1. Intellectual Property Notes  
   1. Everything on this topic (the design, verification, and implementation of systems with a large discrete state space) dates back to 2003 and doesn’t represent IP that can be assigned to any employer.
2. Goals  
   * 1. To present ways of thinking about systems with a large discrete state space.
     2. To suggest:  
        1. A defined way to [manually] design / model systems with a large discrete state space.
        2. A defined way to [manually] implement systems with a large discrete space from a design / model.
        3. Automated implementation (a.k.a. automatic code generation) from a design / model.
        4. Model-checking: verifying important properties of the system automatically.
3. What is State?  
   1. Combinational mapping—outputs are a function only of the inputs, and no state is required.
   2. Sequential mapping:  
      1. Outputs are a function of inputs and current state.
      2. Next state is a function of inputs and current state.
   3. State is what the system “remembers” about the past (before the current inputs).
4. What does it mean to “hold too much state”?  
   1. Case I (less common): The system is designed so that it holds more state than necessary. The excess state is part of the design.  
      1. The prototypical example is a purely combinational mapping designed as a sequential mapping (adding state where none is required).
      2. It is also possible to add unnecessary state to a sequential mapping (add too much state where some is required).  
         1. I’m sure that there is a mathematical test for this, but it doesn’t come to mind (see [RQ1]).
   2. Case II (more common): The system is not designed prior to implementation. The excess state comes about through:  
      1. Lack of a design. Variables are added without a design, in order to get it to work.
      2. Lack of implementation rules.  
         1. RAM is typically used as both connectors (not state) and state variables / timers / counters (state). Without a clear separation between RAM used as connectors and RAM used as state, connectors can become accidental state variables.
   3. Excess state of the second type (Case II, no design activities) is (anecdotally) tied to the probability that a system might contain unknown / unacceptable behavior.  
      1. Theoretically  
         1. If the state space is partially unknown, it isn’t possible to devise all meaningful test cases or to explore the state space fully in testing.
         2. Without a design, humans have no ability to reason about the possible behaviors of the system (the system isn’t documented, let alone understood).
         3. A system that is not understood can possibly be shaken into state “corners” using stimuli that are counterintuitive or would not normally occur in testing:  
            1. Without a design or review of a design by humans, the existence of the corners might be wholly unknown.
            2. In a very complex system, even design documentation and review by humans might not be enough, as human beings are not cognitively equipped to reason about these systems. (For this reason and because of the ever-present possibility of human error, model-checking is attractive.)
      2. Practically  
         1. At a previous employer, there was a practice they called “crazy testing”.   
            1. All of the inputs were flailed wildly in hopes of revealing undesirable behavior from the product.
            2. The technique was more successful than might be imagined.
            3. The successfulness of the technique shows that state space problems are common.
         2. I had exposure to a windshield wiper problem.  
            1. The wipers worked fine, but …
            2. If the stalk switch was held down while the ignition was turned ON, the wipers would enter a mode of improper functioning from which they would not recover until another ignition cycle.
            3. This is ultimately a state space issue: with certain inputs the system entered an undesirable state.
5. State vs. Connectors  
   1. As systems are typically built, RAM is also used for connectors between state machines. By *connector* I mean shuttled data that can be calculated from input and state. It fills the role of a PCB trace, IC trace, or output of combinational logic.
   2. Connectors inherently only require a lifetime less than permanent. They are intermediate calculations. Only state needs to persist.
   3. There are three approaches to connectors I’ve seen:  
      1. Classic stack-based approach.  
         1. Characteristics  
            1. Implementations are written with no permanent-lifetime variables, except possibly state.
            2. All inputs and outputs occur through function parameter lists:

Inputs occur as input parameters.

State is pass-by-reference.

Outputs are output parameters or pass-by-reference.

