WORKSHOP

Certification Using the *New Approach to Safety*

Paul Albertella, Codethink

Summary

- The ‘new approach’: RAFIA
- Deterministic Construction Service (DCS)
  - Background
  - Purpose and principles
  - Goals and challenges
- Certification approach
  - Applying RAFIA to tools
  - Controlled process for all inputs
  - Providing and managing evidence
- Role of DCS in future work
  - Foundation of an ‘open source engineering process’
  - Supporting impact analysis and tool validation

The ‘new approach’: RAFIA

- **R**isk **A**nalysis **F**ault **I**njection and **A**utomation
  - Presented as “A New Approach To Software Safety..." at the last ELISA workshop
- Use STPA to analyse risks and specify safety requirements
  - Identify hazards and derive detailed safety requirements for software *and* wider system
- Derive tests to verify required software behaviour in target system
  - Test using historical versions of pre-existing software to provide confidence
  - Regression testing for updated software to verify that expected behaviour is still present
- Use fault injection to validate tests *and* system safety measures
  - Provoke hazards identified during analysis to verify tests and system-level mitigations
- Automate verification and use deterministic construction techniques
  - Use CI/CD to automate testing and require binary reproducibility of construction outputs

[1] *https://elisaworkshopmay2021.sched.com/event/j3T7*

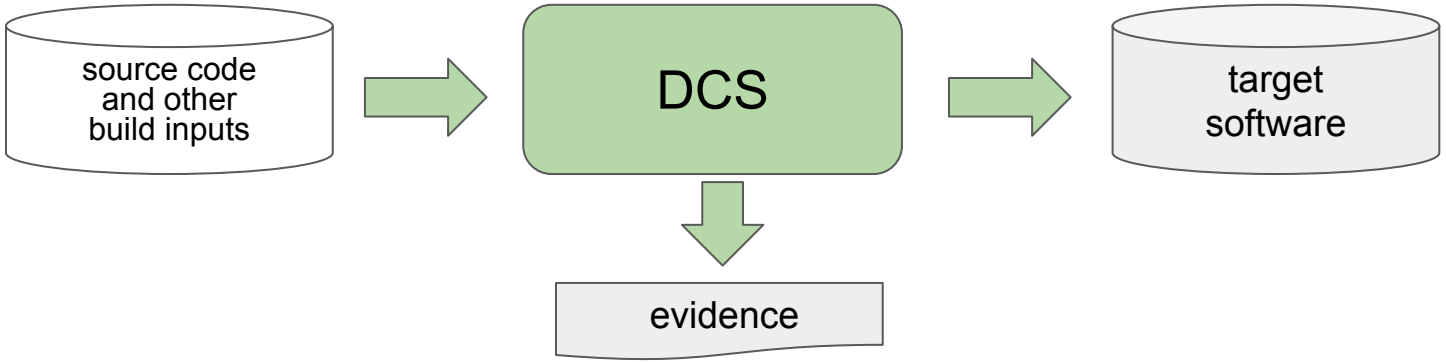
DCS: Background

- DCS: Deterministic Construction Service
  - Codethink reference implementation and design pattern, using open source tooling and a controlled CI process
  - Validates new and updated tooling as part of construction process for critical software components
  - Recently qualified to ISO 26262 / ASIL D by Exida
- Part of a continuing journey
  - Enabled by many years work on safety, construction and integration tooling at Codethink
  - An important step forward, but provides a foundation for work to certify Linux-based OS, not a complete solution

[1] *https://www.exida.com/SAEL-Safety/codethink-ltd.-deterministic-construction-service-reference-implementation*

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

DCS: Purpose



- Deterministic construction of software from controlled source code, build instructions, etc
  - For a set of well-behaved inputs, DCS constructs the target software as a binary fileset
- Re-run of DCS process reproduces *exactly* the same binary fileset
- Reproducibility is shown to be independent of the specific instantiation of DCS
  - Including host hardware, operating system, compilers and other tools
- Generate evidence required to support certification

Deterministic Construction principles

- Construction is the foundation for key engineering processes
  - Tools, processes and inputs used to build and verify the system
  - Build and test environments in which these processes are executed
  - Configuration and change management of these resources
- Use predictable characteristics to support verification and impact analysis
  - Binary reproducibility of artifacts and toolchain components enables cross-validation
  - Deterministic construction enables impact analysis with very fine granularity
- Use automated CI/CD process to drive safety processes
  - Provenance of inputs and evidence of impact analysis for changes
  - Traceability from requirement to test to test results
  - Configuration management aligned around CI process
  - Evidence required for certification is managed or generated by CI

