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
Triggers state update and transition to new page.
Message Page (MP)

Message

cancel

Home Page (HP)

NAME:
PASSWD:

login

Customer Page (CP)

laptop
desktop

Laptop Search (LSP)

RAM:
CPU:
SCREEN:

submit

Desktop Search (DSP)

RAM:
CPU:

submit

Matching products

Product Index (PIP)

Details

Product Detail (PDP)

buy

Confirmation

Confirmation (CoP)

print

High-level WebML-style specification tools

DB

output
Product Info Page (PIP)

Input: \textit{pick}(pid,price)

Input options:

\textit{pick}(pid,price) \leftarrow \exists \text{ram} \exists \text{cpu} \newline \textit{prev-search}(\text{ram},\text{cpu}) \land \text{catalog}(pid,\text{ram},\text{cpu},price)
Product Info Page (PIP)

Input: $\text{pick}(\text{pid}, \text{price})$

Input options:

$$\text{pick}(\text{pid}, \text{price}) \leftarrow \exists \text{ram} \ \exists \text{cpu} \ \ \text{prev-search}(\text{ram}, \text{cpu}) \ \land \ \text{catalog}(\text{pid}, \text{ram}, \text{cpu}, \text{price})$$

Matching products

Product Index (PIP)
Product Info Page (PIP)

Input: $\text{pick}(pid, price)$

Input options:

$$\text{pick}(pid, price) \leftarrow \exists \text{ram} \exists \text{cpu} \\ 
\text{prev-search}(\text{ram}, \text{cpu}) \land \text{catalog}(pid, \text{ram}, \text{cpu}, price)$$

Matching products

Product Index (PIP)
High-level WebML-style specification
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”
Compositions of Web services
Compositions of Web services
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
Typical Web service verification problem
temporal property of conversations: sequence of exchanged messages

LTL properties: Every authorize followed by some bill?
Our abstraction: communicating reactive systems with FO control

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

• Relational Transducers
  Abiteboul+Vianu 1998

• Abstract State Machine Transducers
  Marc Spielmann 2000

• Here: extension + communication
Control: (input, state, db) $\diamondsuit$ (output, state)
Single peer

state

db
Single peer
Single peer

state ➔ FO query ➔ db

---

---
Input options

Single peer

state

FO query

db
user choice

Input options

Single peer

state

FO queries

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

Configuration

- input
- state
- db
- output
Run: infinite sequence of consecutive configurations
• Communicating peers: composition

- channels between peers
- message: finite relation (set or singleton)
- one FIFO queue at recipient of each channel
More on messages

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

- Flat message: single tuple
- Nested message: finite set of tuples
- Messages queued at recipient
- Message contents:

  \[ !M(x) \leftarrow \text{query}(db, \text{state}, \text{input}, \text{in-messages}) \]

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

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

Configuration of a single peer

- Input
- State
- DB
- Output
- Incoming message queues
Configuration of a composition: member peer configurations
Configuration of a composition: member peer configurations

Transitions: one peer at a time
Configuration of a composition: member peer configurations

Transitions: one peer at a time
Configuration of a composition: member peer configurations

Transitions: one peer at a time
Configuration of a composition: member peer configurations

Transitions: one peer at a time
Configuration of a composition: member peer configurations

Transitions: one peer at a time
Configuration of a composition: member peer configurations

Transitions: one peer at a time
Configuration of a composition: member peer configurations

Transitions: one peer at a time
Configuration of a composition: member peer configurations

Transitions: one peer at a time
Configuration of a composition: member peer configurations

Transitions: one peer at a time
Configuration of a composition: member peer configurations

Transitions: one peer at a time
Configuration of a composition: member peer configurations

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
  - FO components
- Apply Boolean and LTL operators: \( X, U, F, G, B \)
- All remaining free variables are universally quantified
  - \( \forall x \varphi(x) \)
Example Property

“any shipped product must be previously paid for”

\[ \forall \text{ pid, uname, price } [\xi(\text{ pid, uname, price}) \lor \text{ Ship(uname, pid)}] \]

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

\[ \text{ pay(price)} \land \text{ picked(uname, pid, price)} \land \text{ prod-price(pid, price)} \]
Example Property

“any shipped product must be previously paid for”

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

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

\[ \text{pay(\text{price})} \land \text{picked(\text{uname}, \text{pid}, \text{price})} \land \text{prod-price(\text{pid}, \text{price})} \]
Example Property

“any shipped product must be previously paid for”

∀ pid, uname, price [ξ(pid, uname, price) \(\text{B Ship(uname, pid)}\)]

Where ξ(pid, uname, price) is the formula

\[
\text{pay(price) \&\& picked(uname, pid, price) \&
prod-price(pid, price)}
\]
Example Property

“any shipped product must be previously paid for”

∀pid, uname, price [ξ(pid, uname, price) \(\mathbf{B}\) Ship(uname, pid)]

Where \(\xi\)(pid, uname, price) is the formula

\[
\text{pay(price)} \land \text{picked(uname, pid, price)} \land \\
\text{prod-price(pid, price)}
\]
Example Property

“any shipped product must be previously paid for”

∀pid, uname, price \[ [ξ(pid, uname, price) \land Ship(uname, pid)] \]

Where ξ(pid, uname, price) is the formula

pay(price) \land picked(uname, pid, price) \land prod-price(pid, price)
The Verification Problem

Given composition $C$ and LTL-FO property $\varphi$

Decide if every run of $C$ satisfies $\varphi$. If not, exhibit a counterexample run.
The Verification Problem

Given composition $C$ and LTL-FO property $\varphi$

Decide if every run of $C$ satisfies $\varphi$. If not, exhibit a counterexample run.

