A Brief History of Synthesis
From Church to ExCAPE

Rajeev Alur
University of Pennsylvania
Programming Technology

Libraries → Compiler → Executable

Program

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

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

Platform
Verification Technology

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

Program Specifications Tests

Analysis Tool

Executable Platform
Challenges

- Software development still remains expensive and error-prone...

- What it means to “code” hasn’t changed...

- 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...

- Demand for new software development approaches
  - Receptive industry
  - Shifting goal of system design from performance to predictability
Synthesis: A Plausible Solution?

- Classical: Mapping a high-level (e.g. logical) specification to an executable implementation

- Recent shift in focus: Integrating different styles of specifications in a consistent executable

Goal of this lecture:

Brief introduction to different formalizations of synthesis problem
Talk Outline

- Reactive Synthesis
  - Program Refinement
  - Program Optimization
  - Syntax-Directed Program Completion
  - Programming by Examples
  - Expeditions in Computer Augmented Program Engineering
Advantages
Automated formal verification, Effective debugging tool

Industrial success
Domains: Multiprocessor coordination, Device drivers...
In-house groups: Intel, Microsoft, Motorola...
Commercial model checkers and consulting companies
Given a model of all components, does the product state-transition graph satisfy given requirements?

Requirements

- No reachable state has two caches in write-exclusive states (Safety/Reachability)
- Every read/write request is eventually completed (Liveness/Linear Temporal Logic)
- From every reachable state, there exists a path leading to a quiescent state (Branching-time/CTL)
Synthesis

**Given:**
- Model of Bus and Processors
- Requirements

**Synthesize** Cache controller so that requirements are met
- Cache controller model can be partially specified
Synthesis = Solving Games

- Model is viewed as a game graph
  - Different components viewed as separate players
  - Each move belongs to one of the players
  - Strategy for a player: choose the next move based on execution history

- Synthesizing Cache controller:
  - Uncontrolled choices: Moves in models of Bus and Processors
  - Requirements: Winning condition
  - Winning strategy: Deciding what Cache Controller should do
Safety Verification = Finding Invariant Set
Controller Synthesis for Safety
= Finding Controlled Invariant Set

Initial States

Controlled Invariant

Safety Set

Initial States
Reactive Synthesis

- **Historical Origin:** Church’s problem (1960s)
  
  Given Spec as a set of allowed sequences over \((I \times O)\)
  
  find \(\text{Imp} : I^* \rightarrow O\) such that \(\text{Imp}\) meets Spec

- **Original solution:** Rabin (72) based on automata over infinite trees

- **Renewed interest in late 1980s (Pnueli/Rosner)**
  
  Spec given by LTL, 2EXPTIME decision procedure

- **Theory of reactive synthesis**
  
  Connections between games and tree automata
  
  Decidability, complexity, expressiveness

- **Towards practical applications**
  
  Fragments with low complexity: GR(1)
  
  Robot motion planning
LTLMoP: Robot Control from High-level Specs

Kress-Gazit et al

Feasible specification

Visit all rooms

Unsynthesizable specification

specification text

proposition lists

log window
Talk Outline

- Program Refinement
  - Program Optimization
  - Syntax-Directed Program Completion
  - Programming by Examples
  - Expeditions in *Computer Augmented Program Engineering*
Program Verification

- Historical roots: Hoare logic for formalizing correctness of structured programs (late 1960s)

- Typical examples: sorting, graph algorithms

- Specification for sorting
  Permute(A,B): array B is a permutation of elements in array A
  Sorted(A): for 0 <= i < n, A[i] <= A[i+1]

- Function sort is correct if following holds
  \{ True \} B := sort(A) \{ Permute(A,B) & Sorted(B) \}

- Provides calculus for pre/post conditions of structured programs
Sample Proof: Bubble Sort

BubbleSort (A : array[1..n] of int) {
  B = A : array[1..n] of int;
  for (i=0; i<n; i++) {
    Permute(A,B)
    Sorted(B[n-i,n])
    for 0<k<=n-i-1 and n-i<=k’<=n B[k]<=B[k’]
    for (j=0; j<n-i; j++) {
      Permute(A,B), Sorted(B[n-i,n],
      for 0<k<=n-i-1 and n-i<=k’<=n B[k]<=B[k’]
      for 0<k<j B[k] <= B[j]
      if (B[j]>B[j+1]) swap(B,j,j+1)
    }
  }
  return B;
}
Program Verification: State-of-the-art

- If the programmer proposes “candidate assertions” for loop-invariants, possible to check automatically if indeed that is the case
  - SMT solvers (Satisfiability Modulo Theories)
  - Booleans, Integers, Bit-vectors, Arrays, Uninterpreted functions

