Medical Device Automation Using Message-Passing Concurrency in Scheme

Vishesh Panchal and Bob Burger
September 22, 2013

2013 CUFP Conference

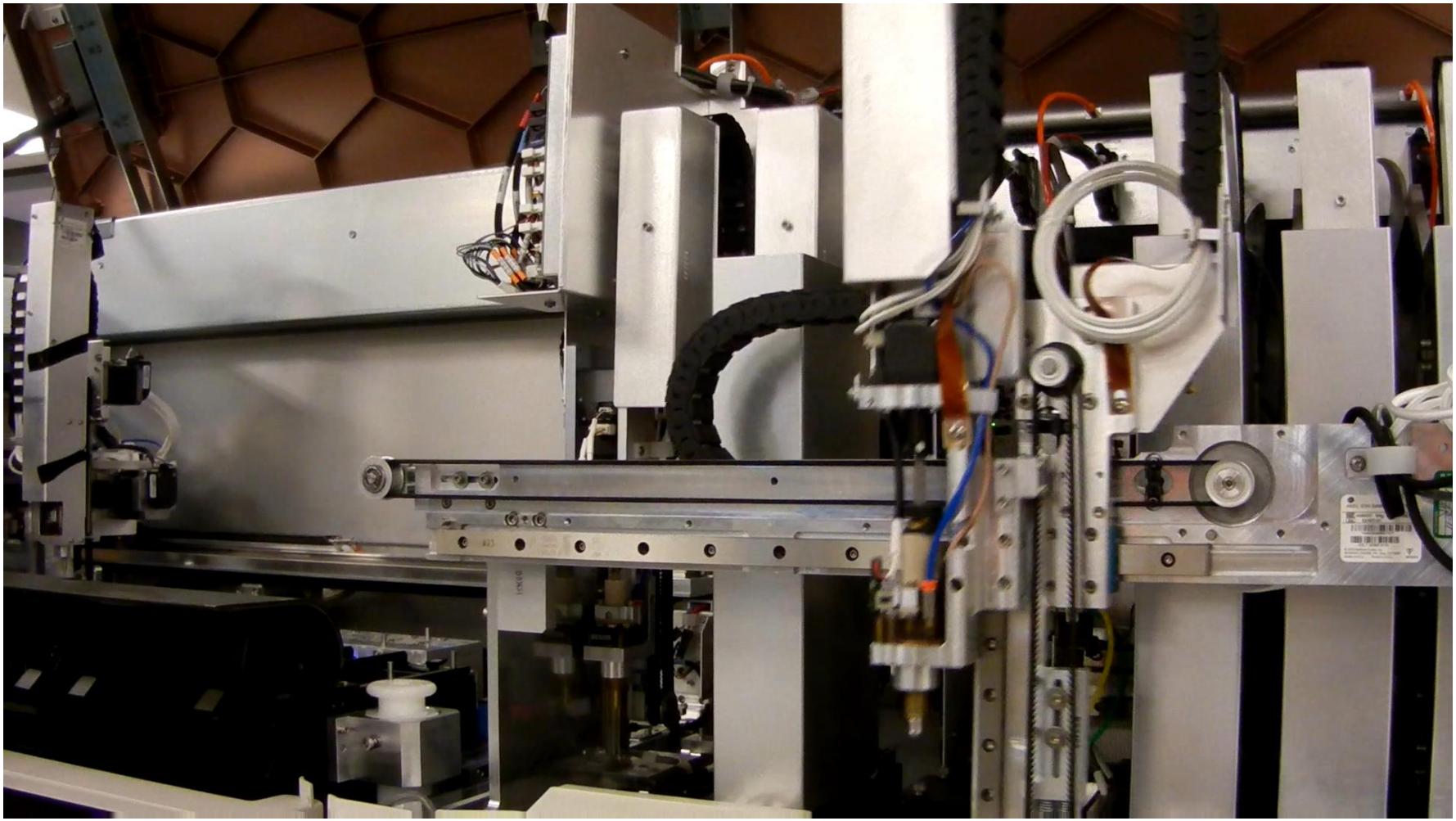
BECKMAN COULTER.

Agenda

- What does a molecular diagnostic device do?
- Overview of the software
  - Scheme
  - Erlang/OTP embedding
  - Fault tolerance and error handling
  - Domain-specific language for chemistry
  - Web server
- Lessons learned
- Take away

What does a molecular diagnostic device do?

