Automatic Verification of Communicating Data-Aware Web Services

Victor Vianu
U.C. San Diego

Joint work with Alin Deutsch, Liying Sui, Dayou Zhou

Web service: service hosted on the Web

- Interactive, often data-driven: accesses an underlying database and interacts with users/programs according to explicit or implicit workflow
- Complex services: Web service compositions peers communicating asynchronously
- Complexity of workflow leads to bugs: see the public database of Web site bugs (Orbitz bug)
- Static analysis required - behavior of individual peers - protocols of communication between peers - global properties

This talk: Automatic, sound and complete verification of data-aware Web service compositions

- Abstraction of web service compositions: communicating data-aware reactive systems
- Verification of single peers and compositions
- Experimental results for single peer verification WAVE verifier

Our target: data-aware Web services
Examples of Desirable Properties

- **Semantic properties**
  - “no product is delivered before payment in the right amount is received”
  - “no user can cancel an order that has already been shipped”

- **Basic soundness of specification**
  - “conditions guarding transition to next Web page are mutually exclusive”

- **Navigational properties**
  - “the shopping cart page is reachable from any page”

Examples of Composition Properties

- “every payment request by a user results eventually in an approval or denial output to the user”

- “the answer to every credit check request message for a user is a credit rating message poor, fair, or good, for the same user”

- “for every two consecutive credit rating messages for the same user there exists an intermediate credit request message for that user.”
Typical previous abstractions of Web services compositions: communicating Mealy machines

Our abstraction: communicating reactive systems with FO control

Control: (input, state, db) ◊ (output, state)

Typical Web service verification problem

Temporal property of conversations: sequence of exchanged messages

LTL properties: Every authorize followed by some bill?

Relational Transducers
Abiteboul+Vianu 1998

Abstract State Machine Transducers
Marc Spielmann 2000

Here: extension + communication

History
Control: \((\text{input, state, db}) \diamond (\text{output, state})\)

Technical point: queries can also refer to \(k\) previous inputs
**Configurations and runs**

- Run: infinite sequence of consecutive configurations

**More on messages**

- Flat message: single tuple
- Nested message: finite set of tuples
- Messages queued at recipient
- Message contents:
  \[ !M(x) :- \text{query}(db, \text{state}, \text{input}, \text{in-messages}) \]

Flat messages: query may generate several tuples, choose non-deterministically one to be sent

**Communicating peers:** composition

- Channels between peers
- Message: finite relation (set or singleton)
- One FIFO queue at recipient of each channel

**Peers with messages**

Control: (input, in-messages, state, db) ◊ (output, out-messages, state)
Configurations and runs

Configuration of a single peer

Configuration of a composition: member peer configurations

Transitions: one peer at a time

Run: infinite sequence of consecutive configurations
Language for properties of runs: LTL-FO

FO + LTL operators + Boolean operators

- Start with FO formulas referring to the states, db, inputs, top and last message of queues in current configuration
- Apply Boolean and LTL operators: $\text{X, U, F, G, B}$
- All remaining free variables are universally quantified

$\forall \bar{x} \varphi(\bar{x})$

Example Property

“any shipped product must be previously paid for”

$\forall \text{pid, uname, price} \left[ \xi(\text{pid, uname, price}) \text{ B Ship(uname, pid)} \right]$

Where $\xi(\text{pid, uname, price})$ is the formula

Typical approaches in Software Verification are unsatisfactory:

- Model checking: developed for finite-state systems described by propositional states. More expressive specifications first abstracted to propositional ones.

Unsatisfactory: can check that some payment occurred before some shipment, but not that it involved the correct amount and product.

- Theorem proving: no completeness guarantees, not autonomous. Prover requires expert guidance.

Our approach: identify a restricted but reasonably expressive class of compositions that can be verified
Main restrictions for decidability

- **bounded queues**, guarded quantification
- guarded quantification: quantified variables must appear in input or (flat) message atoms

“input boundedness”
earlier variant: Spielmann

```
pick(pid,price) ← ∃ram ∃cpu prev-search(ram,cpu) ∧ catalog(pid,ram,cpu,price)
```

Input-bounded compositions

- State, output, and nested message rules use FO formulas with guarded quantification:
  \[
  \exists x \ ( \text{guard}(x) \land \neg x) \\
  \forall x \ ( \text{guard}(x) \Rightarrow \neg x)
  \]
  where guard is an input or flat message atom and state and nested message atoms in \neg have no quantified variables

- Input options and flat message definitions:
  \[
  \exists \ast FO \text{ formulas with ground state and nested message atoms}
  \]

Input-bounded LTL-FO property:
FO components are input bounded

```
∀x G [ X reject-order(x) → (past-order(x) ∧ ¬ ∃y (pay(x,y) ∧ price(x,y))]
```

```
∀x G [ X reject-order(x) → (past-order(x) ∧ ¬ ∃y (pay(x,y) ∧ price(x,y))]
```

