Paper 2: ESP - Writeup
Louis Rassaby, Anschel Schaffer-Cohen, and Karl Cronburg
(with lecture notes written by Mike Shah)

1. Warm-ups
   1.1. Domain: firmware
   1.2. Programmable device
       1.2.1. Definition: a device that has its own CPU and memory and can run code. This can be self-contained, or peripheral to another computer.
       1.2.2. E.g.
           1.2.2.1. x-card, for any device (network card, graphics card, etc.)
           1.2.2.2. Microwave
           1.2.2.3. Embedded computers in cars etc.
   1.3. Event-driven state machines
       1.3.1. What are they?
           1.3.1.1. Finite automata that change state when they receive signals.
       1.3.2. How do they work?
           1.3.2.1. Each process is an individual event-driven state machine, but multiple processes combined can also be thought of as a state machine
           1.3.2.2. Each state is either a send or receive
       1.3.3. Why are they used to program firmware?
           1.3.3.1. They produce low-memory, very efficient code
       1.3.4. Why are they difficult to get right in C?
           1.3.4.1. Race conditions
           1.3.4.2. Bad memory management
           1.3.4.3. Global state
               1.3.4.3.1. No guarantees about state interactions, so the total state space grows exponentially
           1.3.4.4. Functions with high overhead
           1.3.4.5. Difficult to test and debug
           1.3.4.6. Patches difficult to deploy, so it’s worth getting it right the first time
   1.4. Goals: Ease for the developer, ability to connect to test suite, low performance penalty versus C, minimize the state space
   1.5. Appendix B: An updatable key-value store.

2. Design Evaluation
   2.1. Features of ESP
       2.1.1. ESP provides:
           2.1.1.1. Verifiable code
           2.1.1.2. Simpler constructs
           2.1.1.3. Simple interface to C and SPIN
           2.1.1.4. Basic type inference
2.1.1.5. Memory management through reference counting (link and unlink operations)
2.1.1.6. Aggressive compiler optimizations
2.1.1.7. Removal of some boilerplate code from the developer's hands
2.1.2. Are they sufficient?
   2.1.2.1. "I'd use it." - Michael James
   2.1.2.2. The real question for something practical, is "can it be used to program these devices", which the paper seems to answer with a resounding yes
2.1.3. Memory management
   2.1.3.1. Reference counting
      2.1.3.1.1. Memory management being local makes this a useful feature. It also limits the state space.
      2.1.3.1.2. Type system has a hole from reference counting, but this is verified by SPIN.
   2.1.3.2. Some other means of garbage collection
      2.1.3.2.1. More memory and code must be allocated for the collector, so not economically good
      2.1.3.2.2. Stop-the-world collection wouldn't be ideal
      2.1.3.2.3. Verification becomes cheaper once you trust the garbage collector
2.2. Brevity and modularity
   2.2.1. Modifying a state is simple, unlike in C
   2.2.2. Pattern matching
   2.2.3. Clear and simple syntax
   2.2.4. Corresponds closely to the way people think about state machines
   2.2.5. Channels allow for modularity, since you can cleanly replace one process with another that has the same interface
2.3. Efficient compilation
   2.3.1. No function overhead means fast context switches
   2.3.2. Aggressive compiler optimizations based on the theory of an event-driven state machine
   2.3.3. Copy optimizations for immutable data
   2.3.4. No shared data between processes
2.4. Verification and debugging
   2.4.1. Compiles directly into SPIN tests
   2.4.2. Complexity localized to a small portion of the code
      2.4.2.1. In particular, all potential memory unsafety is process-local
   2.4.3. No shared memory between processes, so the state space grows slowly
   2.4.4. The ability to differentiate between mutable and immutable data reduces the state space so it is easier to verify a program
2.5. Implementation
2.5.1. A compiler that generates C and SPIN code as well as the runtime system surrounding the C code

