Formal Verification and Static Analysis

Matching Expertise with Industrial Needs

Marius Minea

Institute e-Austria Timișoara

INTAS project meeting, 10-11 Dec. 2006

Verification in Industry: Needs and Challenges

Two aspects:
– *critical/result*: find and correct a bug with significant impact
– *day-to-day/process*: improve quality/cost of software development

Challenges:
– need pushbutton tools usable within current development process
– need “intelligent” tools that hide formalism and make right choices
– in case studies, need human resources for system understanding
– a lot of overhead effort spent in non-verification tasks
– difficult to change development for existing projects
– difficult to argue cost-efficiency of new approach with hard data

Formal Methods in Software Development: Impact Points

Requirements analysis

- formally specified ⇒ identify omissions / inconsistencies
- min.: rigorous specs ⇒ avoid effort duplication in testing

Modeling / Design

- formal modeling and automated code generation
- min.: models in sync with code; matching model & code semantics

Development

- critical parts of code verified
- min.: static analysis for potential bugs; verified manual models

Testing

- model-based test generation
- min.: automated test generation / relevant coverage guarantees

Case Study 1: Telecom Verification (Alcatel)

Formal verification a communication block written in SDL;
evaluate usability of approach on larger scale

Case study features:

- code written in a specialized language (SDL)
- but with formal semantics, and translatable to verification tool (IF verification toolkit from VERIMAG Grenoble)
- SDL used directly for generation of C code for actual system

Model characteristics:

- single process, 8 procedures
- two dozen messages
- 1500 lines of SDL code (excluding comments)

Issues in Verification

System specifics

– C functionality within special SDL comments

⇒ semi-automated translation

Abstraction

– many message fields; some large in width (e.g. addresses)

– eliminate irrelevant fields; abstract others (e.g. null/non-null)

– some may be done automatically (slicing), others by hand

Model Size

– 50 control states, 25 data bits ⇒ ca. 400 million potential states

Actual state space:

– 875 000 states, with inline procedure expansion

– 140 000 states, with modeling of procedure call/return

– 30 000 stable states, with collapsing of transient states

Performance: average 1 minute / 30MB / spec. ⇒ still scalable

Case Study 2a: static analysis for C

Need: custom analysis task
– verification of buffer copy routine with hundreds of structure fields
⇒ hard to do by manual code inspection
the location of a previously discovered / corrected error

Approach: CIL infrastructure for analysis of C code (UC Berkeley)
– wrote custom analysis that handled vast majority of cases
– 1% of fields remaining for manual inspection

Conclusion:
– specific nature of problem hardly justifies general tool
– but a higher-level property specification language could be useful
(problem could be phrased as an instance of use-def analysis)

Case Study 2b: verification for C

Verification of signaling & circuit management code for a phone switch

- code written in CHILL, automatic translation to C
- analysis with BLAST symbolic model checker (UC Berkeley)
- 5400 lines of C code relevant to suspected error scenario
  (separated semi-automatically from body of system code)
- model involved double indirectation chain stored in arrays
  (vulnerable to index overflows)
- BLAST found potential error scenario by overflow of 4-bit value

Case Study 3: error detection (embedded)

Project in execution
– elusive errror, hard to reproduce, hardware-in-the-loop needed
Mixed approach, using several techniques
– model checking with BLAST: no errors found
(insufficient alias analysis)
– static analysis with Splint: found buffer overflow
but likely not triggerable in practice
– dynamic analysis with Valgrind: pending
– schedulability analysis for RTOS tasks: another possibility

Technology Expertise / Needs: Static Analysis

– successful option for detecting a large class of bugs
  buffer overflows (memory corruption), uninitialized values
  also property checking (using automata – close to model checking)
– scalable to large amounts of code

Problems to address
– friendly user interface, allowing code comprehension
  e.g. possible source of overflowed value
– easily specifiable custom analyses
– modular usage / tool combinations / user guidance
  e.g. annotations for conditions known (not) to be true
– VC generation/discharging by specialized techniques
  (polynomial invariants – Laura Kovács)

Technology Expertise / Needs: Compositional Reasoning

– reasoning about components

– deducing properties of system from properties of its components without the need to construct the entire system model

$$\frac{\begin{array}{c} M1 < S1 \\ M2 < S2 \end{array}}{M1 \parallel M2 < S1 \parallel S2}$$

Such compositional rules are valid for many formalisms, but

– usually models are not built to function in *any* environment

⇒ model M may not refine spec S when standalone

⇒ need *assumptions* about context for *guarantees* about behavior

Circular Assume-Guarantee Reasoning

Decompose proof that an implementation refines a specification

Chandi & Misra'81, Abadi & Lamport'93, Alur & Henzinger'95, McMillan'97

$$\frac{\begin{array}{c} A1 \parallel B2 < A2 \parallel B2 \\ A2 \parallel B1 < A2 \parallel B2 \end{array}}{A1 \parallel B1 < A2 \parallel B2}$$

Refinement often holds only under environment *assumption*

$A1 < A2$ and $B1 < B2$ may not hold:

A2: $x = 0$ A1: $x' = y \ (x_0 = 0)$

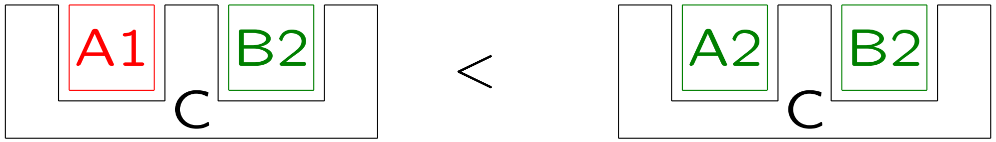
B2: $y = 0$ B1: $y' = x \ (y_0 = 0)$

Assume-Guarantee: components in a context

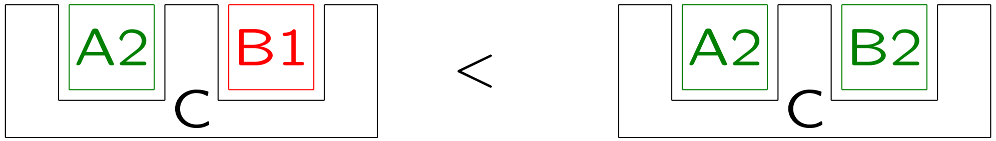
Refinement goal: context with two implementation components

Premises: individually replace components with specification

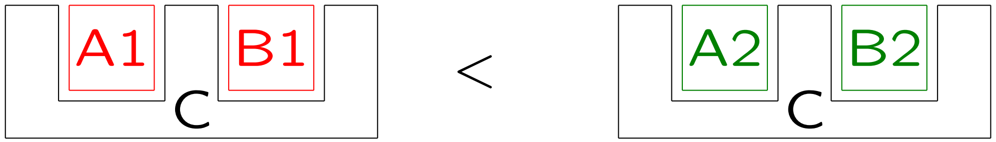
$C[A1,B2] < C[A2,B2]$



$C[A2,B1] < C[A2,B2]$



$C[A1,B1] < C[A2,B2]$



Potential work: Combining model checking and theorem proving

With assume-guarantee reasoning
- model-check individual refinements
- theorem proving (or at least proof assistants) for decomposition

For static analysis
- model checking finite program model
- discharging verification conditions by theorem proving

For specifications
- consistency checking (theorem proving, SAT checking)
- or custom (model checking) algorithms
  (e.g. Message Sequence Charts)