- Detect the presence of specific strands of DNA/RNA in a sample
- Nucleic acid extraction & purification (EP) with magnetic beads, polymerase chain reaction (PCR) amplification, and fluorescent dye based quantification to detect specific viral loads
- Samples are excited using lasers as they undergo PCR which results in spectrographs
- Spectral decomposition of multiple fluorescent dye emissions across the PCR process determines the clinical result
- The device contains 19 boards with temperature, motor control, and sensors, two barcode readers, and a spectrometer
- To avoid cross-contamination it uses cartridge based chemistry and pipettors with disposable tips.



Overview of the Software

- Client/server architecture with a thin stateless client
- The client is written in C# and WPF.
- The instrument server is written in Scheme with Erlang and OTP embedding to leverage the pattern matching and message-passing concurrency of Erlang.
- The supervision structure isolates hardware failures and enables fault tolerance and recovery.
- A domain-specific language (DSL) provides a platform for scientists and other engineers to write assays to perform chemistry and other engineering experiments.
- A web server with support for HTTP, jQuery, and JSON enables debugging and visualization of logged data remotely.

Scheme Overview

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

–*Revised^6 Report on the Algorithmic Language Scheme*, 2007

Characteristics and Advantages of Scheme

- Dialect of Lisp invented by Steele & Sussman in 1975
- Exceptionally clear and simple syntax and semantics
- Few different ways to form expressions
- No restrictions on composing expressions
- First-class procedures and continuations
- Powerful, hygienic syntactic abstraction facility with special forms and pattern matching
- Dynamic, strong typing
- Automatic memory management
- Control: exceptions, threads, backtracking
- Arbitrary precision arithmetic with rational and complex numbers

Erlang Embedding in Scheme

- Records
  - Enables referencing structured data by field name instead of field index
  - Allows for copying a record and changing particular fields

  - Declaration:

```
(define-record <input-event>
  input-state rack-count spu-offload-state is-processing-input?)
```

  - Referencing:

```
(<input-event> event input-state)
```

  - Copying:

```
(<input-event> copy event [input-state 'not-running])
```

Erlang Embedding in Scheme

- Pattern Matching
  - Enables matching on numbers, symbols, strings, lists, vectors, and records succinctly
  - Allows for matching the value against a local variable and/or binding values to local variables in the pattern itself

  - Example:

```
(lambda (event)
  (let ([starting-rack-count 3])
    (match event
      [`(<input-event> [input-state not-running]
                       [rack-count ,@starting-rack-count]
                       ,is-processing-input?)
       (log is-processing-input?)
       (send 'input-manager 'start)]
      [,_ (void)])))
```

Erlang/OTP Embedding in Scheme

- Processes, monitors and links
  - One-shot continuations and software timer interrupts provide light-weight process implementation.
- Gen-Server
  - Provides a functional way of mutating state by serializing requests through a single message queue
  - Implements init, handle-call, handle-cast, handle-info, and terminate
  - Currently, code-change is not supported
- Event Manager
  - Responsible for logging all events and manages subscribers for events across the instrument server
- Supervisor
  - Monitors other processes and is responsible for restarting them or shutting down safely in case of failures

Instrument Server Concurrency Model

- System decomposed into processes.
- Processes communicate with messages, not shared memory.
- Faults are isolated to the process that causes them.
- Processes can monitor and link to other processes to detect faults.
- Supervision trees manage fault monitoring and recovery.
- Time-outs mitigate deadlock.
- Error handling slogans from Erlang:
  - Let some other process do the error recovery.
  - If you can't do what you want to do, die.
  - Let it crash.
  - Do not program defensively.
- Generic servers and event logging aid debugging.

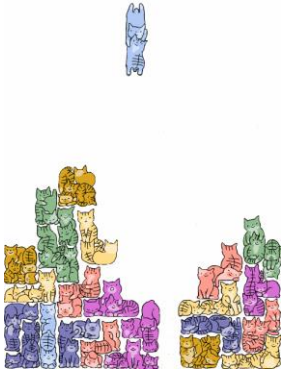
Assay Execution

- Run all the actions described in the DSL in a *confirm mode* to estimate running time, suggest missing resources, and warn about time allotment errors.
- Queue the assay while accounting for resources used by each action.
- Schedule the assay.
- Run each action in its own process for fault tolerance.
- The DSL even exposes an option to ignore action failures, which is handy during test runs and debugging.

Assay Execution

- Run all the actions described in the DSL in a *confirm mode* to estimate running time, suggest missing resources, and warn about time allotment errors.
- Queue the assay while accounting for resources used by each action.
- Schedule the assay.