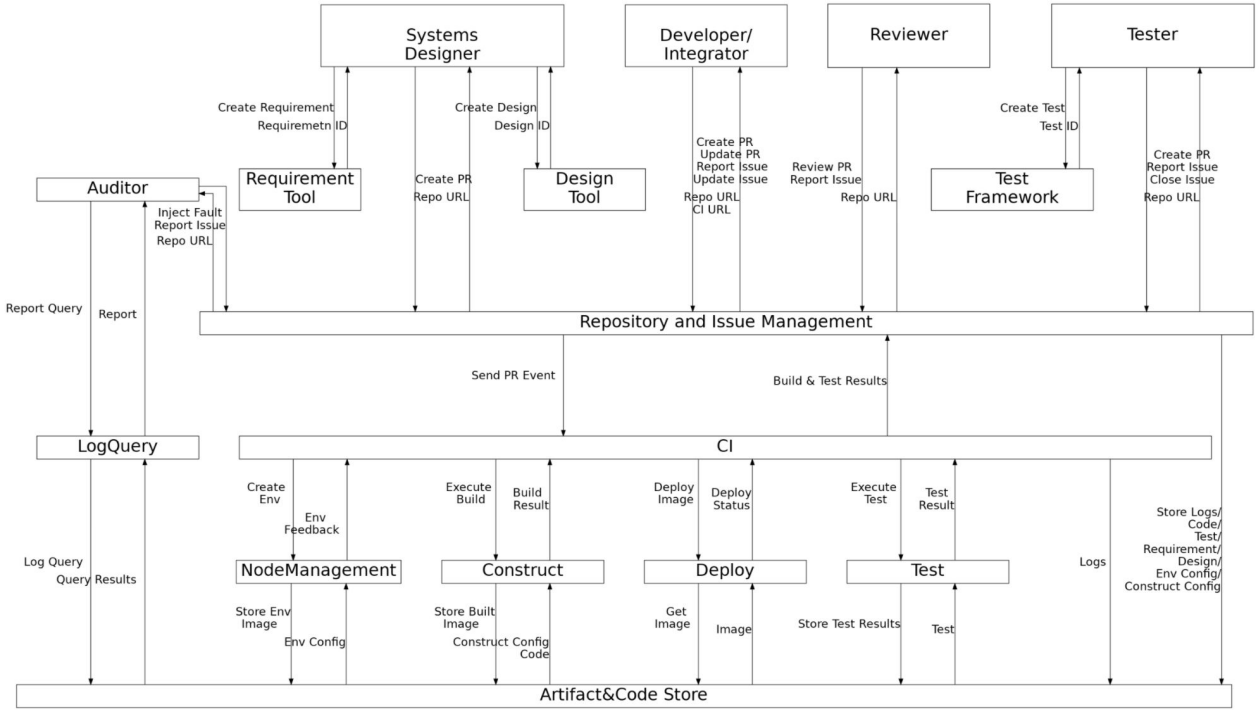
Goals and challenges

- Goals
  - Satisfy ISO 26262 process criteria with a CI-driven workflow
  - Establish a viable strategy for using open source tooling in safety
  - Apply RAFIA to a concrete project
  - Lay the foundations for developing a certifiable Linux-based OS
- Challenges
  - Open source projects do not follow the ISO 26262 “reference process”
  - How to classify tools and determine qualification requirements?
  - How to manage and present documentation to support safety assessment?
  - How to handle future updates to tools without endless re-certification?

Applying RAFIA to tools

- Use STPA to define safety requirements
  - Identify losses and system-level hazards/constraints for use case (CI tooling integration)
  - Specify system architecture, plus roles and responsibilities of individual tools
  - Identify how tool interactions can lead to hazards and associated loss scenarios
  - Define constraints that must be satisfied to prevent or mitigate hazards
- Implement tests, other verification measures and fault injections
  - Tests to show that individual components satisfy constraints
  - Process requirements for constraints that are not implemented by components
  - Provoke hazards with fault injections based on loss scenarios
- Construct, use and verify tools as part of automated process
  - Control the versions and revisions of tools and all other inputs to build / test
  - Use tools in controlled execution environments

