From Formal Verification To Synthesis

Rajeev Alur
University of Pennsylvania
http://www.cis.upenn.edu/~alur/

Software at Scale Workshop, August 2010

Software Design

Libraries → Compiler → Executable → Platform → Analysis Tool → Tests
Program → Specifications
Programming Technology

- Libraries
- Program
- Compiler
- Executable
- Platform

High-level programming abstractions
(object-oriented, synchronous, domain-specific...

Semantics-preserving transformations
(low-level optimizations, type inference ..)

Verification Technology

- Program
- Specifications
- Tests
- Analysis Tool
- Executable
- Platform

Automated verification
(model checking, static analysis, specification-based testing ..)
Challenges

- Today’s software is commonly buggy, fragile, untrustworthy...
- Verification/testing done after design
  - Costly system design cycle
  - Many reported bugs not fixed
- Computing power is transforming many engineering disciplines with the notable exception of programming itself

Opportunities

- Enormous computing power available on desktops of today’s programmers
- Impressive strides in formal verification technology
  - Highly optimized SAT solvers that can solve real-world problems
  - Off-the-shelf tools for static analysis, machine learning...
- Research driven by external demand
  - Receptive industry
  - Shifting goal of system design from performance to predictability
Algorithmic Synthesis

- Mapping "what" to "how"
  Derive an executable implementation from a high-level specification
  Correct-by-construction design

- Church (1963): how to synthesize sequential circuits from temporal-logic specifications

- Harel (2008): Can programming be liberated?

- Computational problem: Find values for controlled decisions so that for all choices of uncontrolled decisions (e.g. inputs), spec is satisfied
  - Quantifier alternation, Games, finding winning strategies
  - Note: No solution may exist to such a synthesis question

How computers can help programmers?

- Program Sketching
  Given a program with holes and assertions, tool fills in the holes

- Concurrency Synchronization
  In sequential code for data structures, tool inserts minimal synchronization to design linearizable concurrent data structure

- Specification Mining
  From library code, tool discovers behavioral specs

- Learning by Examples
  From positive and negative examples of scenarios, tool infers the necessary program logic
How computers can help programmers?

- Program repair
  Verification tool not only finds a counter-example, but recommends a fix by analyzing source of bug

- Controller Synthesis
  From temporal logic specifications, low-level control laws are generated for reactive planning for robot motion

- Component Composition
  From component interfaces, add glue logic for interaction

Synthesis in idealized form is arguably unrealistic, but plausible in these limited forms, allowing more “active” role for computer in programming

User Guided Synthesis

1. Computer and programmer collaborate
2. Synthesizer discovers new artifacts
3. Computational tasks may be heavy-duty
Talk Outline

✓ Motivation

منذ Representation dependence testing via program inversion
   Joint with A. Kanade, S. Rajamani, G. Ramalingam (FSE’10)

☑ Synthesis of behavioral interfaces for Java classes
   Joint with P. Cerny, P. Madhusudan, W. Nam (POPL’05)

☑ Conclusions

Representation Dependence

Client program P using a data structure has representation dependence if P behaves differently on two distinct, but logically equivalent, values
Example from Windows DirectDraw API

typedef struct _ddsurfacedesc{
    dword height, width, pitch;
    lpvoid surface;
}

Spec allows pitch >= width
Documented in text

Client computes (i,j)-th entry to be d.surface[i*width+j]
Implicitly assumes pitch=width
Bug undetected over multiple releases

Testing for Representation Dependence

- **Specification:** Equivalence relation over a data type T
  Equivalent values represent same logical content

- **Goal:** Given client C and test input d, generate multiple inputs d' equivalent to d, and check if C(d) equals C(d')

- **Motivation:** Detect bugs that may show up only later during version upgrades

- **Challenges**
  - How to specify equivalence?
  - How to automate generation of equivalent test inputs?
Normalization Routines

User specifies equivalence by writing a function $f$ that maps data values of type $T$ to canonical values of type $T'$: $d$ and $d'$ are equivalent iff $f(d) = f(d')$

For ddsurfacedesc, the normal form is two-dimensional array without slack bytes (fields: height, width, data)
Normalization Routine

