

# DISCUSSING "SOFTWARE ANALYSIS: A ROADMAP" BY JACKSON AND RINARD

Daniel Popescu



October 10, 2006

# SOFTWARE ANALYSIS: MOTIVATION

Observed Trend: Software systems become more and more part of our environment. Software systems become ubiquitous and is used in more safe-critical systems.

⇒ We need to "get it right."

Software Analysis might help us to mitigate these risks providing us guarantees and additional information about our system.

# THE TERM "ANALYSIS"

Jackson and Rinard use the term **analysis** *for the extraction of behavioral information from software, represented as an abstract model or code.*

The following analysis domains are out of scope:

- Human Factors (e.g. usability)
- Syntactic Properties (e.g. for identifying undesirable couplings between modules)
- Compiler Analysis for Optimizations

# ANALYSIS OF MODELS

*Code* is a poor construction model for

- domain knowledge
- environmental assumptions
- design rationale

*Models* can be used to

- separate concerns more cleanly
- articulate key properties
- find defects earlier

Example: ADL Darwin, Problem Frames, ...

# ANALYSIS VS. DESCRIPTION

Research about abstract models is divided into "two camps": models for analysis vs. models for description.

Models for Analysis: Main motivation for models is to explore the consequences of design decisions by analysis, e.g. Theorem Prover

Models for Description: Main motivation is to precisely record the design intention of a developer. It is not so important to find "the hidden showstopper flaw."

However, are these two dimensions incompatible? Jackson and Rinard argue that both dimensions are synergetic and that every model should be analyzable. (Counter-example: UML)

# GLOBAL VS. LOCAL MODELS

Jackson and Rinard argue that two kind of models dominate software description: object models and state transition diagrams. (I do not agree.)

## Object models

- show the syntactic structure of the code
- are data intensive
- describe the global relationships

## State transition diagrams (local models)

- are control intensive
- describe state of an object and interaction amongst objects
- Challenge: concurrency

Model checking work has focused on local models.

# SIMULATION VS. CHECKING

In a solution space of billions of possibilities, is it better to simulate or to check a model?

Simulation: When building a model incrementally, it is easy to make mistakes that simulation immediately exposes. Early models reduces the subtle design flaws.

Comment: The paper does not explicitly define the difference between simulation or checking. Why cannot checks be made early?

# VERIFICATION VS. REFUTATION

Checking can rarely be fully automated, since any modeling language that is rich enough to be useful is likely to be undecidable.

Verification:

- Finding a proof for a given property. If no proof is found, the property may not hold.
- Application: Theorem Prover

Refutation:

- Trying to refute the given property by finding a counterexample.
- If no counterexample is found, either
  - the artificial set bound was inappropriate
  - or property does hold.
- Application: Model Checker

# DECLARATIVE VS. OPERATIONAL STYLE

Most languages are declarative

- invariants and operations are written as logical formulas
- can be partially: no need to determine the behavior completely
- incrementally, since they can be partially
- separation of concerns
- Challenge: Determining states for analysis
- Examples: Z, VDM, Larch

Operational languages

- implementation bias
- defining state transitions
- Examples: Statecharts

# ANALYSIS OF CODE

Current modeling languages have an Achilles heel: the lack of any enforced or checked correspondence with the actual implementation.

Therefore, models are often not useful, because they cannot be mapped to the implementation or they cannot be kept synchronized.

Jackson and Rinard expect that modeling entities will become a single construct in implementation languages.

Coupling these viewpoints will also facilitate program analysis.

# STATIC VS. DYNAMIC

*Static analyses* analyze the program to obtain information that is valid for *all possible executions*.

- many possible false positives, since errors are reported of unreachable states
- Prediction: will become increasingly important in the future.
- Enabler:
  - use of clean languages
  - increased hardware capabilities
  - more deployed safe-critical systems

*Dynamic analysis* instrument the program to collect runtime information. The results are valid for *observed run*.

- detailed information is easier to obtain
- Challenge: How is the code instrumented?

