TOWARDS MODULAR LARGE-SCALE DARWINIAN SOFTWARE IMPROVEMENT

MICHAEL ORLOV

SHAMOON COLLEGE OF ENGINEERING

ISRAEL

GECCO GI 2018, KYOTO, JAPAN 2018年7月16日

2 THE HOLY GRAIL

- Automatically improve large-scale software systems
  - Which constantly increase in size
  - Which increase in complexity
  - Which are hard to evaluate X times per second
- Practical pathway to making evolution of such systems more practical and efficient:
  - Characterize large-scale systems
  - Propose modular evolution methodology (has costs and benefits)
  - Propose practical integration into programming languages and IDEs

3 WHAT ARE LARGE-SCALE SOFTWARE SYSTEMS?

- Very large amount of code
- Long startup and shutdown times
- Complex, hard to formalize logic
- Dependency on external systems with state (e.g., databases)

4 WHY IS EVOLVING LARGE-SCALE SW HARD?

- Current approaches require huge computational resources in order to scale:
- Large amount of code results in large search space
  - Not a problem per se, but drastically increases evaluation time
- Long startup and shutdown result in long evaluation time
- Complex logic makes system constraints hard to formalize
  - Evolution is expected to introduce bugs that are hard to detect
- External systems with state are difficult to control
  - Can we really replicate a database for each evaluation?
  - Unlike in e.g. testing, we cannot expect to create a limited mockup

5 MOTIVATING EXAMPLE

- Industrial software-based factory control system
- Impossible to start up and shutdown the control system for each evaluation
  - Too inefficient
  - Too insecure, probably against regulatory requirements
  - Likely requires shutdown of most factory processes and not just the control system
- An engineer could designate a specific module to be automatically improved
  - Example: a controller for a system of valves tasked with maximizing the throughput of a fluid through a set of pipes

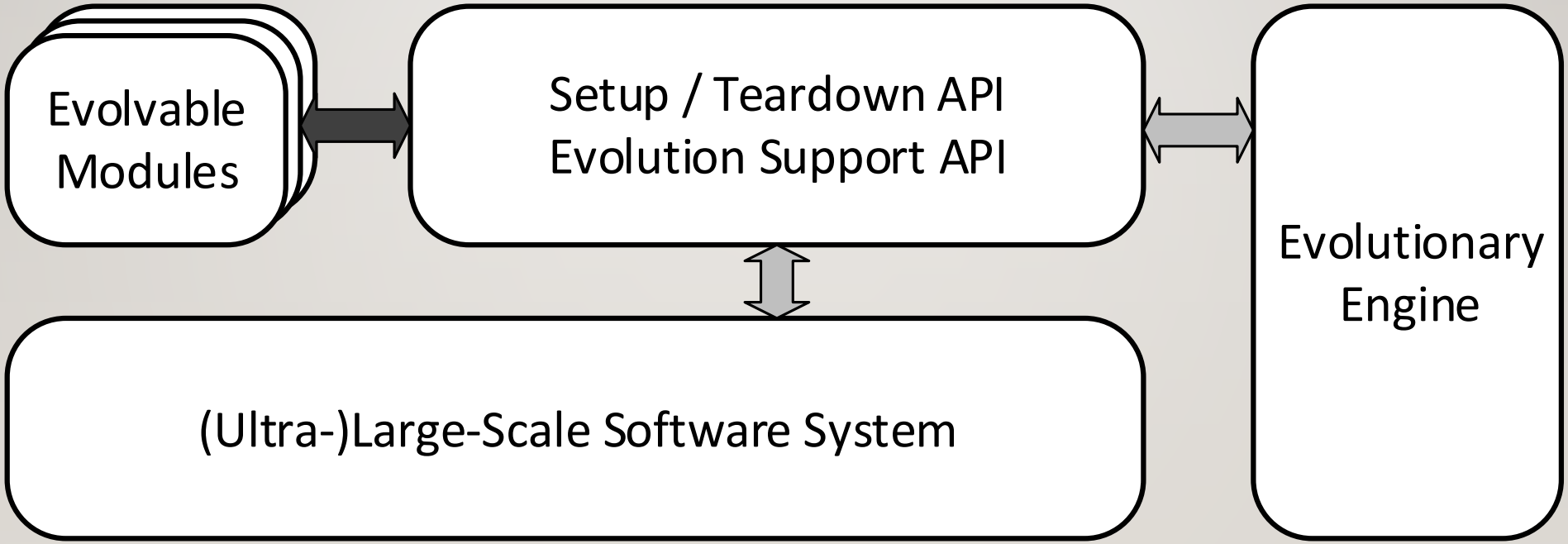
6 VALVES CONTROLLER IMPROVEMENT

- Important to remember: still a module within a large-scale software system
- Engineer needs to implement:
  - Quick setup and release of valves access during initialization and shutdown
  - Logging of current component specifications and outcomes
  - Fitness evaluation of current component performance during and after its operation
  - Restrictions on allowed operations by the component, preventing it from causing physical damage to the system
  - Security restrictions on the component to comply with regulatory requirements on unverified code — e.g., a sandboxed execution
  - Code size, memory and other resource-related strict and soft restrictions stemming from system limits
- Is such modular software improvement approach viable from managerial point-of-view?

7 SUGGESTED PATHWAY TO THE HOLY GRAIL

- Software designer should be central to the evolutionary process
- Unrelated code is excluded from the evolutionary process
- System startup and shutdown are excluded from the evolutionary process
- Focus is on evolvable modules
  - As defined and controlled by software designer

FORMALIZING MODULAR LARGE-SCALE SOFTWARE IMPROVEMENT



9 SUGGESTED PRACTICAL IMPLEMENTATION

- Model on existing successful frameworks (e.g., JUnit)
- Annotate existing code and add new annotated code
  - @EvolvableModule, @Setup, @Teardown, @Evaluate
- Evolutionary engine is a library
  - Used by the large-scale system, not the other way around
- Improvability can be considered as a system-level feature
  - Should be implemented at system level, like testability, verifiability, etc.

WHAT MODULAR SOFTWARE IMPROVEMENT IS AND IS NOT

- Requires stronger software engineer's integration
- Provides faster and more focused software improvement
- Not a way to improve software completely automatically
- Not intended for bug fixing or similar specialized tasks
  - These tasks already have their own strict search space environments