* + - * 1. State (of the actual program, not design state space) is shed as the program returns up the calling tree. Stack frames disappear automatically. In this sort of program, it is hard to hold a lot of excess state.
      1. Advantages  
         1. It is almost impossible to hold excess unintended state in the actual program. The ability of the program to hold state disappears as functions return.
         2. No RAM waste.
      2. Disadvantages  
         1. It decreases performance and increases FLASH consumption because there is overhead in parameter passing and indirection (although certainly modern/bigger processors are better at it than older/smaller processors).
         2. Because the connectors disappear, the program is harder to debug (values not lying around to look at).
    1. Unions  
       1. Characteristics  
          1. Connectors are stored in a union (an overlaid structure).
          2. As a complex stateful system is implemented (i.e. as automatons are evaluated left-to-right on the DFD), first one union variant is used, then another, then another.
          3. The approach gives some of the advantages of a stack-based approach (temporary storage, RAM savings) without the overhead (parameter passing, stack-based instructions, indirection).
          4. Have only seen this used in small microcontroller software, although it would work as a technique on any platform.
       2. Advantages  
          1. Increases performance and decreases FLASH consumption.
          2. Less RAM waste than permanent lifetime approach.
       3. Disadvantages  
          1. Takes effort to balance unions manually.
          2. Small changes in design may result in large changes in union layout.
          3. Code is not the most readable.
          4. Mistakes in one union variant overwriting data of another are possible.
          5. Because the connectors disappear, the program is harder to debug (not many values lying around to look at).
    2. Permanent Lifetime  
       1. Characteristics  
          1. State and connectors are all permanent lifetime.
          2. Bitfields tend to be used because they give the most compact representation for the permanent lifetime data.
       2. Advantages  
          1. Increases performance and decreases FLASH consumption.
          2. Easiest to debug, because connector values persist.
       3. Disadvantages  
          1. Highest RAM consumption.
          2. Because RAM holds state, it is very easy to get confused about the intended state space of the system and introduce logical errors.

Connectors can accidentally become state variables.

Designs and design rules are necessary to prevent that.

* 1. They key logical issue is that RAM used as a connector can accidentally become a state variable. It requires implementation rules to keep this from occurring. [This is unlike hardware digital logic, where the connectors (traces on a silicon die or a PCB) don’t have the capability to hold state.]

1. Timed Automata  
   1. Restricted modeling framework (Alur & Dill, UPPAAL) involving a collection of timed automatons (a.k.a. state machines).  
      1. Restrictions  
         1. The restrictions are to some degree arbitrary and I don’t believe they are unique. They serve only:  
            1. To ensure that the automatons can be mathematically combined into a larger timed automaton that honors the same design rules.
            2. To allow mathematical inferences about the possible values of timers in each discrete state (polytopes, DBMs, discussed later).
            3. To allow state exploration (discussed later).
      2. Typical restrictions:  
         1. Time has to be treated in a certain way. The typical restrictions are:  
            1. Each automaton may have arbitrarily many timers, all independent.
            2. Timers always increase.

They can’t be stopped or started.

They increase in all discrete states.

* + - * 1. Timers can only be set to 0, and only in transitions.
      1. Only certain types of tests may occur in guards (conjunctions and disjunctions involving timers, counters, and the states of other automatons).
  1. System of concurrent state machines that may interact through defined mechanisms (below).  
     1. The discrete state machines may have guards that test each other’s states (hence the action of one automaton may depend on the state another automaton is in).
     2. The automatons may have synchronization events that allow or force simultaneous transitions to be taken by different automatons. (I’m not sure of the exact semantics.)
     3. A note about synchronization:  
        1. For the synchronization events, it might be obvious that when automatons are combined, different transitions guarded by synchronization events are taken at the same time—they really in some sense *are the same transition*.
        2. However, what might be less obvious is that:  
           1. If Automaton A is stuck in a state involving a guard condition requiring Automaton B to be in a certain state, THEN
           2. Automaton B makes a transition so that its state is that required by the guard condition in Automaton A …
           3. The transition in Automaton B isn’t *followed by* the transition in Automaton A. Instead, from a modeling point of view, they happen *simultaneously*. Transitions are zero-time in that sense.
  2. Counters are also allowed, and there are restrictions on how they can be used and tested in guards.
  3. The framework allows the automatons to be combined mathematically into a larger automaton. (Of course, likely exponential growth in the number of discrete states.)
  4. What is the true state space?  
     1. Discrete state.
     2. Timers
     3. Counters