Normalization function $f$:
input $d$, output $n$

$n$.height = $d$.height;
$n$.width = $d$.width;
for ($i = 0; i < n$.height$) {$
  for ($j = 0; j < n$.width$) {$
    n$.data$[i][j] = d$.surface$[d$.pitch$*i + j]$
  }$
}$

Hypothesis: Writing C code for normalization is easier than giving a correct, precise logical spec

Program Inversion

To generate equivalent test inputs, we need inverse $g$ of normalization routine $f$: given $d$, compute $g(f(d))$

$g$ is nondeterministic
Inverse Function

Inverse function $g$: input $n$, output $d$

```plaintext
ensure(d.pitch : d.pitch >= n.width);
d.height = n.height;
d.width = n.width;
for (i = 0; i++ < d.height) {
    for (j = 0; j++ < d.width) {
        d.surface[d.pitch*i+j] = n.data[i][j]
    }
}
```

Automated Program Inversion

- **Key insight**: “Sketch” of inverted program is same as normalization routine (same loop structure)

- Inversion done statement by statement (locally)

- **Need** forward static analysis to compute which input vars are determined by output vars at each program point
  “Free” vars replaced by calls to “ensure” with constraints

- **Current focus**: programs with iterators over arrays

- **Challenges**
  - Constraint propagation over straight-line blocks of code
  - Indirection in array indexing (e.g. $x[y[i]]$)
TIFF Case Study

Multiple representations of same matrix of pixels possible
- Image may be stored left-to-right / right-to-left
- Image may be stored top-to-bottom / bottom-to-top
- Slack bytes possible

1. Wrote normalization routine
2. Automatic program inversion
3. Generated multiple equivalent variants of a TIFF file
4. Tested following open source software
   - Picasa 3.6
   - Open Office 2.0.4
   - GIMP 2.2.13
   - KView 3.5.4
   - FastStone 3.6

Summary of Results of Testing

- Effect of varying the number of rows per strip:
  All clients process image correctly

- Effect of varying the orientation
  Open Office and GIMP display image incorrectly

- Effect of physically reordering logically adjacent strips, in conjunction with change in orientation:
  Picasa displays image incorrectly

Caveat: Bugs detected by human observer of images
Talk Outline

✓ Motivation

✓ Representation dependence testing via program inversion

吕布 Synthesis of behavioral interfaces for Java classes

☐ Conclusions

Static Interfaces for Java Classes

package java.security;
...
public abstract class Signature extends java.security.SignatureSpi {
<variable declarations>
protected int state = UNINITIALIZED;
public final void initVerify (PublicKey publicKey) […]
public final byte[] sign () throws SignatureException { ….}
public final boolean verify (byte[] signature) throws SignatureException { ….}
public final void update (byte b) throws SignatureException { …}
.. }


Behavioral Interface

- Methods: initVerify (IV), verify (V), initSign (IS), sign(S), update (U)

- Constraints on invocation of methods so that the exception signatureException is not thrown
  - initVerify (initSign) must be called just before verify (sign), but update can be called in between
  - update cannot be called at the beginning

```
public Object next() {
    ...
    lastRet = cursor++;
    ...
}
```
```
public Object prev() {
    ...
    lastRet = cursor;
    ...
}
```
```
public void remove() {
    if (lastRet == -1)
        throw new IllegalExc();
    ...
    lastRet = -1;
    ...
}
```
```
public void add(Object o) {
    ...
    lastRet = -1;
    ...
}
```

AbstractList.ListItr

Behavioral Interface
Interfaces for Java classes

- Given a Java class $C$ with methods $M$ and return values $R$, an interface $I$ is a function from $(M \times R)^*$ to $2^M$.
  Interface specifies which methods can be called after a given history.

- Given a safety requirement $S$ over class variables, interface $I$ is safe for $S$ if calling methods according to $I$ keeps $C$ within $S$.

- Given $C$ and $S$, there exists most permissive interface that is safe wrt $S$.

