Automated Design Conformance during Continuous Integration

Robert Nord, Ben Cohen, Shane Ficorilli, James Ivers, John Klein, Lena Pons, Chris Seifried
Software Architecture Enables Our Ability to Innovate

The software architecture community has evolved a body of knowledge that guides design and analysis.

This body of knowledge includes:
- design principles
- reference architectures
- architectural design patterns
- deployment patterns
- tactics
- externally developed components
Implications for the System

The degree to which a system meets its quality attribute requirements is dependent on architectural decisions.

However, architecture can only permit, not guarantee, any quality attribute.

For the implementation to exhibit the quality attributes engineered at the architectural level, it must conform to the architecture.
Software Nonconformance Problem

- Design
  - Extensible, Composable
  - ≠
- Implementation
- Baseline Delivery
- Planned V.2 Release
  - Depends on Extensibility and Composability
  - Discover Nonconformance
  - Remediate…
- Actual V.2 Release
An automated design conformance checker integrated into a continuous integration workflow will reduce time to detect violations from months or years to hours.

Automation enables early detection and allows remediation before the violation gets “baked in” to the implementation.

Detection of nonconformances allows program managers to hold developers (contractor or organic) accountable.
Create an Automated Design Conformance Checker

A design conformance checker automatically checks that the source code reflects the intended design and reports nonconformances.

Recommendations correctly identify nonconformance, precision > .90 and detect at the commit that introduces nonconformance > 90%

Apply developer feedback to improve accuracy and significance within project contexts.
Automated Design Conformance during Continuous Integration

Building on Code Analysis, Software Architecture, Machine Learning, and Continuous Integration

Automated Design Conformance Checker

Extract Design From Code

Source Code

buildCodeGraph → predictDesignConstructs → buildDesignFragment

As-implemented Design

Intended Design

Canonical Design Knowledge

Check Conformance

Non-Conformances
**Machine Learning for Source Code**

Initial applications applied off-the-shelf machine-learning tools with hand-extracted features.

Subsequent applications use the source code itself within machine learning drawing inspiration from natural language processing (NLP).

Current applications promise new machine-learning models informed by programming-language semantics.

What Is a Good Feature Set for Architecture Design?

Representing the code as a graph of entities provides insight into structure.
Modeling context in the form of relations can improve prediction performance.
Design constructs are traceable to a wider range or larger portion of the code base.

Common intermediate representation of object-oriented design.
Extensible representations allow data from different sources to be integrated.

Code-Design Abstraction Gap

Build Graph

Static code analysis tool extracts structural information from C++ object-oriented code.

Sample graph sizes:

**Blobby Warriors** [github.com/visusnet/Blobby-Warriors](github.com/visusnet/Blobby-Warriors)
- 48K code lines
- 8,799 nodes and 50,411 relations

**Hotspot (Qt framework)** [github.com/KDAB/hotspot](github.com/KDAB/hotspot)
- 8K code lines
- 2,648 nodes and 11,427 relations

**Sample Industry System**
- 1,587K code lines
- 274,199 nodes and 1,057,595 relations
Engineer Features

Structural and behavioral features link elements though relations.

Structural

- **Degree of Accessibility, Virtuality**
  \[
  \text{interfaceMethods}(c) = \{ m \mid m \in M_{\text{DEC}}(c) \land \text{Public}(m) \land \text{Abstract}(m) \}\]

- **Association**
  \[
  \text{methodParam}(c_1, c_2) = \text{true} \iff \exists p \in \text{PAR}(M_{\text{DEC}}(c_1)) \land \text{Type}(p, c_2)
  \]

Behavioral

- **Invoker/invokee**
  \[
  \text{toSibling}(c_1, c_2) = \{ \exists c \in C \mid (\ \text{Super}(c_1, c) \land \text{Super}(c_2, c)) \}\]

Check Conformance

Design fragments represented as graphs enable automatic detection.

Detect nonconformances

- Check graph of extracted design fragment for agreement with intended design fragment to locate inconsistencies.
- Augment with checks against canonical design knowledge relevant to design fragment.

“Non-adjacent processing steps do not share information”

“Explicit storage of intermediate results … is error-prone”
Continuous Integration Workflow

Detecting nonconformances produces greatest value when issues are exposed close to the time of injection.

Integrate with Jenkins CI tool to enable an empirical evaluation of the use of automation.

Use developer feedback in rating each nonconformance to improve results.

- **Correctness**—Improve design extraction by providing new labeled data.
- **Significance**—Improve adaptive filtering by capturing context-specific rules and exceptions.
Looking Ahead

**FY20**
- Build out infrastructure: representation, features, design knowledge, and conformance.
- Assemble open source data and initial analyses.

**FY21**
- Broaden the palette: more design knowledge and conformance checks.
- Implement adaptive filtering.

**FY22**
- Fine-tune conformance checking.
- Validate with experienced developers.
- Ready to pilot conformance checking for C++ software.
Next Generation Automation for Software Evolution

Advance the state of the art where automation can

- keep software aligned with needs
- bring projects back into alignment
- realize changes sketched by developers in the language of design