1. Model-Checking Tools  
   1. How a model-checking tool works:  
      1. Polytopes, DBMs, and reasoning about reachability.
      2. State Exploration  
         1. With a continuous system (classic control system), there are certain ways of reasoning about infinite sets (differential equations, stability criteria, etc.) that don’t require a lot of computer capability.
         2. With automata, verifying properties involves state exploration, and is exponential with respect to the number of automatons being combined. (There are of course combinations of automata and properties that will result in quick verification, but this is not the general case.)
         3. Because of the exponential growth, the composite automaton can’t be built as a data structure in memory in advance. The model-checking tools have clever ways of traversing the state space while keeping a reasonable memory footprint.
   2. Additional reading/activities:  
      1. *Model Checking* by Clarke, Grumberg, and Peled.
      2. Papers by Alur and Dill.
      3. Papers surrounding UPPAAL.
      4. It is also possible to ask for a research license for UPPAAL and download and try it out for free.
2. Possible Lessons to Learn from Timed Automata  
   1. A collection of state machines within a software component really make up one much bigger state machine.  
      1. You can actually determine what that larger state machine is (given adequate computing resources).
      2. In this framework, I have no idea how to formally think about systems that are multithreaded—I only know how to formally think about one thread at a time.
   2. Treatment of time.  
      1. Timers that can be stopped and started may be undesirable and unnecessarily complex.  
         1. In the TA framework, timers can’t be stopped (they can only be reset to 0).
         2. Time is part of the ether … you can’t stop time.
         3. Is this simpler style more compatible with psychology? Does it lead to designs that are easier to understand? I’m not sure.
      2. Time is integral.  
         1. This is very consistent with microcontroller work.
      3. Timers count upward  
         1. In the TA framework, I believe this simplifies the determination of polytopes that are paired with discrete states.
         2. In microcontroller work, timers which count down but never below zero are most common.
         3. Whether this would complicate the verification of properties through model-checking is unclear (see [RQ3]).
   3. Not all bit patterns that can be held by RAM represent valid states of the system.   
      1. Reasons:  
         1. In certain discrete states, certain timer values can’t be reached (polytopes, etc.).
         2. In certain discrete states, certain counter values are not possible.
         3. Not all combinations of discrete states are reachable.
      2. Identification of invalid states:  
         1. In the case of a single discrete state variable, it is common practice to equivalence-class the variable (*default:* *switch* case, final *else* clause).
         2. However, I’m not aware of any common implementation practice that covers:  
            1. Interdependencies between automatons so that some combinations of discrete variables are not reachable.
            2. Certain timer values not being possible in some discrete states.
            3. Certain counter values not being possible in some discrete states.
      3. Conclusion:  
         1. In general, in most implementations, a state upset may take the controller outside the state space it might reach in proper operation, and this upset would not be detected.
         2. Even if a state upset were detected and corrected, for that to be meaningful, the system would need to have RXX robustness (mentioned later), and I’m not sure how many do.
         3. Most software systems therefore won’t tolerate an arbitrary state upset while still behaving acceptably.
   4. Composition and decomposition of a sequential system is not unique. More than one set of automatons combined may lead to the same behavior.
   5. What a counter really is.
   6. What a bitfield really is.
3. Notions of robustness.  
   1. A system consists of a plant and a controller. The plant may be a mechanical system, a piece of digital logic, another software system, whatever. Both may be upsettable.
   2. Control systems are typically characterized by performance criteria (stability, step response, overshoot, etc.). But I have no idea how to think about “acceptable behavior” when a controller or plant are upset. (A few notions come to mind.) I believe the notion of “acceptable behavior in the case of state upset” is a different (but possibly in some ways related) notion than control system performance criteria. (See [RQ4].)
   3. R00 robustness: the system behaves acceptably when both the controller and plant start from a unique initial state. This might also be called correctness of the controller.
   4. R0X robustness: the system behaves acceptably if the controller starts in an initial state but the plant starts in an arbitrary state. (Include the transfer case story. I believe this level of robustness is nearly standard design practice for the design of controllers to control mechanical systems, communication peripherals, other smart subsystems that might encounter a state upset, etc.)
   5. RX0 robustness: the system behaves acceptably if the physical plant starts in an initial state but the controller might be upset at any time. I believe it doesn’t make sense to consider this possibility, as it essentially reverses the role of the controller and the plant.
