Verification of Data-Intensive Web Services

Grigoris Karvounarakis
University of Pennsylvania

presented at: CIS 650
December 17, 2004
Data-Intensive Web Services

- Web services supported by databases
  - Large amount of data (e.g., for product catalogs)
  - Common database features (transactions, concurrency control)
  - State transitions involve database operations (e.g., queries)
  - Control flow depends on data values (results of queries)

- As services become more complex, it is important to be able to verify their correctness automatically.
  - But verification models/techniques cannot handle programs that involve “external” data in their transitions
The database approach

- Instead of trying to extend program models to handle interaction with databases, use databases to encode program specifications!
  - Express state transitions, input/output in a database (high-level, declarative) language
  - E.g., datalog-like rules
  - Adding dynamic aspect to databases, reminiscent of active databases
Relevant papers

• Several papers on this problem in database theory conferences in the last several years:
  - Relational Transducers for Electronic Commerce: Abiteboul, Vianu, Fordham, Yesha, PODS’98 & JCSS’00
  - Verification of Relational Transducers for Electronic Commerce: Spielmann, PODS’00 & JCSS’03
  - Specification and Verification of Data-driven Web Services: Deutsch, Sui, Vianu, PODS’04
Example

- Online bookstore that allows users to order books, sends them a bill for their order, receives a payment and, if the amount is correct, ships the product
Outline

• High-level Specification models
  ■ Relational Transducers
  ■ ASM Transducers
  ■ Web Page Schemas

• Automatic Verification
  ■ Past input
  ■ FO-LTL
  ■ CTL/CTL*

• Conclusions
Relational transducers

- Transducers: automata with output
  - Transition function
  - Output function
- Use datalog rules to express transition and output functions
  - RHS: Conjunction of (positive or negative) relational atoms
  - State rules: inflationary, asserted atoms are accumulated
  - Output rules: non-inflationary, at every step only newly inferred facts are retained
Example Schema

- **Database**: `price(product,price)`
- **Input**: `order(user,product), pay(user,product,price)`
- **Output**: `sendbill(user,product,price), ship(user,product)`
- **State**: `past-order(user,product)`

Color code: input state action data
Example Specification

- **State rules:**
  - \texttt{past-order(U,X) +:- order(U,X)}

- **Output rules:**
  - \texttt{sendbill(U,X,Y) :- order(U,X) \land price(X,Y)}

- **Problem:** \texttt{orders} cannot be removed from \texttt{past-order}
  - We cannot go back to a previous state, i.e., we cannot express loops
  - The rule above would \texttt{sendbill} again and again for every future step
A way to side-step the problem

- **Database**: `price(product,price)`
- **Input**: `order(user,product), pay(user,product,price)`
- **Output**: `sendbill(user,product,price), ship(user,product)`
- **State**: `past-order(user,product), past-pay(user,product,price)`
Example Specification

- **State rules:**
  - `past-order(U,X) +:- order(U,X)`
  - `past-pay(U,X,Y) +:- pay(U,X,Y)`

- **Output rules:**
  - `sendbill(U,X,Y) :- order(U,X) ∧ price(X,Y) ∧ ¬ past-pay(U,X,Y)`
  - `ship(U,X) :- past-order(U,X) ∧ price(X,Y) ∧ pay(U,X,Y) ∧ ¬ past-pay(U,X,Y)`
But there are still problems

- **State rules:**
  - \( \text{past-order}(U,X) \vdash \text{order}(U,X) \)
  - \( \text{past-pay}(U,X,Y) \vdash \text{pay}(U,X,Y) \)

- **Output rules:**
  - \( \text{sendbill}(U,X,Y) \leftarrow \text{order}(U,X) \land \text{price}(X,Y) \land \neg \text{past-pay}(U,X,Y) \)
  - \( \text{ship}(U,X) \leftarrow \text{past-order}(U,X) \land \text{price}(X,Y) \land \text{pay}(U,X,Y) \)
    \( \land \neg \text{past-pay}(U,X,Y) \)
But there are still problems