  It's just like playing Tetris …
- Run each action in its own process for fault tolerance.



- The DSL even exposes an option to ignore action failures which is handy during test runs and debugging.

Device Failure Examples

Therac 25

- "A radiation therapy device that overdosed patients with radiation. The realtime operating system (RTOS) for the device did not support a message passing scheme for threads. Therefore global variables were used instead. Insufficient protection of the global variables resulted in incorrect data being used during device operation. A second issue was an overflow condition on an 8 bit integer counter. These software coding errors resulted in overdosing patients with as much as 30 times the prescribed dose."
- In the instrument server, there is no shared, mutable state, so the first issue would not occur.
- Scheme uses exact, arbitrary precision arithmetic and raises an exception if a number does not fit in a fixed-sized integer, so the second issue would not occur.

Device Failure Examples

Ariane 5

- "The very first launch of this rocket resulted in the destruction of the launcher with a loss of the payload. The cause was an overflow in a pair of redundant Inertial Reference Systems which determined the rocket's attitude and position. The overflow was caused by converting a 64 bit floating point number to a 16 bit integer. The presence of a redundant system did not help, because the backup system implemented the same behavior."
- In Scheme, the conversion of a floating-point number to an integer raises an exception if it does not fit in the target size.

Device Failure Examples

Space Shuttle Simulator

- "During an abort procedure all four main shuttle computers crashed. Examination of the code identified a problem in the fuel management software where counters were not correctly reinitialized after the first of a series of fuel dumps were initiated. The result was that the code would jump to random sections of memory causing the computers to crash."
- Scheme code cannot jump to random sections of memory.
- In the instrument server, counter management would likely be done in a gen-server, whose state-change behavior is specified functionally. If the code neglected to update the counter after the first of a series of fuel dumps, the gen-server would likely crash. The crash report would include the state and message at the time of the crash to help debug it.

Run-Time Error Causes and Their Mitigations

- Non-initialized data
  - In Scheme, variables are defined with their values. Mutations are handled with separate processes that manage state predictably (e.g., gen-servers).
- Out of bounds array access
  - In Scheme, all memory accesses are safe, and a run-time exception is raised if an array is accessed beyond its bounds.
- Null pointer deference
  - In Scheme, there are no NULL pointers, but it is possible to pass an unexpected datum to a procedure. When this occurs, a run-time exception is raised.
- Incorrect computation
  - In Scheme, numeric operations return the mathematically correct result or raise an exception

Run-Time Error Causes and Their Mitigations

- Concurrent access to shared data
  - In the instrument server, there is no shared, mutable state.
- Illegal type conversions
  - In Scheme, all type conversions preserve information or raise an exception.
- Dead code
  - Code inspections identify dead code, and profile data from testing highlights code that has never been executed.
- Non-terminating loops
  - In the instrument server, most operations involving gen-servers have a time-out. When the time-out occurs, an exception is raised, and the supervision hierarchy handles it.

Need for a DSL

- Assay developers
  - Which reagents should be added, and in what order?
  - What mixing and waste removal steps need to occur?
  - Are incubations needed? When, how long, and how hot?
- Mechanical and electrical engineers
  - Run module-level tests with custom logging
- Reliability engineers
  - Run the whole system to determine long-term failure modes
- Systems engineers
  - Develop and characterize pipetting, mixing, waste, and tip loading protocols at the motor level
  - How do the operations fit together on the system timeline?

DSL for Molecular Diagnostic Device

- Extraction/purification language
  - tip loading & unloading
  - pipetting
  - moving magnets
  - incubating
  - timing
- Protocol language
  - aspirating
  - dispensing
  - moving the tip
  - waiting
  - moving motors
  - controlling shuck valves
- PCR language
  - thermal cycling
  - reading spectrographs

DSL PCR Example

1. Bake the enzyme for 1 minute at 37°C
2. Bake for 2 minutes at 60°C for the reverse transcription (RT) process to occur.
3. Bake for 1 minute at 95°C to activate the enzymes.
4. Thermal cycle the sample 40 times as follows:
   a. Bake for 5 seconds at 95°C to denature the DNA.
   b. Bake for 20 seconds at 60°C for the DNA to anneal.
   c. Take spectrographs at the midpoint
      i. with a red laser for configured-time.
      ii. with a green laser for 60 milliseconds.

