ExCAPE Kick-off Meeting: Computational Engines Scribe Notes  
Jean Yang, 5.4.12

**Reactive Synthesis** (Moshe Y. Vardi, Rice University)
- Synthesis = automated design
  - Start from specification $\psi$, design $P$ such that $P \models \psi$
  - Advantage: no verification; no redesign
  - In essence, declarative programming taken to the limit
- Synthesis of ongoing programs
  - In case of temporal formulas, work goes back to 1980s
  - Synthesis: extracting formulas from models
  - One big counter
- Reactive synthesis
  - By mid-80s people realized common ideas: interaction with environment
  - Reactivity (Harel & Pnueli, 1985)
  - Example: printer specification (safety; liveness)
- Satisfiability and synthesis
  - Continuation of printer example
  - Issue: did not make distinction between environment actions (submit job) and printer actions (print the job)
  - Need to make distinction between input and output--satisfiability is inadequate for synthesis
- Realizability: distinction between input and output
  - Some point about linear temporal logic?
  - Several people concluded we need game-theoretic view of synthesis
  - Pnueli & Rosner, 1989
    - Realizability - existence of winning strategy for specification
    - Synthesis - constructing winning strategy
- Strategy Trees
  - Connection to $\omega$ automata in 1970s
  - Infinite tree; labeled infinite tree
  - Rabin's insight: strategy is labeled tree with directions $D = 2^I$ and alphabet $\Sigma = 2^O$
  - Winning: every branch satisfies specification
- Rabin 1972's Realizability Algorithm

**Post-1972 Developments**
- LTL as spect language
- Exponential translation from LTL to automata
- Doubly exponential construction of tree automata
- ...

**Standard Critique**
- Impractical! 2EXPTIME
- In many cases, performance is okay
- Programmers do not tend to write pathological specifications

**Real critique**
- For $\omega$, took 40 years to develop algorithm ready for industrial
In practice, theory and practice very different—need to think about putting user in the center when it comes to building specifications.

More research needed.

Sometimes things take longer than 3-5 years.

Want to discuss not how to solve whole problem, but how things fit into bigger ecosystem.

**Synthesis techniques from discrete event systems (Stephane Lafortune, UMichigan)**

- From control engineering community
- Many in control community work with differential equations
- Two popular techniques: finite-state automata and petri nets
- Petri nets - instead of explicitly modeling transitions, use mechanism of transition nodes and place nodes; put tokens and transition functions describe movement w/ tokens. More compact representations.
- Safety: specification automaton; constraints on tokens in petri net. State is set of tokens in each place.
- Liveness: non-blocking properties (not allow deadlocks/livelocks)
- Optimality: view safety/liveness in maximally permissive manner
- Cannot achieve all of it because of limited actuation, sensing capabilities
  - Subset of controllable events representing actuators
  - Sensors may not capture all possible events in system
- Synthesizing feedback controller given limited capabilities in maximally permissive manner
- When all events are observable & have to deal w/ uncontrollable events: notion of optimally (supremal controllable operation)
- More complicated w/ unobservable events
- If all controllable events are observable, then have supremal controllable normal operation
- Tons of research on modular decomposition (horizontal/vertical)
- Challenge is scalability: scalability w/ automata has been focused on exploiting system structure
- System has interacting components—do controller synthesis using inherent decomposition of system as well as specifications
- Scalability with petri nets: supervision based on place invariants—enforce linear inequalities on state vector
  - Deadlock/liveness in Gadara Petri nets can be mapped to presence of certain types of siphons and map this to linear inequalities
- In recent work, use SAT solvers to encode Petri net structures + specifications of not reaching bad siphons—achieved good scalability
- Dining philosophers fall in class of Gadara petri nets
- Conclusion: in control of discrete-event systems, have a lot of theoretical results on structural properties of systems and characterization of advisors. Can scale by exploiting this structure for efficient representation.

**Synthesizing robust software (Paulo Tabuada, UCLA)**

- Using ideas from control theory to synthesize software
- Why robustness? Software system designed based on environment assumptions
- Real environment either unknown or changing—assumptions
necessarily violated

- Still want some assurance about behavior: would ideally like modest
deviation from assumptions to yield modest deviation from
guarantees—**robustness**!

- **Control theory**
  - Models are essential but always wrong (car controller: weight of
car, aerodynamic characteristics, etc.)
  - Most basic designs do not design for specific deviations but are
robust against all deviations

- **Recent work in software robustness in CS**
- **Control theory**: robust control
- **Classification**: state-based vs. I/O-based
- **State based**: finite-state automaton extended as metric automata,
  notion of behaviors being “close”
  - Special symbol denoting environment not doing anything
  - Parameter measuring power of disturbance
  - Transitions can take us to balls centered around nominal states
  - Results parameterized on \( \beta \) (environment)—don’t need to
  know it a priori
  - Definitions of winning under no disturbance; derive under
  disturbance \( \gamma \)

- **Questions we can ask**
  - Verification: Is the strategy \( \gamma \)-robust?
  - Synthesis: Can we synthesize a \( \gamma \)-robust strategy?
  - All problems polynomially solvable (dynamic programming)

- **I/O-based robustness**
  - Instead of looking at automata, can look at transducers—no
  longer need to worry about how functions are implemented
  - Put costs on input/output strings
  - Properties of robustness: bounded disturbance should lead to
  bounded consequences; effect of sporadic disturbance should
  disappear after finite steps

- **Input-to-State Dynamic Stability**

- **In principle, should be possible to robustify any synthesis methodology**
  - How to bring these ideas closer to the programmer? Specifying
    robustness requires programmer to rank possible env behaviors
    and possible program behaviors

---

### Programming with Constraint Solvers (Ras Bodik & Emina Torlak, UC Berkeley)

- Code checking, angelic execution, debugging, synthesis, & beyond
- Constraint solver allows programmer to ask questions about program
during programming—pose these as SAT/SMT formulas
  - Code checking - is there valid input which violates
    post-condition? If satisfiable, get model w/ assignment that can
    be lifted to concrete counterexample
  - Debugging - w/ desired concrete input/output, can figure out
    why output is not produced using maxsat/mincore queries;
    system can find repair candidates
  - Angelic execution - find model that satisfies spec, generate
    trace that can be generalized to find actual expression
  - Synthesis - ask solver to find model; tool finds expression

- **Fundamental challenges**
  - Decidability, tractability, scalability
    - Do analysis in finite universe
Stay within more restricted logics--mostly decidable
- Building these tools is hard--need to become expert;
  - fragmented infrastructure
- Using these tools is hard
  - How to keep system in the loop? Want user to express high-level knowledge to pass to solver.
  - How to keep user in loop? Most of the time, no feedback if system cannot find a solution--or feedback not at sufficiently high level to be usable by non-expert.
- Goal: create infrastructure for programming with solvers
- Tool: Rosette for code checking, angelic execution, and synthesis for high-performance computing
  - If need to prototype a synthesizer for programming model/DSL:
    - write an interpreter to define language, define what "holes" look like--Rosette will do the rest
  - For more scalable synthesizer, Rosette will help you develop an efficient translator to formulas

**Inductive learning with SMT solvers and beyond** (Armando Solar-Lezama, MIT)
- Doing synthesis from examples--has been around for a while, for instance in machine learning community
- One of first questions: what is an example?
  - What happens when input/output traces can be infinite?
  - What is an example for thread interleaving?
- How much detail? Partial vs. complete, positive vs. negative, noisy vs. sanitized.
- Holistic value: interaction with the user. Active vs. passive learning: should the synthesis algorithm participate in how the examples are selected?
- Basic strategy
  - Choose a parameterized family of possible solutions--language-based mechanisms for defining spaces of solutions
  - Search space of parameters for good instances
- Two questions
  - How to represent the search space?
  - How to eliminate more than one candidate at a time?
- Inductive synthesis with SAT/SMT--approaches lend themselves well to programming by examples. Once you have parameterized family of programs, and given set of I/O pairs, can define constraints on I/O pairs.
  - Candidate input/output pairs eliminate whole spaces of programs that don't satisfy constraints. Once have good set of examples, can pick something out of satisfying programs.
  - Want to find other examples to help with learning. When using solver to come up with function, don't have a good metric for "close to boundary." Can use predicate and vote--closest to boundary means maximum disagreement.
- Synthesis with abstract examples: ex. data structure manipulations
  - More powerful--infinite set of examples
  - Less powerful--less detail
  - Challenge: scenario is quantified constraint
- Beyond user-provided examples
  - Examples can come from sample code
  - Programming by imitation
Combining induction and deduction for synthesis (Sanjit Seshia, UC Berkeley)

