1. Intellectual Property Notes

	1. Everything on this topic (the design, verification, and implementation of systems with a large discrete state space) is pre-existing and pre-dates my employment.
	2. The appearance of this presentation within the company should not be interpreted to mean that the ideas were developed while employed by the company or that any intellectual property is assigned to the company.
	3. The company is welcome to use, without restriction or remuneration of any type, this presentation in any form and everything within this presentation.
2. Overview

	1. Presentation is partially an analysis of an existing corporate software component and a corporate product.
	2. This presentation is inherently about a specific software component in a specific product in a specific company. I’d like to pose some questions to researchers, but I can’t refer to the actual product, actual software component, or actual company.

		1. The product name is anonymized to *Widget*.
		2. The component name is anonymized to *Component*. The three files of the component are anonymized to *x.c*, *x.h*, and *x\_comm.c*.

			1. *x.c*: the major stateful component.
			2. *x\_comm.c*: the portion of the component that deals with network communication.
			3. *x.h*: the header file for both *x.c* and *x\_comm.c*.
		3. The company name is anonymized to *Tuple* (an incredibly bad joke, I know).
	3. Goals

		1. To present ways of thinking about systems with a large discrete state space.
		2. To steer the company towards:

			1. A defined way to [manually] design systems with a large discrete state space.
			2. A defined way to [manually] implement systems with a large discrete space from a design.
			3. Automated implementation (a.k.a. automatic code generation) from a design.
			4. Model-checking: to verify 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 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.
	2. Case II (more common): The system is not designed at all. The excess state comes about through:

		1. Lack of a design. Variables are added without a design 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.
5. Timed Automata

	1. Restricted modeling framework (Alur & Dill, UPPAAL).
	2. System of concurrent state machines.
	3. Time.
	4. Counters.
	5. Interaction mechanisms:

		1. One state machine may take a transition or not take a transition based on the state of another (weak synchronization mechanism).
		2. Events (stronger synchronization mechanism).

			1. Synchronization events.
			2. Input and Output events.
	6. The framework allows the automatons to be combined mathematically into a larger automaton. (Of course, exponential growth.)
	7. What is the true state space?

		1. Discrete state.
		2. Timers
		3. Counters
6. Model-Checking Tools

	1. How a model-checking tool works:

		1. Polytopes 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.
7. 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. You can actually determine what it is (given adequate computing resources). (Note: 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. The maintenance of time.
	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. And 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 bitfield really is.
8. 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. (ORQ: How should one think about this?)
	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 systems 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.

		1. RXX robustness: the system behaves acceptably if the physical plant and controller may begin operation in any pair of arbitrary states. This level of robustness
9. 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), and when this function
	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.
10. Recommendations for the component (*x.c*, *x.h*, and *x\_comm.c*).

	1. Conceptual.

		1. Get the code clear on the meaning of event versus the meaning of signal.
		2. Get the customer clear on the meaning of event versus the meaning of signal.
	2. Correspondence between spec. and code.

		1. Bring the code as close as possible to the spec.
		2. Bring the spec. as close as possible to the code.
		3. Find a non-sequential way of tagging the spec. to trace the code to the spec.
	3. Time as maintained in the code.

		1. Eliminate timers that count down in some states but not in others. (Time should be part of the ether.)
		2. Consider eliminating timers that expire.
		3. If that won’t work, enhance the definition of timer to include expired and inactive without adding additional variables.
	4. Function naming conventions

		1. Consider naming functions based on how they are called and threading issues.
		2.
11. Recommendations for Specs

	1. 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.)