   6. RXX robustness: the system behaves acceptably if the physical plant and controller may begin operation in any pair of arbitrary states.
4. Software engineering perspective  
   1. Global variables, the coupling spectrum.
   2. Complexity, abstraction, understanding complex systems.
   3. Intangibles.
5. Implementation Styles  
   1. Discrete time ticks.  
      1. One tick of time is one invocation of the code. A certain segment of code executes (usually a function call).
   2. Boundary synchronization  
      1. Left boundary: collecting all asynchronous data.
   3. Block-zeroing.
   4. Time  
      1. Arrays of timers.
      2. 1-2-5, 2N versus .
      3. Value-dependent counting.
6. Recommended implementation style  
   1. Observations about practical systems.  
      1. What would be modeled as synchronization events are fairly rare. Guard conditions involving the states of other automatons are far more common.
      2. Cyclical data flow that occurs in a software component is fairly rare. It is usually possible to model a system as a set of automatons with data flow from left to right. (Cyclical data flow does occur, but it usually involves the plant external to the controller.)
   2. The most dominant design material is a DFD (with the other design rules understood, but not part of the abstraction).
   3. Time.
   4. Interfaces between sub-components:  
      1. W1RN variables.
      2. TCSMs.  
         1. Semaphores and fanout.
         2. The general case.
   5. Sub-component design.
7. Recommendations:  
   1. Conceptual.  
      1. [ISG1] Use a consistent standard in the software and documentation for the meaning of event versus the meaning of signal.
   2. Customer Specs.  
      1. [ISG2] Use a consistent standard for the meaning of event versus the meaning of signal.
      2. [ISG3] Eliminate complementary conditions, i.e. <<Rear Wheel Fell Off>> and <<Rear Wheel Still Attached>>. (It should always be <<Rear Wheel Attached>> and !<<Rear Wheel Attached>>, when possible.)
      3. [ISG4] Use a non-sequential way of tagging requirements in the spec. to trace to design materials and code.
   3. Correspondence between spec. and code.  
      1. [ISG5] Harmonize the code and spec. (The specs. needs to closely reflect the way the software is likely to be constructed.)
   4. Design / Design Documentation / Code Documentation  
      1. [ISG] Adopt a standard design, design documentation, and implementation approach for stateful systems.
      2. [ISG] Code should meet documentation minimums.
   5. General structure  
      1. [ISG] Flat calling tree.
      2. [ISG] Adopt the RCS (RAM-connector state-minimal) method of design/implementation described in this presentation.
   6. State machine implementation  
      1. [ISG] Guard conditions:  
         1. Do not use the following practices involving guard conditions:  
            1. Evaluating guard conditions in called functions.
            2. State changes in called functions.
            3. Multiple tests of guard conditions that need to be kept consistent.
         2. Instead, use this recommended style.  
            1. Need to discuss the alternate way of handling complex guards.
      2. [ISG11] Eliminate transitions involving the same discrete state variable thousands of lines apart.
   7. Time as maintained in the code.  
      1. [ISG] Eliminate timers that count down in some states but not in others, as much as possible. (Time should be part of the ether.)
      2. [ISG] Consider eliminating timers that expire.
      3. [ISG] Enhance the definition of timer to include expired and inactive without adding additional variables.
   8. Modularization of code  
      1. [ISG15] Eliminate separate modules (as known to the compiler/linker) for code in the same thread.
      2. [ISG] Create thread-safe versions of block memory operations, and place them in a common location (where they are available to all threads).
      3. [ISG] Create thread-safe versions of timer operations, and place them in a common location (where they are available to all threads).
   9. Variables  
      1. [ISG18] Eliminate special purpose structures and unions. (The schematic “connector” arugment.)
   10. Function naming conventions  
       1. [ISG19] Use function naming conventions based on which threads functions are intended to be called from.
   11. Preprocessor directives  
       1. [ISG20] Do not use a “DISABLE”/”ENABLE” convention.
       2. [ISG] Document configuration switches.
   12. Source File Revision History  
       1. [ISG] Place revision history at the *end* of the source file. (Distraction / psychology.)
8. Research Questions  
   1. [RQ1] What is the mathematical test for a sequential design that holds too much state?
   2. [RQ2] Is there anything analogous to the FTA (Fundamental Theorem of Arithmetic)? Can an automaton be decomposed (i.e. factored) in any provably unique way? (I have the feeling that this might be a linchpin of proving the equivalence of systems of automatons and of good code generation.)
   3. [RQ3] Are automatons using timers which count down but never below zero as amenable to model checking as classic TA’s? If not, why not?
   4. [RQ4] How does one think about plant and controller upset in the context of model checking?  
      1. I’ve never noticed any mention of plant or controller upset in TA literature.
      2. The notion of “no unacceptable behavior” while a system recovers from upset seems to requires that properties to be verified in the case of upset be kept separately from properties to be verified in normal operation. I’m not sure how to think about this problem.