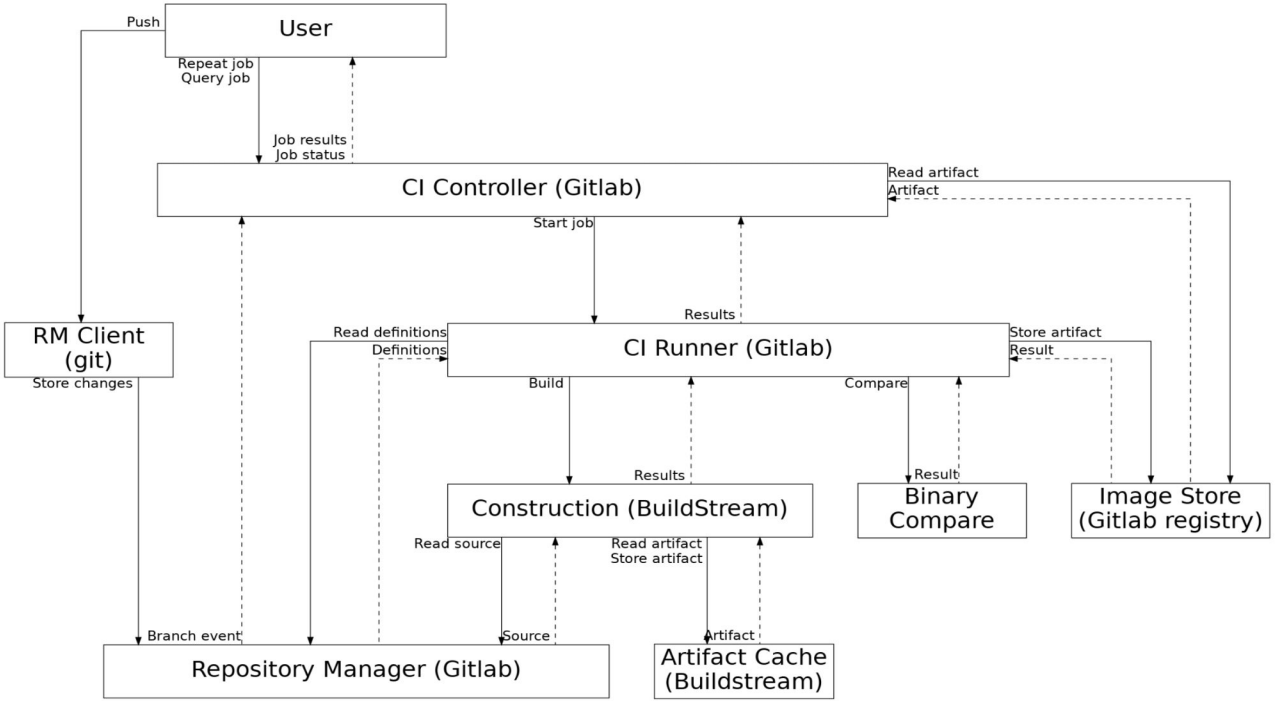
Control structure for CI-driven workflow



- Provides context for specific scope of DCS
- Defines system-level constraints for controlled process required by DCS

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

DCS control structure



- Defines tool roles and responsibilities
- Tools used in reference implementation are shown in brackets
- Used to identify *unsafe control actions* (UCAs) and *loss scenarios*

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

Recording and managing STPA results

- Record using YAML
  - Format is open source and described in stpatools[1]
  - Identifiers can provide links between data elements
  - Validate syntax and data against rules in a ‘schema’
- Manage under source control
  - Supports collaboration and review process
  - Also used to control changes to CI process (validation tools, syntax rules)
  - Generate human-readable documents and reports

```
Losses:
- Identifier: L-1
  Text: Loss of life
- Identifier: L-2
  Text: Loss of / damage to property
- Identifier: L-3
  Text: Loss of OEM brand reputation
- Identifier: L-4
  Text: System validation is not timely or cost-effective (Loss of mission)

Hazards:
- Identifier: H-1
  Text: >
      A tool that is used to construct the software deployed to a vehicle
      introduces a fault
  Losses: [L-1, L-2, L-3, L-4]
  Scenarios:
  - Identifier: H-1-LS-001
      Text: >
      A compiler or other tool used in construction generates a binary that
      does not behave as intended by the software developer
```


[1] *https://gitlab.com/CodethinkLabs/stpatools*

Constraints, tests and fault injection

- Constraints identified through STPA provide safety requirements
- Tests are implemented to verify that these constraints are satisfied
- Fault injections to verify that tests and system mitigations are effective
  - e.g. Simulate out of memory (OOM) to verify that build error is still detected
- Document constraints that must be applied by end-user in the safety manual
  - e.g. Users must not subvert the authentication and access control processes enforced via GitLab

```
- Identifier: C-012
  Text: >
    An error must be produced if any of the referenced source files cannot
    be accessed at build time.
  Hazards: [ H-6 ]
  Tests:
  - test_nonexistant_source

- Identifier: C-013
  Text: >
    An error must be produced when there is insufficient storage space
    where the build is taking place.
  Hazards: [ H-6 ]
  Tests:
  - test_insufficient_storage

- Identifier: C-014
  Text: >
    An error must be produced when there is insufficient memory
    where the build is taking place.
```

