Automated Synthesis and Design Error Repair of Systems

Georg Hofferek
IAIK – Graz University of Technology
greg.hofferek@iaik.tugraz.at

(some) slides courtesy of Roderick Bloem
DIAMOND Project
Design Timeline

Specify  Implement  Detect  Localize  Correct

Debugging

Time
Same Thing Twice?

- Specification
- Implementation
- Requirements
- Verification

Synthesis
Synthesis == Repair
Overview

- **Property Synthesis**
  - Properties $\rightarrow$ System
  - Formal Specification
    - Temporal Logic
    - Assertions
    - Automata
  - 2-Player Game

- **Abstraction with Uninterpreted Functions**
  - Applicable to more complex systems
  - Mixed Imperative/Declarative Paradigm
Reactive Systems

- Infinite Sequence of Inputs
- Infinite Sequence of Output

- Constant interaction, No Termination

- Examples:
  - Bus Arbiter
  - Lift Controller
  - Traffic Lights
Properties

- **Property**: subset of input/output traces
  - “Traffic lights show green infinitely often”
  - “Signals ack1 and ack2 never both high”
  - “Button pushed \(\rightarrow\) Lift will eventually arrive”

- Different ways of formalization
  - Temporal Logic
  - Assertions
  - Büchi Automata

- “What to do” vs. “How to do it”
Synthesis is a Game

- Two Players:
  - Environment provides inputs
  - System provides outputs

- Specification fulfilled → System wins
Example I: Chess

- **Environment** determines black moves
- **System** determines white moves
- Winning condition:
  - Black moves legal $\rightarrow$ white moves legal
  - White reaches checkmate

Easy to specify!
Computing Games: Reachability

Label on edges:
- Environment input
- System output

Dash (–) means don’t care

Find all states from which system can force visit to goal state + a strategy

Winning region
Symbolic Computation

- **States**: sets of atomic propositions

- **(Sets of) States**: represented by formulas
  \[ x_1 \land \neg x_2 \]

- **Transition relations**
  \[ x_1 \land \neg x_2 \land x_1' \]

- Formulas can be manipulated efficiently
  - Binary Decision Diagrams (BDDs)
More General Games

- **Reachability:**
  
  \[ \text{Eventually } p \]

- **Büchi:**
  
  \[ \text{Always Eventually } p \]

- **Generalized Büchi:**
  
  \[ \text{Always Eventually } p_0 \]
  
  \[ \text{and} \]
  
  \[ \text{Always Eventually } p_1 \]
Elevator

- Button pushed → Elevator eventually arrives
- No movement if door is open
- Environment Assumptions!
More General Games

- **Reactivity(1):**
  Always eventually $e$ implies
  Always eventually $p$

- **Gen. Reactivity(1):**
Synthesis

Specify
- Temporal Logic
- Assertions
- Automata

Create Game

Solve Game
- Strategy

Create System
- Determinize Strategy
Strategy

- **Look-up Table:**
  Current state $\rightarrow$ *conforming moves*
# Freedom in Strategies

<table>
<thead>
<tr>
<th>Input ((i_1, i_2))</th>
<th>Output ((o_1o_2o_3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 1 0</td>
</tr>
<tr>
<td>0 1</td>
<td>1 0 –</td>
</tr>
<tr>
<td>1 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td></td>
<td>0 0 1</td>
</tr>
<tr>
<td>1 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>0 1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1 0</td>
<td>–</td>
</tr>
</tbody>
</table>

- **Fixed Output, No Freedom**
- **“Don’t Care”:** \(1 0 –\) = \(1 0 0, 1 0 1\)
- **Multiple Vertices, Not Expressible with Don’t Cares.**
### Compatible Function

<table>
<thead>
<tr>
<th>Input ($i_1i_2$)</th>
<th>Output ($o_1o_2o_3$)</th>
<th>Compatible Function (example)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 1 0</td>
<td>0 1 0</td>
</tr>
<tr>
<td>0 1</td>
<td>1 0 –</td>
<td>1 0 0</td>
</tr>
<tr>
<td>1 0</td>
<td>1 1 0, 0 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1 1</td>
<td>1 0 0, 0 1 1, 1 1 –</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>
Resubstitution