Main verification result

Theorem: It is **decidable** whether an input-bounded composition with bounded queues and lossy channels satisfies an input-bounded LTL-FO property.

```
Theorem: It is **decidable** whether an input-bounded composition with bounded queues and lossy channels satisfies an input-bounded LTL-FO property.
```

Complexity: **PSPACE-complete** for bounded arity schemas, **EXPSPACE** otherwise
**Tightness:** even small extensions lead to undecidability

- Relaxing the requirement that state atoms must be ground in formula defining the input options.  
  Reduction: Does TM halt on input epsilon?
- Lifting the input-bounded requirement by allowing state projection.  
  Reduction: Implication for functional and inclusion dep
- Allowing perfect flat queues.  
  Reduction: Post Correspondence Problem
- Disallowing non-deterministic choice for flat messages 
  Reduction: Post Correspondence Problem

**Expressivity** of input-bounded specs

Significant parts of the following Web applications could be modeled:

- Dell-like computer shopping website
- Expedia
- Barnes&Noble
- GrandPrix motor sports Web site

See demo site http://www.db.ucsd.edu

**PSPACE verification:** outline for single peer

To check that $C$ satisfies $\psi$, verify that there is no run satisfying $\neg \psi$

Recall model checking approach (finite-state):
- Build Büchi automaton $A(\neg \psi)$ for $\neg \psi$
- Build automaton $R$ accepting all runs
- Check that there is no counterexample run: emptiness of $R \times A(\neg \psi)$

**Our case:** infinite-state system

Same idea: build $A(\neg \psi)$, then search for counterexample runs accepted by $A(\neg \psi)$

But: no automaton $R$ for the runs!

Problem in searching for counterexample runs: infinite runs, infinitely many underlying databases

How to limit the search space?
Infinite search space for runs

Bounding the search for counterexample runs

Key insight for PSPACE complexity

- No need to explicitly materialize entire configuration:
- Instead, at each step construct only those portions of DB, states and outputs which can affect property.
- Call them pseudoconfigurations.

Pseudoconfigurations

\[ C = \text{a set of relevant constants extracted from the spec. and prop.} \]

+ a fixed number of variables

\[ \text{restriction of states to constants in } C \]

\[ \text{restriction of DB to } C \]

\[ \text{restriction of outputs to constants in } C \]

\[ \text{input picked from } C \]

\[ \text{output} \]

Size polynomial in spec + prop
Pseudoruns

\[ \exists \text{counterexample run} \iff \exists \text{counterexample pseudorun} \]

• Can compute next possible pseudoconfigurations from current one
• Never construct entire DB, just "slide" poly window over it

Verification of compositions

\[ \text{PSPACE verification algorithm} \]

• Can compute next possible pseudoconfigurations from current one

\[ \text{PTIME reduction preserving input boundedness} \]

• Reduction applies to input-bounded compositions with bounded, lossy channels
• Flat message queues simulated by inputs
• Nested message queues simulated by states
• Non-deterministic choice of peer at each transition simulated with additional input
• Some tricky timing issues in translation of property

Reduce to single peer verification
Additional verification problems

- Conversation protocols
  sequences of messages observed in runs
  data-agnostic: message parameters ignored
  data-aware: parameters taken into account

- Modular verification
  specs of some peers not available
  information limited to input/output behavior

Verification of conversation protocols

- data-agnostic protocol: Büchi automaton over alphabet of message names
- Possible semantics with lossy channels:
  observer-at-recipient
  observer-at-source

  Theorem: It is undecidable if an input-bounded composition with bounded, lossy channels satisfies a data-agnostic conversation protocol with observer-at-source semantics

Verification of conversation protocols

- Similar results for data-aware protocols:
  formalized as Büchi automaton whose alphabet is a finite set of FO formulas on message relations
  \[ G( \text{get-rating}(x) \ B \text{rating}(x,y) ) \]

  Theorem: It is PSPACE-complete if an input-bounded composition with bounded, lossy channels satisfies a data-aware conversation protocol with observer-at-recipient semantics

Modular verification

Black box peers: input-output behavior
Modular verification

Environment specification: LTL-FO description of input and output messages

Properties under given environment

Composition C satisfies LTL-FO property $\varphi$ under environment specification $\psi$: every run of C in which messages to/from the environment satisfy $\psi$ and use values from some finite domain, satisfies $\varphi$

Verification under given environment

Additional restriction needed for decidability

LTL-FO property $\psi$ is strictly input-bounded if its FO components have no free variables

Example:

$G \forall ssn \ [ \text{getRating}(ssn) \Rightarrow (\neg \text{rating}(ssn, \text{“poor”}) \lor \text{rating}(ssn, \text{“fair”}) \lor \neg \text{rating}(ssn, \text{“good”})) ]$

Verification under given environment

Theorem: It is undecidable if an input-bounded composition C with bounded queues and lossy channels satisfies an input-bounded LTL-FO property $\varphi$ under an input-bounded but not strictly-input-bounded environment specification $\psi$
Putting the pieces together

WebML-style spec of Web service composition

peer composition spec

single peer spec

Implementation so far
WAVE: verifier for single Web service peer
[SIGMOD'05]

• Essentially implements search for a counterexample pseudorun
• Many tricks and heuristics to achieve good verification times

Some techniques

• Dataflow analysis to identify all constants to which a DB attribute may be compared (directly or indirectly).