2.6. Evaluating the language

2.6.1. What they did:
2.6.1.1. Case study: VMMC firmware
2.6.1.2. Introducing bugs and trying to find them

2.6.2. Improving what they did?
2.6.2.1. More variety of testing--if this is supposed to be useful for lots of programmable devices, write firmware for more than one

3. Evaluating ESP as a DSL

3.1. Advantages and disadvantages

3.1.1. Advantages
3.1.1.1. Familiarity to firmware programmers
3.1.1.2. Less of a disconnect between C code and ESP code

3.1.2. Disadvantages
3.1.2.1. There’s probably a better way to think about automata than through these machines

3.2. ESP-specific tool support
3.2.1. GUI with state diagrams for development and debugging would be very helpful for analysis of programs
3.2.2. Existing C debuggers wouldn’t know about ESP code

3.3. Type system
3.3.1. Type safe?
3.3.1.1. No. It is possible to unlink a resource and then later to try to link to the freed resource, creating an error.
3.3.2. No functions, enums, or recursive types

3.4. Yes. The runtime system is the glue that runs the C code created by compiling ESP code and creates the interfaces for a developer’s SPIN and C code to build on.

3.5. The design could be improved by
3.5.1. Support for push-based events as well as the normal pull-based events
3.5.1.1. I’m not sure what this would mean, in practice. Presumably the scheduler doesn’t try to wake a process unless it can be unblocked; so a receive() in an infinite loop is essentially a push-based event.

3.5.2. Compilation into other testing frameworks apart from SPIN
3.5.3. More customizable fast path and event prioritization support
3.5.3.1. The authors cited the simplicity of avoiding convoluted fast paths as a benefit to using ESP, but there’s definitely a use case for hand-optimized fast paths
3.5.3.2. Smart programmers might be able to specify which processes deserve more scheduling time
3.5.4. Pure functions, which can be inlined by the compiler, so as to write more modular code
   3.5.4.1. Macros might be the best of both worlds

3.6. Is ESP a DSL?

4. More detailed questions
   4.1.
   4.2. The $ has two meanings
      4.2.1. Passing variables through channels
      4.2.2. Initialization of variables
   4.3. Only immutable data travels over channels. Why?
      4.3.1. Safety for concurrency
   4.4. Why does ESP not copy objects sent as messages over channels? Not necessary with the idea of event-driven state machines
   4.5. For devices with very limited memory and CPU, having a low function overhead means that resources can be prioritized to handling requests rather than expensive context switches.
   4.6. One can unlink a resource then later link that some resource creating a type error.
   4.7. Using channels as the external interface
      4.7.1. Advantages
         4.7.1.1. Can pattern match on channels, allowing you to easily send a message on multiple channels
         4.7.1.2. Channels are static compile-time entities, drastically reducing the state-space necessary for a verifier to deal with concurrency in external code
         4.7.1.3. Seamless blocking mechanism for events
         4.7.1.4. Use of channels promote modularity
      4.7.2. Disadvantages
         4.7.2.1. Existing code needs to be refactored to use channels, namely code which currently uses shared memory
         4.7.2.2. Channels require explicit construction whereas shared memory is implicit (more explicit code to write with channels)
   4.8. SPIN is a model checker for systems made up of concurrently running state machines. ESP uses SPIN to simulate running multiple copies of the program, with SPIN making random decisions about when to execute which state machine and when to transmit which message. The randomness of SPIN tends to help it find more bugs.
   4.9. SPIN does not support pointers, making it difficult to create SPIN specifications from C code.
   4.10. The higher-level semantic information available to ESP during translation to C makes dead-code elimination easier to detect at the ESP level, thus reducing the chances of the C compiler not detecting the ability to perform these optimizations.
4.11. Implementing channels as queues on which writers wait requires $O(# \text{ patterns})$ runtime before a process can block on a channel whereas the implementation in ESP (per-process bitmask) requires $O(# \text{ processes})$ which will usually reduce to $O(1)$ since the bitmask for multiple processes can be co-located into one 4 byte region of memory.