- Powerful mathematical logics (e.g. first-order logic, Higher-order logics, separation logic) used for formalization

- Contemporary theorem provers: HOL, PVS, ACL2, Coq, Boogie
  - Decision procedures for fragments of various logics
  - Tactics for decomposition of proofs

- Main applications: Microprocessor verification, Correctness of JVM
  - New trend: Clean-slate design (fully verified systems)
From Verification to Refinement

- Specification is program $P_0$ with “high-level” constructs

- Build a sequence of programs $P_1, P_2, \ldots P_n$ such that
  - Each $P_{i+1}$ obtained from $P_i$ by adding some more detail
  - $P_{i+1}$ “refines” $P_i$ in a formal way
  - $P_n$ is desired implementation

- Similar concept of “correct-by-construction” prominent in model-based-design for systems
To show Imp refines Spec,
User provides mapping $f$ from Imp states to Spec states
Tool checks if every step in Imp is allowed in Spec

Diagram:
- Imp-State $\rightarrow f \rightarrow$ Spec-State
- Step in Imp $\rightarrow f \rightarrow$ Possible step in Spec
- Imp-State' $\rightarrow f \rightarrow$ Spec-State'
Compositional Refinement

- When adding detail to one part of the program, local reasoning about the change should suffice:

Formally, refinement relation should be a congruence
Refinement-based Synthesis in Practice

- Requires great expertise, but process is similar to developing a proof
  - Automation of proving one-step-refinement via decision procedures
  - Guidance for what to refine by libraries and tactics

- Specware (Kestrel)
  - Concurrent garbage collector
  - Efficient SAT solver

- Well-developed refinement-based program development: B method
Talk Outline

Program Optimization

- Syntax-Directed Program Completion
- Programming by Examples
- Expeditions in Computer Augmented Program Engineering
Modern Compilation is a form of Synthesis

- Mapping C code to x86 assembly code involves many transformations and optimizations
  - Each transformation is like a refinement step
  - Fully automated
  - Guaranteed to preserve semantics

- In Electronic Design Automation (EDA), synthesis means from hardware description language (Verilog, VHDL, SystemC) to gate-level description of circuit (Netlist)
  - Many transformations and optimizations for area, power, timing
  - Industrial impact of academic research (e.g. Synopsis)
Superoptimizing Compiler

- Given a program $P$, find a “better” equivalent program $P'$

```cpp
multiply (x[1,n], y[1,n]) {
    x1 = x[1,n/2];
    x2 = x[n/2+1, n];
    y1 = y[1, n/2];
    y2 = y[n/2+1, n];
    a = x1 * y1;
    b = shift( x1 * y2, n/2);
    c = shift( x2 * y1, n/2);
    d = shift( x2 * y2, n);
    return ( a + b + c + d)
}
```

Replace with equivalent code with only 3 multiplications
Synthesis-enabled Compilers

- Highly effective in high-performance computing

- Success story: Spiral (CMU)
  - Focus on rewriting of matrix operations
  - Can generate highly optimized platform-specific code for FFTs
  - Beats human programmers and saves effort

- Under the hood:
  - Transformations built in
  - Desired sequence of transformations obtained by heuristic search
  - Key to success: domain specific
Paraglide: From Sequential to Parallel Code

Ref: Vechev et al (POPL 2010)

Sequential Program

```c
bool add(int key) {
    atomic
    Entry *pred,*curr,*entry
    locate(pred,curr,key);
    k = (curr->key == key)
    if (k) return false
    entry = new Entry()
    entry->next = curr
    pred->next = entry
    return true
}
```

Minimal Synchronization

```c
bool add(int key) { 
    Entry *pred,*curr,*entry
    restart:
    locate(pred,curr,key)
    k = (curr->key == key)
    if (k) return false
    entry = new Entry()
    entry->next = curr
    val= CAS(&pred->next,<curr,0>,<entry,0>)
    if (!val) goto restart
    return true
}
```

Architecture Description

- Target: Highly concurrent work queue in C/C++
- Infers minimal number of fences needed for synchronization
- Unexpected, correct, minimal solutions now deployed in IBM

Enables programmers to meet new programming challenges
Talk Outline

- Syntax-Directed Program Completion
  - Programming by Examples
  - Expeditions in Computer Augmented Program Engineering
Program Verification

SelectionSort(int A[], n) {
    i1 := 0;
    while (i1 < n-1) {
        v1 := i1;
        i2 := i1 + 1;
        while (i2 < n) {
            if (A[i2] < A[v1])
                v1 := i2;
            i2++;
        }
        swap(A[i1], A[v1]);
        i1++;
    }
    return A;
}

post: \forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k + 1] \ (\text{swap ensures perm.})
Template-based Automatic Invariant Generation