# SOUND VS. UNSOUND

## Sound static analysis

- can produce guarantees which hold on
  - all program executions. (static)
  - analyzed execution. (dynamic)
- depends on underlying programming language

## Unsound analysis

- does not provide completeness or correctness guarantees.
- is easy to implement
- efficient
- good starting point for further investigations

Example: Pointer Analysis vs. locating direct assignments

# SPEED VS. PRECISION

Trade-off in static analysis between:

- speed
- precision

Trade-off is based on discussion about

- Flow-sensitive vs. Flow-insensitive
- Context-sensitive vs. Context-insensitive

# FLOW-SENSITIVE VS. FLOW-INSENSITIVE

## Flow-sensitive analysis

- takes the execution order of the program's statements into account
- e.g. shape analysis techniques (detailed information about object referencing relationships)
- attainable through encoded design information in code
- higher precision of results

## Flow-insensitive

- does *not* consider execution order
- e.g. insensitive pointer analysis
- will stay popular for reverse engineering of legacy systems
- more efficient

# CONTEXT-SENSITIVE VS. CONTEXT-INSENSITIVE

Programming language constructs like procedures can be used in different context.

## Context-sensitive

- produces different result for each different context
  - reanalyze for every context
  - analyze once - obtain *parameterized* result (compositional analysis)
- exponential worst case complexity
- more precise

## Context-insensitive

- produces a single result that can be used in all context
- polynomial worst case complexity

# MULTITHREADED VS. SINGLETHREADED

Most classic program analysis techniques were developed for singlethreaded sequential languages.

Singlethreaded:

- code runs on one single sequential machine

Multithreaded:

- code uses explicitly parallel constructs
- instruction streams may interleave
- analyzing all interleaving paths *not* possible (exponential increase of paths)
- alternative flow-insensitive analysis (unreliable results)

Jackson and Rinard anticipated Future Trend: *Future modeling languages will contain expressions or policies for enabling analysis.*

# DISTRIBUTED VS. LOCALIZED

Another important source of *concurrency* exists between *distributed components* executing on different machines.

Jackson and Rinard anticipated Future Trend: Development of code analyses that are designed for programs expressed as interacting distributed components.

# MODEL-DRIVEN CODE ANALYSIS

Idea: Connecting models and code for analyses

Value of models increases if they can be connected to code.

Models can also amplify the power of code analysis.

# MODULAR ANALYSIS BY INDUCTION

Instead of analyzing the whole code at once, analyze the program at the granularity of components.

Larger programs can be analyzed, since irrelevant code statements can be hidden using information hiding at the component level.

Jackson and Rinard anticipate that modeling languages become a popular way for specifying invariants required for code analyses.

# ANALYSIS USAGE CONTEXT AND COMMUNICATION OF RESULTS

Multiple Software Analyses: Engineers need different precision in different situations and in different phases of the software development.

⇒ Results of analysis must match expected level of precision.

Software analysis results are easier to understand with models.

Future challenge: Finding good models or visualizations to communicate results.

# WEAKNESSES OF THE PRESENTED PAPER

- Constructed dichotomies (Simulation vs. Checking)
- Dichotomies are often not equally presented (Declarative vs. Operational Style)
  - Authors almost only talk about one property
- Disagreeable assumptions (Global vs. Local Models)
- Compared concepts are not always defined (Simulation vs. Checking)

# STRENGTHS OF THE PRESENTED PAPER

- Good selection of relevant properties for software analysis
- Separation in two levels
  - Models
  - Code
- Importance of connecting models and code

# RELEVANCE TO EMBEDDED SYSTEMS

Software analysis is relevant for *all* software engineering domains.

⇒ Software analysis is important for embedded systems

Embedded Systems have their own requirements for software analyses techniques, since they require domain-specific models.

Sound results provided by software analysis is especially important for safe-critical systems, which are mostly embedded systems.

# THANK YOU

Thank you for your attention.

Final Discussion