- Interfaces can be useful for many purposes:
  - Documentation
  - Modular software verification (check client conforms to interface)
  - Version consistency checks

- JIST: Automatic extraction of finite-state interfaces
  Phase 1: Abstract Java class into a Boolean class using predicate abstraction
  Phase 2: Generate interface as a solution to game in abstract class

Game in Abstracted Class

From black states, Player0 gets to choose the input method call.
From purple states, Player1 gets to choose a path in the abstract class till call returns.

Objective for Player0: Ensure error states (from which exception can be raised) are avoided.
Winning strategy: Correct method sequence calls.
Most General winning strategy: Most permissive safe interface.
Game is partial information!
Interface Synthesis

- Most permissive safe interface can be captured by a finite automaton (as a regular language over $M \times \mathbb{R}$)
  - For partial information games, the standard way (subset construction) to generate the interface is exponential in the number of states of abstract class
  - Number of states of abstract class is exponential in the number of predicates used for abstraction
  - Use of symbolic methods (e.g. OBDDs) desired

- Novel approach: Use algorithms for learning a regular language to learn interface
  - Angluin’s L* algorithm
  - Works well if we expect the final interface to have a small representation as a minimized DFA

L* Algorithm for Learning DFAs

Infers the structure of an unknown DFA by
- membership queries
- equivalence queries
Observation table $(S, E, T)$
\[
T: (S \cup S \cdot \Sigma) \cdot E \rightarrow \{0, 1\}
\]
Constructs a minimal DFA using a polynomial number of queries $O(|\Sigma|n^2 + n \log m)$ member at most $n-1$ equivalence

\[
S := \{\varepsilon\}; // states of DFA \quad E := \{\varepsilon\}; // distinguishing expts \quad \text{repeat:} \quad \text{Update } T; \quad // member tests for $(S \cup S \cdot \Sigma) \cdot E \quad \text{MakeTClosed}(S, E, T); \quad C := \text{MakeConjecture}(S, E, T); \quad \text{if } !(c=\text{IsEquiv}(C)) \text{ then return } C; \quad \text{else}\{ \quad e = \text{FindSuffix}(c); \quad \text{Add } e \text{ to } E; \quad \}
\]
Implementing $L^*$

- Transform abstract class into a model $M$ in NuSMV (a state-of-the-art BDD-based model checker)

- **Membership Query:** Is a string $s$ in the desired language?
  - Are all runs of $M$ on $s$ safe?
  - Construct an environment $E_s$ that invokes methods according to $s$, and check $M/E_s$ safe using NuSMV

- **Equivalence Query:** Is current conjecture interface $C$ equivalent to the final answer $I$? If not, return a string in the difference
  - Subset check: Is $C$ contained in $I$? Are all strings allowed by $C$ safe? Check if $C/M$ is safe using NuSMV
  - Superset check: Does $C$ contain $I$? Is $C$ most permissive?

---

**JIST: Java Interface Synthesis Tool**

Diagram:

- **Java** → **Java Byte Code** → **Soot** → **Jimple**
- **Interface Automaton** → **NuSMV Language** → **BJP2SMV** → **Boolean Jimple**
Signature Class

3 global variable predicates used for abstraction
24 boolean variables in abstract model
83 membership, 3 subset, 3 superset queries
time: 10 seconds
JIST synthesized the most permissive interface

package java.security;

public abstract class Signature extends java.security.SignatureSpi {
    //variable declarations
    protected int state = UNINITIALIZED;
    public final void initSignature(PublicKey publicKey) { ... }
    public final byte[] sign() throws SignatureException { ... }
    public final boolean verify(byte[] signature) throws SignatureException { ... }
    public final void update(byte b) throws SignatureException { ... }
    ...}

User Guided Synthesis
An Emerging Paradigm for System Design

1. Computer and programmer collaborate
2. Synthesizer discovers new artifacts
3. Computational power exploited for non-trivial programming tasks
Discussion Questions

1. Is synthesis really different than high-level programming? Isn’t the synthesizer just another compiler?

2. Synthesis is computationally hard, and researchers have tried it for decades. So what’s new?

3. Formal verification has seen some real-world success recently, but only in a limited form. Is that the best we can hope for?

4. This workshop is on "Software at Scale", can synthesis go beyond "toy" problems?

5. Are tools/techniques ready? Are they robust?

6. Do components have to be "correct"?