SelectionSort(int A[], n) {
    i1 := 0;
    while (i1 < n - 1) {
        v1 := i1;
        i2 := i1 + 1;
        while (i2 < n) {
            if (A[i2] < A[v1])
                v1 := i2;
            i2++;
        }
        swap(A[i1], A[v1]);
        i1++;
    }
    return A;
}

post: \forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k + 1]
SelectionSort(int A[], n) {
    i1 := 0;
    while (i1 < n-1) {
        v1 := i1;
        i2 := i1 + 1;
        while (i2 < n) {
            if (A[i2] < A[v1])
                v1 := i2;
            i2++;
        }
        swap(A[i1], A[v1]);
        i1++;
    }
    return A;
}

post: \( \forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k + 1] \)

Invariant:
\( \forall k_1, k_2. \ 0 \leq k_1 < k_2 < n \land k_1 < i_1 \Rightarrow A[k_1] \leq A[k_2] \)

Invariant:
\( i_1 < i_2 \land i_1 \leq v_1 < n \land (\forall k_1, k_2. \ 0 \leq k_1 < k_2 < n \land k_1 < i_1 \Rightarrow A[k_1] \leq A[k_2]) \land (\forall k. \ i_1 \leq k < i_2 \land k \geq 0 \Rightarrow A[v_1] \leq A[k]) \)
SelectionSort(int A[], n) {
  i1 := 0;
  while (i1 < n - 1) {
    v1 := i1;
    i2 := i1 + 1;
    while (i2 < n) {
      if (A[i2] < A[v1])
        v1 := i2;
      i2++;
    }
    swap(A[i1], A[v1]);
    i1++;
  }
  return A;
}

post: \forall k: 0 \leq k < n \Rightarrow A[k] \leq A[k + 1]
Template-based Program Synthesis

Bodik, Solar-Lezama 2005

SelectionSort(int A[], n) {
    ?;
    while(?) {
        ?;
        while (?) {
            ?;
        }
        ?;
    }
    return A;
}

Invariant:
∀k1, k2. ? ∧ ?

Invariant:
? ∧ ? ∧ 
(∀k1, k2. ? ∧ ?) ∧ (∀k. ? ∧ ?)

post: ∀k : 0 ≤ k < n ⇒ A[k] ≤ A[k + 1]
Template-based Program Synthesis

SelectionSort(int A[], n) {
    i1 := 0;
    while (i1 < n - 1) {
        v1 := i1;
        i2 := i1 + 1;
        while (i2 < n) {
            if (A[i2] < A[v1])
                v1 := i2;
            i2++;
        }
        swap(A[i1], A[v1]);
        i1++;
    }
    return A;
}

post: \( \forall k : 0 \leq k < n \Rightarrow A[k] \leq A[k + 1] \)

Invariant:
\( \forall k_1, k_2. \ 0 \leq k_1 < k_2 < n \ \land \ k_1 < i_1 \Rightarrow A[k_1] \leq A[k_2] \)

\( i_1 < i_2 \ \land \ i_1 \leq v_1 < n \ \land \ (\forall k_1, k_2. \ 0 \leq k_1 < k_2 < n \ \land \ k_1 < i_1 \Rightarrow A[k_1] \leq A[k_2]) \ \land \ (\forall k. \ i_1 \leq k < i_2 \ \land \ k \geq 0 \Rightarrow A[v_1] \leq A[k]) \)

Synthesized program and proof of correctness
Syntax-Directed Program Synthesis

- Given (1) partial program $P$ with some parts left as holes
  (2) template to limit search space of how to fill holes
  (3) specification in form of assertions / pre-post conditions
  synthesize the program completion

- CEGIS strategy (Counter-example guided inductive synthesis)
  1. Find candidate expressions to fill holes
  2. Check if resulting completion meets specification (SMT solver)
  3. If not, repeat step 1 based on counter-examples generated in 2

- Sketch (Bodik, Solar-Lezama et al) and its variants
  - Efficient code for bit-vector operations
  - Grading of programming assignments
  - Active ongoing research to improve scalability, usability
Parallel Parking by Sketching

Err = 0.0;
for(t = 0; t<T; t+=dT){
  if(stage==STRAIGHT){
    if(t > ??) stage= INTURN;
  }
  if(stage==INTURN){
    car.ang = car.ang - ??;
    if(t > ??) stage= OUTTURN;
  }
  if(stage==OUTTURN){
    car.ang = car.ang + ??;
    if(t > ??) break;
  }
  simulate_car(car);
  Err += check_collision(car);
}
Err += check_destination(car);

When to start turning?
Backup straight
How much to turn?
Straighten

Enables programmers to focus on high-level solution strategy
Talk Outline

- Programming by Examples
- Expeditions in Computer Augmented Program Engineering
Can one “program” by drawing all “interesting” scenarios?
Scenario-based Programming

- Historical roots: Features of telecommunication software
  What sequence of events should occur if you pick your phone, press *, then a digit, and then “Dial” key?

