Design Methodology
Overview

Rastislav Bodík  UC Berkeley
Goals of design methodology

**Integrate** synthesis in **design** and **development:**

programmer-tool dialogue
  - programmer conveys problem-specific **insights**
  - interaction model to refine her **understanding**

language/tool support for synthesizer construction
  - including specs and user queries, in various abstractions

algorithms to combine computational engines
  - and to generate code
Outcome

Collaborative distillation of methods and practices
share success stories of industrial, education impact

Disseminate widely tested approaches
internally, then externally
1. Extend the language

Solver-aided Domain-Specific Languages

Modern systems are structured with **DSL**

* regular expressions, grammars, SQL, linear algebra, automata-based controllers, Datalog, statistical models, jQuery, Xpath, specification languages eg Spec#, network sockets, MapReduce, ...

**Solver-aided** DSLs: constructs for verif, synth, fault loc

– Automata tutor (Rishabh’s talk in Online Education)
– Python autograder (Loris’ talk in Online Education)
– Chlorophyll (Mangpo’s talk in Distributed Protocols)
2. Don’t reinvent the wheel

Tools for construction of solver-aided DSLs,

Exploit well understood (and popular) embedding of DSLs into host languages

- **Fan**  
  statically-typed host (Zdancewic, Penn),
- **Rosette**  
  dynamically-typed host (Torlak, Berkeley),
- **Sketch**  
  symmetry reduction (Solar-Lezama, MIT)
3. Just say it

Multi-modal specifications

Allow expressing requirements and hints naturally:

– safety assertions
– reference implementations
– temporal logic
– demonstrations: traces of execution and i/o pairs
– resource constraints
4. Don’t assume knowledge of spec language

Frameworks contain ~10k classes. Their names form a spec language. But programmers don’t know them.

**CodeHint** queries the concrete program state, allowing queries such as:

- synthesis code that produces an object with a field whose value is a string that contains the string “Clipboard”
5. Expect the unexpected

**Robustness**: function correctly on invalid inputs

Robotic control (Tabuada, Kress-Gazit):
- bounded disturbances have bounded consequences
- effect of sporadic disturbances disappears over time

Autograding for a CPS lab (Seshia)
- trace simulated student robots; then perturb the trace
- see Garvit tomorrow in Online Education
6. Embrace ambiguity

Specifications may be under-constrained.

Weighted SAT sampling

– provides guarantees on converging to optimal solution
  – Seshia (Berkeley), Vardi (Rice)

Enumerate spaces of alternative models

– analyze alternative biological models
  – Bodik, Fisher, Piterman (Berkeley, Microsoft, Leicester)
7. Interact

Programmer-synthesizer dialogue

A debugging scenario

1. **find a failing input** by attempting to verify
2. **localize the fault** via UNSAT core
3. **repair the run** via nondeterministic execution
4. **repair the program** via synthesis with syntactic insight

It’s similar in automated grading of programs
uses a fixed error model

Automata tutor

Loris’ talk tomorrow
8. Invent a calculus

Search for the desired program in a reduced space
  – free of symmetries and other redundancies

Synthesis of relational queries
  – from source code     Solar-Lezama (MIT)
  – from user demonstrations  Bodik (MIT)

Superoptimization
  – Polish notation: variable-free, and hence symmetry-free

Decomposition is crucial to reach practical scale.

Many approaches:
- automatic synthesis of abstractions in robotics
- a synthesis-aided compiler
  - instead of analysis/transformation, solves several solvers

Three talks in this session:
- models of libraries Solar-Lezama
- checking refinement Sangiovanni-Vicentelli, Tripakis
- views on a system Tripakis
10. Talk to me

Let me know if you have a design method to share

Volunteer to coordinate distillation of principles