Controlled process for all inputs

- Change management of *all* construction and verification inputs
  - All inputs stored in git repositories with changes managed by a Repository Manager
    - Reference implementation: Gitlab, using *Merge Request* process
  - Construction and verification measures defined per repository
    - Including integration tests that consume inputs from other repositories
  - Apply the same principles for tools, dependencies, build/test execution environments
- Controlled ingress for *all* upstream inputs
  - Mirror upstream project repositories in resources under your control
    - Protect release branches to block any attempt to rewrite history
    - Reference implementation uses Lorry[1]
  - Consume only specific revisions (e.g. identified by SHA-1 or tag)
    - Changes to revision references must be verified and approved

[1] *https://gitlab.com/CodethinkLabs/lorry*

Controlling execution environments

- Environment for construction and verification can affect outputs
  - Create container images to provide consistent environment for build and test jobs
  - Control inputs for these and construct as for all other software
    - e.g. Docker compose scripts stored in git repository
  - Update images via CI only, to ensure that changes have been verified and approved
    - Reference implementation uses Gitlab Container Registry
- Sandboxed build environment
  - Use build orchestration tools to obtain inputs for each build stage
  - Control environment variables, filesystem, etc
  - Block access to external networks to avoid uncontrolled inputs from Internet

Enforcing and exploiting reproducibility

- Constructed software must be *exactly* reproducible
  - Controlled inputs + controlled build + controlled environment = identical binaries
  - Principles established and explored by Reproducible Builds project[1]
  - Enforce this by verifying reproducibility regularly
- Once established, the property can be exploited
  - Identical outputs with *different* inputs means that the changes:
    - To *target software* inputs have no impact, so we can avoid unnecessary testing
    - To *tools* or *environment* have no impact, so we can be confident in updates
  - Can verify artifact cache integrity by rebuilding from source before a target software release
    - Tests performed using cached artifacts do not need to be repeated

[1] *https://reproducible-builds.org/*

Providing and managing evidence

- Apply controlled process to *all* inputs to certification assessment
  - Includes documentation, requirements and certification criteria as well as build inputs
    - For reference implementation: store *everything* in git repositories managed by Gitlab
- Map evidence to certification criteria
  - Individual criteria based on the applicable safety standard (ISO 26262 for DCS)
  - For each requirement, assert *how* it is satisfied
  - Link to document, source, test or CI-generated output that provides supporting evidence
- Manage criteria, assertions and evidence alongside software
  - Updates managed by the same CI-driven change control process
  - Verify that supporting evidence links are valid and up-to-date
  - Trace requirements to tests, and to test results
  - Generate human-readable reports for safety-assessors