- IEEE Standardized format for scenarios: MSC'96

- MSCs supported by UML and other software engineering frameworks

- Initial work: Tools for managing large repositories of MSCs and detecting conflicts (e.g. uBET)

- Live Sequence Charts (Damm and Harel)
  Extension of MSCs with many features (may/must scenarios...)
  Let’s Play toolkit for programming reactive controllers
FlashFill: Programming by Examples Success Story

Ref: Gulwani (POPL 2011)

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(425)-706-7709</td>
<td>425-706-7709</td>
</tr>
<tr>
<td>510.220.5586</td>
<td>510-220-5586</td>
</tr>
<tr>
<td>1 425 235 7654</td>
<td>425-235-7654</td>
</tr>
<tr>
<td>425 745-8139</td>
<td>425-745-8139</td>
</tr>
</tbody>
</table>

- Infers desired Excel macro program
- Iterative: user gives examples and corrections
- Being incorporated in next version of Microsoft Excel

Enables non-programmers to program interactively
From Examples to Programs

- Formally, a learning problem: given a set of positive and negative example learn a program that is syntactically correct and consistent with all the examples

- Computational techniques
  - Heuristics to efficiently search program space
  - Domain-specific techniques for scalability
  - Relies on ranking of potential solutions
Expeditions in Computer Augmented Program Engineering

http://excape.cis.upenn.edu/

Cornell, Maryland, Michigan, MIT, Penn, Rice, UC Berkeley, UCLA, UIUC

Annual Meeting, June 2013
ExCAPE Vision

Harnessing computation to transform programming:
Programming made easier, faster, cheaper

Diagram:
- Designers
- Synthesis Tool
- Software
- Partial Source Code
- Scenarios
- Partial Specifications
- Libraries
- Platform Constraints
- Performance Metrics
Synthesis Tool: Intelligent Assistance

- Designer expresses “what”, possibly using multiple input formats
- Synthesizer discovers new artifacts via integration and completion
- Synthesizer solves computationally demanding problems using advanced analysis tools
- Interactive iterative design
- Integrated formal verification
Research Organization

Tools and Evaluation

Design Methodology

Computational Engines

Challenge Problems

- Apps for Mobile Platforms
- Multicore Protocols
- Networked Systems
- Robotic Systems

Education and Knowledge Transfer
Goal: Simplify Protocol Design

- Design challenging due to asynchronous model of communication
- Cache coherence protocols, Distributed coordination algorithms
- Successful application domain for formal verification / model checking
- Correctness involves both safety and liveness properties
- Proposed solution: Allow programmers flexibility

\[ \text{Protocol} = \text{Skeleton based on Extended-Finite-State-Machines} \]
\[ + \text{High-level requirements} \]
\[ + \text{Example behaviors} \]
TRANSIT for Distributed Protocol Design
Computational Problem

- **Inputs:**
  - Variable types and corresponding expression grammar
  - For each process,
    1. Control states of EFSM
    2. List of all variables, input/output messages
    3. Set of concrete examples + symbolic constraints
  - High-level requirements (invariants and temporal logic formulas)

- **Solution strategy:**
  - **Expression Inference:** For each EFSM transition, generate expressions for guards and updates. Solution uses Counter-example-guided-inductive synthesis using SMT solver Z3
  - Check if resulting protocol meets all requirements, using a model checker (Murphi) and if not, report a counter-example
Challenging Case Study: SGI Origin protocol

- Source: Laudon and Lenoski; The SGI Origin: A CCNUMA highly scalable server; ISCA 1997

- Directory-based MESI protocol that handles multiple concurrent requests to same requests over unordered network

- Textual description directly leads to protocol skeleton, and symbolic (incomplete) descriptions of most of the transitions

- During debugging, programmer focuses on local fixes of counterexamples and adds concrete examples

- Final iteration required 30 min synthesis time (with 5 Million states explored by Murphi)

- SMT solver / model checker in the loop is feasible for programming
Paradigm shift in synthesis:
Old: Allow more concise, high-level description
New: Designer uses multiple, natural formats,
      Synthesis tool assists in discovering tricky logic

Paradigm shift in design tools:
Old: Any compiler transformation must be polynomial-time
New: Computational intractability not a show-stopper

Common theme: Guided search in a space of programs to find one that
        meets multiple design goals
        A bit like model checking, but can be interactive!