- State rules:
  - past-order(U,X) +:- order(U,X)
  - past-pay(U,X,Y) +:- pay(U,X,Y)

- Output rules:
  - sendbill(U,X,Y) :- order(U,X) \land \text{price}(X,Y) \land \neg \text{past-pay}(U,X,Y)
  - ship(U,X) :- past-order(U,X) \land \text{price}(X,Y) \land pay(U,X,Y) \land \neg \text{past-pay}(U,X,Y)

After payment has been received and entered in past-pay, a user cannot buy the same item again (sendbill, ship cannot be fired!)
Solutions?

- **Make X in** past-order(U,X)/past-pay(U,X,Y) **unique for every different item of a product**
  - Then price(X,Y) would have to contain one tuple for every item in the store inventory!

- **Add unique order id:** past-order(U,N,X)/past-pay(U,N,X,Y))
  - However, the fact that it is unique cannot be expressed in the model (and we would not be able to verify properties that depend on this fact)
    - e.g., the property that “every time an order is submitted and that book is in stock, then eventually a bill will be sent to the customer” holds for this specification under this assumption, but not without it...
Solution: ASM transducers

- Allow state rules to delete tuples from state relations
  - datalog\neg
  - Arbitrary FO on right-hand side of rules
- State rules:
  - past-order(U,X) +:- order(U,X)
  - \neg past-order(U,X) +:- past-order(U,X) \land pay(U,X,Y) \
    \land \textbf{price}(X,Y)
- Output rules:
  - sendbill(U,X,Y) :- order(U,X) \land \textbf{price}(X,Y) \
    \land \neg past-order(U,X,Y)
  - ship(U,X) :- past-order(U,X) \land \textbf{price}(X,Y) \land pay(U,X,Y)
Solution: ASM transducers

- Allow state rules to delete tuples from state relations
  - datalog
  - Arbitrary FO on right-hand side of rules
- State rules:
  - past-order(U,X) +:- order(U,X)
  - ¬past-order(U,X) +:- past-order(U,X) ∧ pay(U,X,Y) ∧ price(X,Y)
- Output rules:
  - sendbill(U,X,Y) :- order(U,X) ∧ price(X,Y) ∧ ¬past-order(U,X,Y)
  - ship(U,X) :- past-order(U,X) ∧ price(X,Y) ∧ pay(U,X,Y)

Remove order after payment is received and go back to state before the order was received.
Web Page Schemas

- Observation: web services are usually
  - Sequence of web pages
  - User input is usually picked from lists of choices compiled by the system (based on information in the database).

- Adjust ASM transducer model to handle this:
  - One ASM transducer per page
  - Rules for page transitions
  - Rules to pre-populate input options
Page transitions

HOMEPAGE ($W_0$)
- Welcome to Bookstore.com
- Find books

ORDER PAGE
- Book 1
- Book 2
- Book 3
- order
- cancel

PAYMENT PAGE
- Your bill is: $xx.xx
- Enter payment amount:
- pay

order(X) \land button(“order”) \land \neg past-order(U,X)
WP schema of example

- **Order Page**
  - Input: `order(X), button(X)`
  - Input Rules:
    - \( \text{Options}_{\text{order}}(X) \leftarrow \text{price}(X,Y) \)
    - \( \text{Options}_{\text{button}}(X) \leftarrow X="\text{order}" \lor X="\text{cancel}" \)
  - State Rules (this example: one state relation error):
    - \( \text{past\_order}(U,X) \leftarrow \text{order}(X) \land U = \text{user} \land \neg \text{past\_order}(U,X) \)
  - Target WP: PP, HP
  - Target Rules:
    - PP \leftarrow \text{order}(X) \land \text{button}("order") \land \neg \text{past\_order}(U,X)
    - HP \leftarrow \text{button}("cancel")
  - Output rules:
    - \( \text{send\_bill}(U,X,Y) \leftarrow \text{order}(U,X) \land U = \text{user} \land \text{button}("order") \land \text{price}(X,Y) \land \neg \text{past\_order}(U,X) \)
WP schema of example (cont’d)