<table>
<thead>
<tr>
<th>Input ($i_1i_2$)</th>
<th>Output ($o_1o_2o_3$)</th>
<th>Compatible Function (example)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0 0</td>
<td>1 . . .</td>
</tr>
<tr>
<td></td>
<td>0 0 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 1 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 1 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 1 1</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>1 0 0</td>
<td>1 . . .</td>
</tr>
<tr>
<td></td>
<td>1 0 1</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>0 0 1</td>
<td>1 . . .</td>
</tr>
<tr>
<td></td>
<td>1 0 1</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>1 0 0</td>
<td>1 . . .</td>
</tr>
</tbody>
</table>

Loss of freedom for $o_2$ and $o_3$
Case Study: AMBA Bus Arbiter

- Industrial standard
- ARM’s AMBA AHB bus
  - High performance on-chip bus
  - Data, address, and control signals (pipelined)
  - Arbiter part of bus (determines control signals)
  - Up to 16 masters and 16 clients
AMBA Arbiter

- Specification
  - 3 Assumptions, 12 Guarantees.

- Example:

  “When a locked unspecified length burst starts, new access does not start until current master (i) releases bus by lowering HBUSREQi.”

\[ \bigwedge_i G( HMASTLOCK \land HBURST=INCR \land HMASTER=i \land START \rightarrow X(\neg START \lor \neg HBUSREQ_i) ) \]
New Spec

Circuit size (std cell grid count)

- KS
- cofactors
- new spec
- manual

More recent results go up to 16 masters
Summary: Property Synthesis

- It works!
  - Properties $\rightarrow$ Systems
  - Real-World Examples
  - Efficiency

But:
- Debugging Specs
- Robustness
- Quantitative Aspects
- Data-oriented Circuits
Imperative vs. Declarative

- **Imperative Paradigm**
  - *How* to do something

- **Declarative Paradigm**
  - *What* to do
A Processor

Tough:
- 64-bit datapath
- very complex arithmetic logic unit

How do I pipeline that?
A Pipelined Processor

That’s trivial!
A Pipelined Processor

Instructions:
- r1 := mem[1]
- r2 := r1 + r2

r1 = 15
r2 = 27

15

stall

forward
A Pipelined Processor

Not so trivial!

stall

forward
Specification

Basic elements are the same → use uninterpreted functions
Abstraction by Uninterpreted Functions

- Datapath
  - Bit-wise description prohibitively large
    - e.g. ALU

- Abstraction by uninterpreted functions
  - neither know nor care about “internals”

- Functional Consistency
  - \( a = b \rightarrow f(a) = f(b) \)
Sufficient Condition: Commutative Diagram

Pipelined and non-pipelined processor give same result for any instruction sequence
ψ = 
\[(\text{mem}', \text{reg}') = \text{flush} \cdot \text{non-pipe-instr (mem, reg)} \land \]
\[(\text{mem}'', \text{reg}'') = \text{pipe-instr} \cdot \text{flush (mem, reg)} \]

Part of ψ:
\[\text{res\_ex} = \text{ALU(opc\_de, arg1\_de, arg2\_de)}\]

Pipeline correct iff ψ valid
ψ written in logic with uninterpreted functions, arrays, and equality
From Verification to Synthesis

- **stall, forward** are **Boolean** control signals
- **Mixed Quantifiers**

\[ \varphi = \forall \text{mem, reg, pl. state } \exists \text{stall, forward } \forall \text{mem', reg', mem'', reg'’. } \psi \]

1. Solve \( \varphi \)
2. Get **certificate**: (stall, forward) = \( f(\text{mem, reg, pl. state}) \) that makes \( \varphi \) true
Synthesis Procedures

- Reduction to Propositional Logic
  - Following standard algorithms
  - Solve e.g. using Binary Decision Diagrams
  - Inefficient!

- Interpolate over Uninterpreted Functions
  - Work in Progress
  - Promising first results
Summary: Abstraction with Uninterpreted Functions

- Datapath Abstraction
  - Makes interesting problems tractable

- Mixed Paradigm
  - Specify control
  - Implement datapath
Concluding Remarks

- Synthesis is practical
  - Prototyping
  - Safety/Security Critical Systems

- Not all problems solved (yet)
  - Efficiency
  - Formal specifications
  - (Prototype) Tool Support

- Mixed Approaches
  - Very Promising
Further Information