiplication
31623 = Sqrt(1000000000)
$multiplication\_lower\_bound(a, \ b) \stackrel{\Delta}{=}$
        $(aN! \div 31623)N! * (bN! \div 31623)$
        $N! + (((aN!\%31623)N! * (bN! \div 31623))N! \div 31623)$
        $N! + (((bN!\%31623)N! * (aN! \div 31623))N! \div 31623)$

$a ** b \stackrel{\Delta}{=}$
$[lower \mapsto multiplication\_lower\_bound(a.lower, \ b.lower),$
 $upper \mapsto multiplication\_lower\_bound(a.upper, \ b.upper)N! + 1]$

Formats a single fraction
$ProtoFracToString(a) \stackrel{\Delta}{=}$
        $T!ToString(aN! \div 10000000)S! \circ \text{“.”} S! \circ$
        $T!ToString((aN! \div 1000000)N!\%10)S! \circ$
        $T!ToString((aN! \div 100000)N!\%10)S! \circ$
        $T!ToString((aN! \div 10000)N!\%10)S! \circ$
        $T!ToString((aN! \div 1000)N!\%10)S! \circ$
        $T!ToString((aN! \div 100)N!\%10)S! \circ$
        $T!ToString((aN! \div 10)N!\%10)S! \circ$
        $T!ToString(aN!\%10)S! \circ \text{“%”}$

Formats an upper-lower bound pair
$FracToString(a) \stackrel{\Delta}{=}$
        $ProtoFracToString(a.lower)S! \circ \text{“...”} S! \circ ProtoFracToString(a.upper)$

C DPM TLA+ code

──────── MODULE *DPM* ────────

Specification of a Dynamic Power Management system consisting of
disk and disk requester

EXTENDS *Naturals*, *Sequences*, *Probability*

Experiment parameters
CONSTANTS *Duration*, *MaxQ*, *MaxAvgPower*

CONSTANTS *Divisor* divisor for all fractions in config file

Policy parameters
It would be better to have a fixed timeout, but this would create
many more states
CONSTANTS *DiskSleepProb* probability that the disk will sleep if idle

Workload parameters
CONSTANTS *RequesterWakeupProb*, *RequesterSleepProb*

Hardware parameters
CONSTANTS *StartupEnergy*, *SpinEnergy*, *AccessEnergy*

VARIABLES *Time*, *DiskRequestQ*, *DiskState*, *RequesterState*
VARIABLES *StartupCount*, *SpinCount*, *AccessCount*

Probability is a special variable that indicates the probability of reaching
the current state.
p1 and p2 are probabilities of each of the two decisions made.
These are integers representing hundredths of a percent
VARIABLE *p1*, *p2*

Physical state
$HardwareState \triangleq \langle DiskRequestQ, DiskState, RequesterState \rangle$

Intangible state
$VirtualState \triangleq \langle Time, StartupCount, SpinCount, AccessCount, p1, p2 \rangle$

$Init \triangleq \wedge Time = 0$
$\wedge DiskRequestQ = 0$
$\wedge DiskState =$ "asleep"
$\wedge RequesterState =$ "idle"
$\wedge StartupCount = 0$
$\wedge SpinCount = 0$
$\wedge AccessCount = 0$
$\wedge p1 = 1/1$

$\land\ p2 = 1/1$
$\land\ InitSum$

$TypeOK\ \triangleq$
 $\land\ Time \in Nat$
 $\land\ DiskRequestQ \in Nat$
 $\land\ DiskState \in \{\ \text{"spinning"},\ \text{"asleep"}\ \}$
 $\land\ RequesterState \in \{\ \text{"active"},\ \text{"idle"}\ \}$
 $\land\ StartupCount \in Nat$
 $\land\ SpinCount \in Nat$
 $\land\ AccessCount \in Nat$
 $\land\ p1 \in Frac$
 $\land\ p2 \in Frac$

$TotalEnergy\ \triangleq$
  $StartupEnergy * StartupCount$
  $+ SpinEnergy * SpinCount$
  $+ AccessEnergy * AccessCount$

Total probability is the product of the probabilities of the two components
$Probability\ \triangleq\ p1 ** p2$

ideally there would be a better latency measure than Q length
$Safety\ \triangleq$
 $\land\ TotalEnergy \div Duration < MaxAvgPower$
 $\land\ DiskRequestQ < MaxQ$

ACTIONS

$ClockTick\ \triangleq$
  $\land\ Time < Duration$ Stop clock and execution after Duration ticks
  $\land\ Time' = Time + 1$