DSL PCR Example

```
(bake enzyme
  (time 1 minute)
  (temperature 37 C))
(bake RT
  (time 2 minutes)
  (temperature 60 C))
(bake enzyme-activate
  (time 1 minute)
  (temperature 95 C))
(cycle 40 times
  (bake denature
    (time 5 seconds)
    (temperature 95 C))
  (bake anneal
    (time 20 seconds)
    (temperature 60 C)
    (read-spectrum
      (wait-time 10 seconds)
      (integration-times
        (red configured-time)
        (green 60 ms)))))
```

1. Bake the enzyme for 1 minute at 37°C
2. Bake for 2 minutes at 60°C for the reverse transcription (RT) process to occur.
3. Bake for 1 minute at 95°C to activate the enzymes.
4. Thermal cycle the sample 40 times as follows:
   a. Bake for 5 seconds at 95°C to denature the DNA.
   b. Bake for 20 seconds at 60°C for the DNA to anneal.
   c. Take spectrographs at the midpoint
      i. with a red laser for its configured time
      ii. with a green laser for 60 milliseconds

Cycle Form Definition

```
(define-syntax cycle
  (syntax-rules (time times)
   [(_ n times body ...)
    (let ([count n])
      (do ([i 1 (+ i 1)]) ((> i count)) body ...))]
   [(_ n time body ...)
    (cycle n times body ...)]
   [(_ n body ...)
    (cycle n times body ...)]))
```

Example usage:

```
(cycle 40 times
  (move motor ‘xyz-x (position home))
  (move motor ‘xyz-x (position (offset home 20 inches))))
```

Web Server

- Supports HTTP requests, jQuery, and JSON
- Provides an s-expression syntax to emit HTML code
- Enables rendering of pages using Scheme for logic and emit HTML code for rendering the information
- Used by software engineers for:
  - Debugging devices remotely
  - Rapidly prototyping UI's since the web pages can call Scheme functions directly
  - Provide developer-only functionality without having to make special adjustments in the actual UI
- Used by non-software engineers for:
  - Pulling data from various prototype devices using JSON and pushing the information to other data collection systems. Example: validation tools for submitting data to the FDA
  - Querying data by typing in SQL directly into the webpage without having to worry about connections and having to install separate software

Lessons Learned

- Message passing concurrency is not a magic bullet. Concurrency is still hard, gen-servers can deadlock, and race conditions have to be handled correctly.
- The supervision structure was built with a lot of trial and error since we are probably the first or among very few using the OTP ideology for device automation.
- Automated and unit testing frameworks had to be built up by us.
- DSL gives non-software folks a lot of power. Conversely, when the system is being stressed and problems are discovered, system-level debugging has to depend a lot on the software team, since they are not proficient in Scheme, same with testing groups.

Lessons Learned

- Ramp-up time for new team members because of unique Scheme/Erlang language.
- IDE/debugging support. We don't miss it much since we have good logging, but it contributes to the slow ramp-up time for new team members.
- Hard to find support from the web community because of heavy customization.
- Problems with quality metrics like defect density. Scheme is very terse, so the defect density seems higher than for other languages.
- Be prepared for more scrutiny from the FDA, since this is different from the usual C/C++ or C# code bases, and we use other open-source tools like Emacs, SQLite, git, etc.

Take Away

- The instrument server uses message-passing concurrency to eliminate by design the race conditions and non-modularity of shared-state concurrency.
- Message-passing paradigm makes reasoning about concurrent processes easier.
- Fault isolation, the supervision tree, logging, and time-outs mitigate problems caused by non-determinism.
- Let-it-crash error handling philosophy leads to shorter code with fewer branch points and test cases.
- Strong typing and run-time checks provide cognizant failure instead of silent corruption.
- Mixing languages makes them more powerful and allows you to borrow the best bits of each one.
- DSL's provide the ability for rapid prototyping and experimenting for non-software developers.

References

- Jay Abraham (The MathWorks) and Paul Jones and Raoul Jetley (FDA/CDRH). Sound Verification Techniques for Developing High-Integrity Medical Device Software, 2009.
- Joe Armstrong. Making reliable distributed systems in the presence of software errors, 2003.
- Simon Peyton-Jones. Beautiful concurrency, *Beautiful Code*, 2007.

Contact Information

- Vishesh Panchal

  Senior Software Engineer, Interactive Intelligence, Inc.

  visheshpanchal@gmail.com | 574.302.6233

- Bob Burger

  Senior Staff Software Engineer, Beckman Coulter, Inc.

  rgburger@beckman.com | 317.808.4204