Limits the relevant combinations of constants when constructing partial DBs. Spectacular reduction: for the computer shopping website, from $2^{(17,270,412,688)}$ partial DBs to 8!

• Internal representation of pseudoconfigs to
  – Efficiently detect loop in periodic run
  – Efficiently evaluate queries

• Early pruning of pseudoruns

Experimental Evaluation of WAVE Tool

• Online Demo at http://www.db.ucsd.edu/

• Evaluated experimentally on 4 Web applications:
  – Dell-like computer shopping
  – Part of Expedia, Barnes&Noble, GrandPrix

• Verification times for a battery of properties:
  all within seconds, below one minute.

• Here, report only Dell experiment. All others are similar.
Some of the Verified Properties

<table>
<thead>
<tr>
<th>Property type</th>
<th>Property name</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence $pBq$</td>
<td>$P5$ (true)</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>$P7$ (true)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$P9$ (true)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$P10$ (true)</td>
<td>0.23</td>
</tr>
<tr>
<td></td>
<td>$P11$ (false)</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>$P12$ (true)</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>$P13$ (false)</td>
<td>0.44</td>
</tr>
<tr>
<td>Response $p \not \Diamond Fq$</td>
<td>$P14$ (false)</td>
<td>0.19</td>
</tr>
<tr>
<td>Reachability $Gp$ or $Fq$</td>
<td>$P2$ (true)</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>$P3$ (false)</td>
<td>0.37</td>
</tr>
<tr>
<td>Recurrence $G(Fp)$</td>
<td>$P17$ (false)</td>
<td>0.15</td>
</tr>
<tr>
<td>Strong non-progress $F(Gp)$</td>
<td>$P15$ (false)</td>
<td>0.26</td>
</tr>
<tr>
<td>Weak non-progress $G(p \not \Diamond Xp)$</td>
<td>$P6$ (false)</td>
<td>0.49</td>
</tr>
<tr>
<td>Guarantee $Fp$</td>
<td>$P1$ (true)</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>$P8$ (false)</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Property name 0.02 Guarantee $Fp$ 0.37 Response $p \not \Diamond Fq$ 0.19 Reachability $Gp$ or $Fq$ 0.9 Recurrence $G(Fp)$ 0.15 Strong non-progress $F(Gp)$ 0.26 Weak non-progress $G(p \not \Diamond Xp)$ 0.49 Guarantee $Fp$ 0.02

Failure of Classical Tools

- SPIN model checker
  Abstraction is unsatisfactory.
  Alternative trick:
  Try to use SPIN to verify pseudoruns.
  The resulting SPIN input is too large to handle.

- PVS theorem prover
  Not guaranteed to find a counterexample.
  Gets stuck during search, asks for guidance from expert user.

Conclusions

- Sound and complete verification for a significant class of database-driven (hence infinite-state) Web services and their compositions.
- Encouraging experimental results for single peers.
- Coupling of database and model-checking techniques is extremely effective.
- Database-driven Web applications may be unusually well suited for automated verification.
- Significant to both the database and automatic verification areas.

Demo Site

http://www.cs.ucsd.edu/users/lsui/project/index.html

Papers

Single peer verification: PODS 2004 invited to JCSS also results on CTL, CTL*
Branching-time temporal properties

Current state

Need path quantifiers

Branching-time temporal properties

• Computation tree logic (CTL*|CTL)
  Add path quantifiers:
  • A---“for every path”
  • E---“there exists a path”

Computation tree logic (CTL)

Verification results for CTL(*)

• Propositional transducers:
  --states and outputs are propositional
  --prev-I atoms are disallowed

• CTL* formulas using state, output, and inputs interpreted as propositions

From every page, there is a way back to the home page

(AGEF)homepage
• Verification of CTL(*) formulas for propositional transducers:
  --CO-NEXPTIME for CTL
  --EXPSPACE for CTL

Proof idea:
(i) show that there is a bound on the databases that need to be considered in order to detect a violation;
(ii) for a fixed database, reduce checking violation to model checking for a Kripke structure generated from the database.

Getting down to PSPACE:
• Fully propositional transducers: inputs are also propositional

Reduce to checking emptiness of a one-letter word HAA.

Alternative restriction: capturing “user-driven search”

• Propositional states and actions
  • Inputs are monadic, propagated using prev-I atoms

Example: allows conducting a user-driven search going through consecutive stages of refinement

For transducers with “user-driven search”:

• CTL formulas can be verified in EXPTIME
• CTL* formulas can be verified in 2-EXPTIME

EXPTIME for fixed out-degree of input choice

Proof: reduce to satisfiability of CTL(*) formulas by a Kripke structure