$Service\ \triangleq$
IF $(RequesterState = \text{"active"} \land DiskState = \text{"asleep"})$ THEN
  $DiskRequestQ' = DiskRequestQ + 1$
 ELSE IF $(\land\ RequesterState = \text{"idle"}$
$\land\ DiskState = \text{"spinning"}$
$\land\ DiskRequestQ > 0$ $)$ THEN
$DiskRequestQ' = DiskRequestQ - 1$
 ELSE
  UNCHANGED $DiskRequestQ$

$ConsumeEnergy\ \triangleq$
  $\land$ IF $(DiskState = \text{"spinning"})$ THEN
  $\land\ SpinCount' = SpinCount + 1$
  $\land$ IF $(DiskRequestQ > 0)$ THEN

$AccessCount' = AccessCount + 1$
ELSE UNCHANGED $AccessCount$
ELSE
 UNCHANGED $\langle SpinCount,\ AccessCount \rangle$
$\wedge$ IF $(DiskState = \text{"asleep"} \wedge DiskState' = \text{"spinning"})$ THEN
$StartupCount' = StartupCount + 1$
ELSE
 UNCHANGED $StartupCount$

$UpdateRequesterState \triangleq$
 IF $(RequesterState = \text{"active"})$ THEN
  PROBABILISTIC NONDETERMINISM
  Active requester will become idle with some fixed probability
  $\vee$ $\wedge p1 \doteq RequesterSleepProb/Divisor$
   $\wedge RequesterState' = \text{"idle"}$
  $\vee$ $\wedge p1 \doteq 1/1 -- RequesterSleepProb/Divisor$
   $\wedge$ UNCHANGED $RequesterState$
 ELSE

   PROBABILISTIC NONDETERMINISM
   Idle requester will become active with some fixed probability
   $\vee$ $\wedge p1 \doteq RequesterWakeupProb/Divisor$
    $\wedge RequesterState' = \text{"active"}$
   $\vee$ $\wedge p1 \doteq 1/1 -- RequesterWakeupProb/Divisor$
    $\wedge$ UNCHANGED $RequesterState$

$UpdateDiskState \triangleq$
 IF $(DiskRequestQ > 0)$ THEN
  $\wedge DiskState' = \text{"spinning"}$
  $\wedge$ UNCHANGED $p2$
 ELSE

   PROBABILISTIC NONDETERMINISM
   Idle disk will sleep with some fixed probability.
   This is in liu of a timeout, to simplify the model.
   $\vee$ $\wedge p2 \doteq DiskSleepProb/Divisor$
    $\wedge DiskState' = \text{"asleep"}$
   $\vee$ $\wedge p2 \doteq 1/1 -- DiskSleepProb/Divisor$
    $\wedge DiskState' = \text{"spinning"}$

$Next \triangleq$
 Encounted a trace that violates safety
 IF $(\neg Safety)$ THEN
  add probability of this behavior to the sum
  $\wedge RecordFailure(Probability)$
  halt execution
  $\wedge$ UNCHANGED $\langle HardwareState,\ VirtualState \rangle$

 Halt execution if finished. This is normal termination.

ELSE IF $(Time = Duration)$ THEN
UNCHANGED $\langle HardwareState,\ VirtualState\rangle$

the system actions
ELSE
  $\wedge\ ClockTick$
  $\wedge\ Service$
  $\wedge\ UpdateDiskState$
  $\wedge\ UpdateRequesterState$
  $\wedge\ ConsumeEnergy$

$Fairness\ \triangleq$
  $\mathrm{WF}_{\langle HardwareState,\ VirtualState\rangle}(Next)$

$Spec\ \triangleq\ Init \wedge \square[Next]_{\langle HardwareState,\ VirtualState\rangle} \wedge Fairness$

C.1   DPM config file

```
SPECIFICATION Spec
INVARIANT TypeOK


\* Experiment parameters
\**********************
CONSTANT Duration = 20   \* Simulation duration
CONSTANT MaxQ = 5 \* Max allowed request queue size
CONSTANT MaxAvgPower = 50 \* Consuming more power than this consitutes failure

CONSTANT Divisor = 1000 \* Divisor for the fractions below


\* Policy parameters
CONSTANT DiskSleepProb = 100 \* 10% chance of idle disk falling asleep


\* Disk workload parameters, from BBPM99
CONSTANT RequesterWakeupProb = 898 \* idle requester starts up w/ 89.8% probability
CONSTANT RequesterSleepProb = 454 \* active requester stops w/ 45.4% probability


\* Hardware parameters
CONSTANT StartupEnergy = 500 \* energy consumed by asleep->spinning transition
CONSTANT SpinEnergy = 10 \* energy consumed in idle spinning state
CONSTANT AccessEnergy = 5 \* SpinEnergy + AccessEnergy = energy consumed when active
```