Challenge: infinite-state system!
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.
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
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

\[
pick(pid,price) \leftarrow \exists ram \exists cpu \text{ prev-search}(ram,cpu) \wedge \text{catalog}(pid,ram,cpu,price)
\]
Input-bounded compositions

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

- Input options and flat message definitions:
  \[ \exists \text{FO} \] formulas with ground state and nested message atoms
Input-bounded LTL-FO property:
FO components are input bounded

“An order is rejected in the next step only if it has already been ordered but not paid correctly in the current input”

$$\forall x \ G [ X \ reject-order(x) \rightarrow (past-order(x) \land \neg \exists y \ (pay(x,y) \land price(x,y))) ]$$
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.

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

• Relaxing the requirement that state atoms must be ground in formula defining the input options.
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?
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.
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
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.
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
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
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

number of underlying DBs

length of run
Bounding the search for counterexample runs
Bounding the search for counterexample runs

Periodic runs suffice: \( \exists \) counterexample iff \( \exists \) periodic one
Bounding the search for counterexample runs

Sufficient to consider only DBs over a fixed domain of cardinality exponential in size of spec + prop

Periodic runs suffice: \( \exists \) counterexample iff \( \exists \) periodic one
Bounding the search for counterexample runs

Sufficient to consider only DBs over a fixed domain of cardinality exponential in size of spec + prop

Finite search space yields **decidability of verification**

**Periodic** runs suffice: \( \exists \) counterexample iff \( \exists \) periodic one
Bounding the search for counterexample runs

Sufficient to consider only DBs over a fixed domain of cardinality exponential in size of spec + prop

Finite search space yields decidability of verification

Periodic runs suffice: \( \exists \) counterexample iff \( \exists \) periodic one

number of underlying DBs

doubly-exponentially many DBs

doubly-exponential length in size of spec+prop

length of run
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

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

\( \mathbf{C} = \text{a set of relevant constants extracted from the spec. and prop.} \)

\[ + \]

\( \text{a fixed number of variables} \)

Input picked from \( \mathbf{C} \)

Restriction of states to constants in \( \mathbf{C} \)

Input

Output

Restriction of outputs to constants in \( \mathbf{C} \)

Restriction of DB to \( \mathbf{C} \)

Size polynomial in spec + prop
Pseudoruns
Pseudoruns
Pseudoruns
Pseudoruns

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

- Can compute next possible pseudoconfigurations from current one.
Pseudoruns

- Can compute next possible pseudoconfigurations from current one
Pseudoruns

- Can compute next possible pseudoconfigurations from current one
Pseudoruns

- Can compute next possible pseudoconfigurations from current one
Pseudoruns

- Can compute next possible pseudoconfigurations from current one
- Never construct entire DB, just “slide” poly window over it
Pseudoruns

- Can compute next possible pseudoconfigurations from current one
- Never construct entire DB, just “slide” poly window over it

← → PSPACE verification algorithm
Verification of compositions

Reduce to single peer verification

• 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

→ PTIME reduction preserving input boundedness
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 **PSPACE-complete** if an input-bounded composition with bounded, lossy channels satisfies a data-agnostic conversation protocol with observer-at-recipient semantics
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

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:

$$G \ \forall ssn \ [ \ ?getRating(ssn) \ \Rightarrow \ \left( \neg \text{rating}(ssn, \ "poor") \lor \neg \text{rating}(ssn, \ "fair") \lor \neg \text{rating}(ssn, \ "good") \right) ]$$
Verification under given environment

Theorem: It is PSPACE-complete if an input-bounded composition $C$ with bounded queues and lossy channels satisfies an input-bounded LTL-FO property $\varphi$ under a strictly-input-bounded environment specification $\psi$. 
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></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P5 (true)</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>P7 (true)</td>
<td>2</td>
</tr>
<tr>
<td>Session $Gp \mathbin{\Diamond} Gq$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P9 (true)</td>
<td>1</td>
</tr>
<tr>
<td>Correlation $Fp \mathbin{\Diamond} Fq$</td>
<td></td>
<td></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 $p \mathbin{\Diamond} Fq$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P14 (false)</td>
<td>0.19</td>
</tr>
<tr>
<td>Reachability $Gp \text{ or } Fq$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></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 $G(p \mathbin{\langle} Xp)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P6 (false)</td>
<td>0.49</td>
</tr>
<tr>
<td>Guarantee $Fp$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>P1 (true)</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>P8 (false)</td>
<td>0.11</td>
</tr>
</tbody>
</table>
### 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>Response $p \Diamond Fq$</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>Reachability $G p$ or $F q$</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 $G(p \Diamond Xp)$</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>

**Note:** Shipments are only allowed after proper payment.
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*

Implementation of WAVE: SIGMOD 2005
demo SIGMOD 2006

Verification of compositions: PODS 2006
Branching-time temporal properties
Branching-time temporal properties
Branching-time temporal properties

Current state

homepage

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)

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

(AGEF)homepage
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**
- 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

Proof technique: highly efficient model-checking technique of Kupferman, Vardi, Wolper using hesitant alternating tree automata (HAA). 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-l 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