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

Tool qualification *vs* product certification

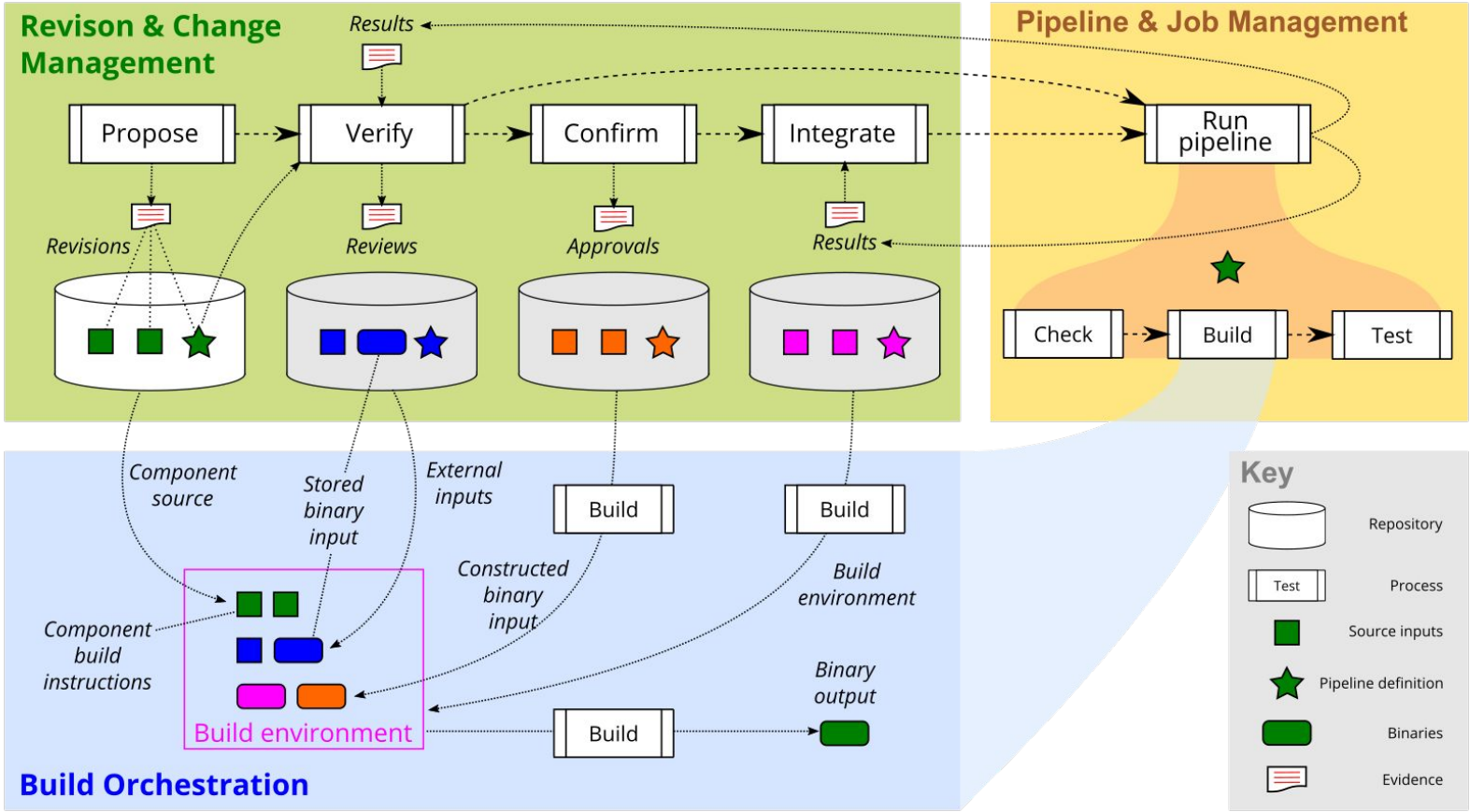
- DCS certification is based on a tailored ISO 26262 process
  - Omits inapplicable requirements (e.g. hardware design)
  - Includes management processes
    - e.g. Functional Safety Management, Change Management, Impact Analysis
  - Includes some concept phase and software development criteria
    - e.g. HARA, software implementation
- Same approach can be extended and applied for product certification
  - Use STPA to define safety goals and derive detailed safety requirements for components
  - Manage *all* inputs relating to certification criteria in a coordinated process
    - ‘Documentation as code’ principle
  - Enable iterative refinement of safety case alongside software development
    - e.g. Add new loss scenarios identified during system testing

Role of DCS in future work

- Basis for qualifying and validating open-source toolchains
  - Determine impact on constructed output for new tools
    - Tools with no impact on deployed software (TI1) do not require qualification
  - Validate tool upgrades by determining impact on known/verified previous builds
- Implementation and validation of controlled process
  - Allows a new implementation of the process to be verified
    - For a different instance of the same set of tools
    - For a different toolset that meets the same requirements
- Support qualification of other TI2 tools (e.g. compiler)
  - TI2: “can introduce or fail to detect errors in a safety-related item or element being developed”
  - Use DCS to build and analyse impact; use RAFIA to define and verify safety requirements

Applying the approach to a Linux-based OS

- Design pattern for using and building open source inputs
  - Independent from upstream projects, but able to consume updates and determine impact
  - Equally applicable to deployed software
- Coherent and defensible specification of certification scope
  - Specific integration of specific FOSS inputs with a specific purpose
  - Provides model for managing and generating supporting evidence
- Demonstrates viability of RAFIA approach
  - Safety analysis feeds directly into development process
  - Derivation of detailed safety requirements makes implementation and testing goals clear
  - Iterative refinement of safety analysis supports both product development and maintenance



ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

Verifying STPA results using CI

- Check YAML syntax for ‘well-formedness’ rules
  - Using yamlint; avoids errors and ambiguities resulting from invalid document structure
- Check data against rules in a ‘schema’
  - Implemented as a Python script (open source and available in stpatools[1]) for DCS; could be a JSON schema
  - Check rules and log errors or warnings in CI job
- Reject changes if verification jobs fail
  - Change cannot be merged into ‘main’ branch of repository until all checks pass (no errors)
- Check or generate ‘human-readable’ documents
  - Automate generation of documents from YAML
  - Either generate in CI (output as part of job) or check that version stored in repo matches generated output

[1] *https://gitlab.com/CodethinkLabs/stpatools*

License

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS