Compositional Temporal Synthesis

Moshe Y. Vardi

Rice University
What Good is Model Checking?

Model Checking:

- **Given**: Program $P$, Specification $\varphi$.
- **Task**: Check that $P \models \varphi$

Success:

- **Algorithmic methods**: temporal specifications and finite-state programs.
- **Also**: Certain classes of infinite-state programs
- **Tools**: SMV, SPIN, SLAM, etc.
- **Impact** on industrial design practices is increasing.

Problems:

- Designing $P$ is hard and expensive.
- Redesigning $P$ when $P \not\models \varphi$ is hard and expensive.
Automated Design

Basic Idea:

• Start from spec $\varphi$, design $P$ such that $P \models \varphi$.

  Advantage:
  – No verification
  – No re-design

• Derive $P$ from $\varphi$ algorithmically.

  Advantage:
  – No design

In essence: Declarative programming taken to the limit.
Program Synthesis

**The Basic Idea:** Mechanical translation of human-understandable task specifications to a program that is known to meet the specifications.

**Deductive Approach** (Green, 1969, Waldinger and Lee, 1969, Manna and Waldinger, 1980)

- Prove *realizability* of function, e.g., \( (\forall x)(\exists y)(Pre(x) \rightarrow Post(x, y)) \)

- Extract *program* from realizability proof.

**Classical vs. Temporal Synthesis:**

- **Classical:** Synthesize transformational programs

- **Temporal:** Synthesize programs for ongoing computations (protocols, operating systems, controllers, etc.)
Synthesis of Ongoing Programs

**Specs:** Temporal logic formulas

**Early 1980s:** Satisfiability approach
(Wolper, Clarke+Emerson, 1981)

- **Given:** $\varphi$
- **Satisfiability:** Construct $M \models \varphi$
- **Synthesis:** Extract $P$ from $M$.

**Example:** always $(\text{odd} \rightarrow \text{next } \neg \text{odd}) \land$
always $(\neg \text{odd} \rightarrow \text{next odd})$

\[
\text{odd} \xrightarrow{} \text{odd}
\]
Reactive Systems

Reactivity: Ongoing interaction with environment (Harel+Pnueli, 1985), e.g., hardware, operating systems, communication protocols, etc.

Example: Printer specification –

$J_i$ - job $i$ submitted, $P_i$ - job $i$ printed.

- **Safety**: two jobs are not printed together
  
  always $\neg(P_1 \land P_2)$

- **Liveness**: every job is eventually printed
  
  always $\land_{j=1}^{2}(J_i \rightarrow \text{eventually } P_i)$
Satisfiability and Synthesis

**Specification Satisfiable?** Yes!

*Model* $M$: A single state where $J_1$, $J_2$, $P_1$, and $P_2$ are all false.

**Extract program from** $M$? No!

**Why?** Because $M$ handles only one input sequence.

- $J_1$, $J_2$: input variables, controlled by environment
- $P_1$, $P_2$: output variables, controlled by system

**Desired:** a system that is receptive to all input sequences.

**Conclusion:** Satisfiability is inadequate for synthesis.
Realizability

$I$: input variables, $O$: output variables

Game:

- **System**: choose from $2^O$
- **Env**: choose from $2^I$

Infinite Play:

$i_0, i_1, i_2, \ldots$

$0_0, 0_1, 0_2, \ldots$

Infinite Behavior: $i_0 \cup o_0, i_1 \cup o_1, i_2 \cup o_2, \ldots$

Win: behavior $\models$ spec

Specifications: LTL formula on $I \cup O$

Strategy: Function $f : (2^I)^* \rightarrow 2^O$

Realizability: Abadi+Lamport+Wolper, 1989
Existence of winning strategy for system.

Synthesis: Pnueli+Rosner, 1989
Extraction of winning strategy for system.
Church’s Problem

Church, 1957: Realizability problem wrt specification expressed in MSO (monadic second-order theory of one successor function)

Büchi+Landweber, 1969:

- Realizability is decidable.
- If a winning strategy exists, then a finite-state winning strategy exists.
- Realizability algorithm produces finite-state strategy.


**Question:** LTL is subsumed by MSO, so what did Pnueli and Rosner do?

**Answer:** better algorithms!
Post-1972 Developments

- **Pnueli, 1977**: Use LTL rather than MSO as spec language.

- **V. + Wolper, 1983**: Elementary (exponential) translation from LTL to automata.

- **Safra, 1988**: Doubly exponential construction of tree automata for strategy trees wrt LTL spec (using V. + Wolper).

- **Pnueli + Rosner, 1989**: 2EXPTIME realizability algorithm wrt LTL spec (using Safra).

- **Rosner, 1990**: Realizability is 2EXPTIME-complete.
Standard Critique

**Impractical!** 2EXPTIME is a horrible complexity.

**Response:**

- 2EXPTIME is just worst-case complexity.

- 2EXPTIME lower bound implies a doubly exponential bound on the size of the smallest strategy; thus, hand design cannot do better in the worst case.
Real Critique

- Algorithmics not ready for practical implementation.
- Complete specification – unrealistic.
- Construction from scratch – unrealistic.

**Response**: More research needed!

- Better algorithms
- Incremental synthesis – write spec incrementally.
- *Compositional synthesis* – synthesis from components.
Synthesis from Components

**Basic Intuition:** [Lustig+V., 2009]

- In practice, systems are typically not built from scratch; rather, they are constructed from existing components.
  - **Hardware:** IP cores, design libraries
  - **Software:** standard libraries, web APIs
  - **Example:** mapping application on smartphone
    - location services, Google maps API, graphics library
- Can we automate “construction from components”?

**Setup:**

- **Library** $L = \{C_1, \ldots, C_k\}$ of component types.
- **Linear temporal specification:** $\varphi$

**Problem:** Construct a finite system $S$ that satisfies $\varphi$ by composing components that are *instances* of the component types in $L$.

**Question:** What are components? How do you compose them?
Components I: Transducers

Transducer: A simple model of a reactive system – a finite-state machine with inputs and outputs (Moore machine).

- Transducers are receptive.
- Output depends on state alone.
Dataflow Synthesis from Components

Setup:

- **Components**: multi-input multi-output transducers e.g., hardware IP blocks
- **Dataflow composition**: connect input and output ports so outputs become inputs, e.g., connect sequential circuits

**Theorem**: [Lustig+V., 2009]
Dataflow synthesis from components is undecidable.

**Crux**:

- Number of component instances not bounded, a priori.
- Cell of Turing-machine tape can be viewed as a component, connected to cells to its left and right.
Components II: Transducers with Exits

![Diagram of transducers with exits]

- States: $q_0$, $q_1$, $q_2$, $q_3$
- Transitions:
  - $a$: $q_0 ightarrow q_1$
  - $b$: $q_0 ightarrow q_2$, $q_1 ightarrow q_3$, $q_3 ightarrow q_0$, $q_2 ightarrow q_0$
- Start state: $q_0$
- Accepting states: $q_1$, $q_3$

15
Control-flow Composition I

**Motivation:** Software-module composition – exactly one component interacts with environment at one time; on reaching an exit state, *goto* start state of another component.

A library of two components: \( L = \{ M_1, M_2 \} \)
Pick three component instances from $L$:
Control-flow Composition III

Connect each exit to some start state – resulting composition is a transducer and is receptive.
Controlflow Synthesis

Setup:

- **Components**: single-input single-output transducers with *exit states*, e.g., software module
- **Controlflow composition**: upon arrival at an exit state, *goto* start state of another component – composer chooses target of branch.
- *No* a priori bound on number of component instances!

**Theorem**: [Lustig+V., 2009]
Controlflow synthesis from components is 2EXPTIME-complete.

Crux:

- Consider general (possible infinite) composition trees, that is, unfoldings of compositions
- Use alternating automata to check that all possible computations wrt composition satisfy $\varphi$
- Show that if general composition exists then finite composition exists.
Controlflow Synthesis from Recursive Components

Key Idea: Use call and return, instead of goto.

- An online store may call the PayPal web service, which receives control of the interaction with the user until it returns the control to the online store with approval/disapproval of payment.

Modeling:

- calls: component has a set of call states; when a call state is reached, another component is called.
- returns: component has a set of return states; when a return state is reached, control returns to the calling component.
- re-entry: component has a set of re-entry states; when control returns to a component, the component enters a re-entry state.
- return value: modeled by means of re-entry states.
- call value: not modeled explicitly here.
Recursive Components

Setup:

- **Components**: single-input single-output transducers with call states, return states, and re-entry states
- **Controlflow composition**: calls and returns

Related: recursive state machines of Alur at el.

- The result of composing recursive components is a recursive state machine.
- Equivalent to an infinite-state transducer.
Specifying Call-and-Return Computations

Need: In a call-and-return computation, specification may need to refer to call-and-return structure [Alur-Etessami-Madhusudan, 2004]

- E.g., “if the pre-condition \( p \) holds when a procedure \( A \) is called, then if \( A \) terminates, then the post-condition \( p \) is satisfied upon return.

Solution: Alur et al.

- Nested Word: Description of call-and-return computations – sequence of letters, plus calls, and matching returns, when exist
  – Traces of pushdown machines with pushes and pops made visible
- Nested-Word Temporal Logic (NWTL): logic refers to call-and-return structure
  – next: refers to next state
  – next_{\mu}: refers to return that matches a call

Now: Controlflow synthesis from recursive components wrt NWTL properties.
Automata-Theoretic Approach

Key Idea of Temporal Synthesis:

- Use tree automata to accept “good” strategy trees
- Use word automata to accept “good” tree branches

V. Wolper, 1983: Exponential translation from LTL to Büchi automata

Needed Aere: automata-theoretic counterpart to NWTL

Answer: NWBA – Nested-Word Büchi Automata [Alur et al., 2008]

- Standard transition relation
- Call transition relation
- Return transition relation

Theorem: Exponential translation from NWTL to NWBA
Automata-Theoretic Approach to Controlflow Synthesis

Key Idea Lustig+V, 2009

• Composition tree is bad if it enables a computation that violates $\varphi$, i.e., accepted by NBW $A_{\neg\varphi}$.
• Construct NBT that searches for a bad computation by guessing a computation and simulating $A_{\neg\varphi}$.
• Complement NBT and test for nonemptiness.

Extending to Recursive Components:

• Computations go up and down the composition tree – use 2-way automata to track them.
• Need to have an NBT simulate NWTL – NBT needs to track cycles, from call to return and back to call.

Bottom Line: Doable, but construction is rather messy. Complexity: 2EXPTIME-complete.

Question: Can construction be simplified?

• Note: using alternation and 2-wayness simplified earlier messy automata-theoretic constructions.
Controlflow Synthesis from Probabilistic Components

**Goal**: Build reliable systems from unreliable components.

- **Example**: How do you turn a fair coin into a completely biased coin?

- What are probabilistic components?

- How are they connected together?

- What is the specification formalism?

- What is the appropriate notion of realizability?
**Probabilistic Components**

**Examples:** noisy sensors, probabilistic CMOS

**Probabilistic Components:** transducers with *exit states* and *probabilistic transition function.*
Probabilistic Components

A probabilistic component is a probabilistic transducer w. exits – \((\Sigma_I, \Sigma_O, Q, q_0, \delta, F, L)\):

- \(Q\) — finite set of states
- \(q_0\) — start state
- \(F \subseteq Q\) — set of exit states
- \(\Sigma_I\) and \(\Sigma_O\) — input and output alphabets
- \(\delta : Q \times \Sigma_I \rightarrow \text{Dist}(Q)\) — Transition function that assigns a prob. distribution to state/input pairs
- \(L : Q \rightarrow L\) — output function

**Input:** \(w \in \{a, b\}^\omega\)

**Output:** probability distribution on \(\{0, 1\}^\omega\)
Pick three component instances from library $L = \{M_1, M_2\}$:
Controlflow Composition

Connect each exit to some start state – resulting composition is a probabilistic transducer.
Modeling Controlflow Composition

**Crux**: (Current component, Exit state) → Next state

**Composer**: A *deterministic* transducer that captures controlflow in a composition.

- Composer — describes how to connect components
- Composition — resulting probabilistic transducer
DPW Specification

**DPW** — Deterministic Parity Word Automaton $A$

- Each state of $A$ has a **priority** (a natural number).

- $A$ **accepts** an infinite word if the corresponding run of $A$ satisfies the **parity condition**.

- **Parity condition**: the lowest priority that occurs infinitely often is even.

DPW can express all $\omega$-regular specifications.

LTL can be translated to DPW.
Probabilistic Correctness

**Key Idea**: System must satisfy DPW specification in face of *every* possible input.

- With prob 1, the run of the system is accepted by DPW.
- Probability defined by input: so which input?
- We assume adversarial environment: for every possible input, with prob 1, DPW must accept.
**Probabilistic Realizability**

**Environment Strategy**: The environment probabilistically chooses the next input depending upon history of the system.

- Environment: a function $f : Q^* \rightarrow \text{Dist}(\Sigma_I)$

- Each strategy $f$ induces a probability distribution $\mu_f$ on the set of runs of $M$.

- Environment *wins* if run of $M$ is rejected by $A$ with probability $> 0$.

**Realizability**: System $M$ *realizes* spec $A$ iff the environment has no winning strategy against $M$. 

Controlflow Synthesis from Probabilistic Components

- What are probabilistic components? *Probabilistic Transducers w. Exits*
- How are they connected together? *Deterministic Controlflow*
- What is the specification formalism? *DPW*
- What is the appropriate notion of realizability? *Probabilistic*
- What is the object being synthesized? Composer

**Note:** Components are now probabilistic, but controlflow is still deterministic.

**DPW Synthesis problem:** Given library \( L \) and DPW \( A \), find composer \( C \) over \( L \) such that the composition defined by \( C \) realizes \( A \).

**Theorem:** [Lustig-Nain-V., 2011] DPW synthesis from probabilistic components is *decidable*
# Embedded-Parity Synthesis: Simplifying DPW Synthesis

**Key Idea:** Instead of using a DPW as spec, assign priorities directly to each state of each component in the library and use the parity condition.

<table>
<thead>
<tr>
<th>DPW Synthesis</th>
<th>Embedded-Parity Synthesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification given as DPW</td>
<td>Specification embedded as priorities of component states</td>
</tr>
<tr>
<td>Environment wins if output rejected by DPW with prob. &gt; 0</td>
<td>Environment wins if output satisfies parity condition with prob &lt; 1</td>
</tr>
<tr>
<td>Natural problem</td>
<td>Artificial problem</td>
</tr>
</tbody>
</table>
Theorem: [Lustig-Nain-V., 2011] embedded-parity synthesis from probabilistic components is decidable in EXPTIME.

Proof Idea:

- Composer is finite, so composition is finite.
- Suffices to focus on pure, memoryless environment strategies.
- Finite probabilistic transducer + pure, memoryless environment strategy = Markov chain.
- Apply ergodic analysis: with prob 1, limit behavior in ergodic set.
- Unfold chain into tree, translating ergodicity onto tree.
- Construct Büchi tree automaton for bad composition trees.
- Complement automaton and check nonemptiness.
Theorem: [Lustig-Nain-V., 2011] DPW synthesis from probabilistic components is decidable in 2EXPTIME.

- **Proof Idea:** Take product of components in library $L$ with DPW $A$ and reduce to embedded-parity synthesis.

- **Difficulty:** Transitions of composers must depend only on components, cannot depend on states of $A$.

- **Solution:** Use techniques from synthesis with incomplete information, pay another exponential in complexity.

- **Note:** Upper bound in 4EXPTIME for LTL spec.
Controlflow Composition

Questions:

- If components are probabilistic why not allow probabilistic controlflow?
- Is probabilistic controlflow more powerful than deterministic controlflow?

**Theorem:** [Nain&V., 2012] Probabilistic and deterministic composers have the same expressive power for embedded-parity specifications.

**Theorem:** [Nain&V., 2012] Probabilistic composers are more expressive than deterministic composers for DPW specifications.

Similar to memory vs randomness tradeoff in games [[Chatterjee-De Alfaro-Henzinger, 2004]].
Synthesizing Probabilistic Composers

Main difficulties:

- **Expressiveness Barrier:**
  - For deterministic composers, DPW synthesis is solved via embedded parity.
  - Expressiveness result rules this out for probabilistic composers.

- **Unbounded Branching of Tree Representation:**
  - For deterministic composers:
    * branching of transition function is bounded.
    * depends on number of exits, fixed for given library.
    * So automata-theoretic techniques can be used.
  - For probabilistic composers:
    * branching of transition function is potentially unbounded.
    * depends on size of composition.
Synthesizing Probabilistic Composers

**Theorem:** [Nain&V., 2012] Controlflow synthesis of probabilistic composers from probabilistic components is decidable.

**Proof Idea:** Simulate probabilistic controlflow via deterministic controlflow.

- Add to library a component $M_D$ whose sole purpose is to express probabilistic branching.
- Modify spec to ignore $M_D$. 

![Diagram](attachment:diagram.png)
In Conclusion

**Framework**: Compositional Synthesis = Synthesis from Component Libraries:

- What types of components?
- How are components composed?
- How are requirements specified?

**Future Work**

- Connection to games with incomplete information
- Tighter bounds
- Better algorithms