- Common ingredient on research: combination of inductive and deductive reasoning
- Induction: specific examples $\rightarrow$ general rules
- Deduction: general rules $\rightarrow$ specific conclusions (logical inference; constraint-solving)
- Three synthesis stories--purely deductive approach is hard
  - Reverse engineering malware--code from Conficker Worm
    - Obfuscated code for $y = y \times 45$
    - Figured this out by treating obfuscated code as specification & deriving simpler equivalent program
    - Challenge: specification controlled by attacker
  - Synthesizing switching logic for hybrid systems w/ nonlinear dynamics (nonlinear differential equations or nonlinear equations in solved form)
    - Challenge: undecidability! Need way that makes the approach work in practice.
  - Reactive synthesis from LTL. Two pairs: environment assumptions and system requirements. Often tool comes back with "unrealizable;" have to figure out what went wrong. Often due to missing environment assumptions--this is hard!
- Common approach: sciduction (structure-constrained induction and deduction). Inductive reasoning (active learning--generalizing from examples) + deductive reasoning (*lightweight logical inference & constraint solving) + structure hypotheses (on artifacts to be synthesized)
- Demonstration
  - Reverse engineering malware--make assumption about kind of program (loop-free compositions of primitive components) + learning from distinguishing I/O examples + SMT solving (bit-vector arithmetic)
  - Switching logic: guards are hyperboxes + hyperbox learning from examples + numerical simulation (constraint solving)
  - Generating environment assumptions from LTL: environment assumptions are restricted GR(1) + version space learning (from counterstrategies) + (finite-state) model checking
- Inductive strategy at the top-level--structure hypothesis defines concept (program) class
  - Inductive engines makes queries to deductive engine (give examples of certain nature to prune space)
  - Deductive engine generates examples for inductive engine
  - Deductive engine generates labels for examples
  - Deductive engine serves as verifier: verification or counter-example
- Deductive strategy at top-level: structure hypothesis defines logic & domain theory
  - Ded eng provides background facts
  - Ind eng provides conjectured lemmas
- Hard parts of verification problems often involve synthesis (invariants, environment assumptions)
- Methodology - if system doesn’t come up with answer, programmer must change design in some way and tool must work with different hypothesis. If we want interactive exploration tool, want portfolio of engines under hood. Want environment to allow user to formulate &
check multiple hypotheses.

- Future directions
  - Portfolio of computational engines
  - Choice of engine determined by problem definition, structure hypotheses, application domain

Questions:
- Mark to Sanjit: Could you take incomprehensible code and describe it more easily? Can you convert COBOL programs to Ilog (?).
  - Answer: In practice, depends on the engines. What is the class of problems you want to convert from COBOL to Ilog? A lot of code they looked at was doing fairly simple transformations.
- Moshe: guy started a company to define simplified manipulations on straight line programs; find way to simplify them.
- Armando: we are helping people translate Java code into simpler SQL queries.
- Moshe: need to understand users better.
- Rajeev: want to focus on allowing people to express different things differently. There will be some parts of what you want to express that is better done with one abstraction, and others with another abstraction.
- Sanjit: in Excel synthesis, ultimate goal is to have some interface where people can go in and look at outliers. Both in macro case and this case, goal is to remove the tedious task.
- Ras: good idea to have multi-modal specification--write in the way you best know how to express it. But need notion of feedback.
- Ras: should accept that things are not perfect--for this expedition goal should be to provide something more reliable than testing & code inspection.
- Moshe/his student: paradox of confidence & competence in programmers
- Alberto: we are trying to reach different fields and we are borrowing a lot from approximation theory. This is particularly well done in continuous domain. How do we choose a set of functions that approximate well? Goal of program synthesis is to eliminate the programmers.
- Madusudan: one of main challenges is to capture programmer specifications. If you can capture human intuition for large set of examples, then people can adopt this.
- John Field: one outcome that would be nice is to come up with catalog of techniques and where they work well vs. not.
- Sanjit: we are still figuring this out, but the four application domains we have identified provide some guidance as two which techniques work well in each application domain.
- Mark Wegman: when you look at core domains, need to study them. Some problems that we think are important are not. Does not believe majority of bugs come from incorrect translations of specifications into code, but for people making errors because they did not think of something or had wrong assumptions about what functions do. Suspect problem of having code nobody understands is a big one--need to study programmers and core domains to understand what is going on. Need to study what will work well and what will have an impact.
- Paulo: on point of how to pick metrics and costs, this is important but just another way of specifying what the programmer should do.
- Sosmit: as soon as you interact with humans, coming up with even
partial models of env becomes difficult. Trying to synthesize env assumptions is a big challenge. If we want to port synthesis to real world, will need to do this. There should be more research in this direction.