• Payment Page
  □ Input: pay(U,X,Y), button(X)
  □ Input Rules:
    • Options\_order(X) ← price(X,Y)
    • Options\_button(X) ← X="order" ∨ X="cancel"
  □ State Rules (this example: one state relation error):
    • past\_order(U,X) ↑:- order(X) ∧ U = user ∧ ¬ past\_order(U,X)
  □ Target WP: PP, HP
  □ Target Rules:
    • HP :- button("pay")
  □ Output rules:
    • ship(U,X,Y) :- past\_order(U,X) ∧ pay(U,X,Y) ∧ button("pay")
      ∧ price(X,Y) ∧ ∃ Z (credit-limit(U,Z) ∧ Z ≥ Y)
Outline

- High-level Specification models
  - Relational Transducers
  - ASM Transducers
  - Web Page Schemas

- Automatic Verification
  - Past input
  - FO-LTL
  - CTL/CTL*

- Conclusions
Verifying properties about past input

• Properties of the form:
  \( \forall x [\varphi(state,db,in)(x) \rightarrow \psi(state,db,in)(x)] \)

• E.g., “Whenever a product is shipped, at some previous step a proper payment has been received”

• Such properties can be verified on Spocus transducers (in NEXPTIME)
  - Semi-positive output: every variable appearing in an output rule must occur in at least one positive atom
  - Cumulative state: state rules can only accumulate previous input
Spocus transducers

- Cumulative state is a bit too restrictive
  - Cannot express the following simple machine:

```
(a) → b → c
   \     \    
  b   a  c
```

because state can only contain past input, and relations are sets (unordered)
Some properties require more expressiveness

- Cannot express properties like:
  “if a customer pays for a book, then eventually the book will be shipped to them”
First-Order Linear-time Temporal Logic

- Temporal operators
  - $\Box p$: $p$ holds in the next time
  - $p U q$: $p$ holds until $q$ holds
  - $F p$: $p$ holds eventually
  - $G p$: $p$ always holds
  - $p B q$: either $q$ always holds or $p$ holds before $q$ fails

- Examples:
  - If a customer pays for a book, then eventually the book will be shipped to them:
    \[ \forall u \forall x \forall y [ \text{past\_order}(u,x) \land \text{price}(x,y) \land \text{pay}(u,y) \rightarrow F \text{ship}(u,x)] \]
  - A product must be paid for before it is shipped:
    \[ \forall u \forall y [ \text{pay}(u,y) \land \exists x \text{price}(x,y) \land \neg \text{ship}(x,y)] \]
LTL-FO Example

- Verification is undecidable for ASM transducers:
  - Restrictions:
    - Maximal input flow
    - Input-boundedness:
      - Input rules: $\exists^*\text{FO}$ with ground state atoms
      - All other rules: quantification only on variables that are bounded by expressions on (current or last but not further “past”) input relations

- Given an input bounded Web Service $W$ and an input bounded LTL-FO formula $\varphi$, checking $W \models \varphi$ is in EXPSPACE (PSPACE-complete for fixed schema)
Input-boundedness

- Not input-bounded formula:
  \[ \forall u \forall y [ \text{pay}(u,y) \land \exists x \text{ price}(x,y) \land \neg \text{ship}(x,y)] \]

- Input-bounded formula:
  \[ \forall u \forall y [ \text{last\_order}(u,x) \land \text{pay}(u,y) \land \exists x \text{ price}(x,y) \land \neg \text{ship}(x,y)] \]
Boundaries of decidability

- Relaxing the requirement that state atoms must be ground in formula defining the input options, by allowing state atoms with variables.
  
  **Reduction:** Does TM halt on input epsilon?

- Lifting the input-bounded requirement by allowing state projection (i.e., state rules of the form: \( S(x) :\exists y \ S'(x,y) \)).
  
  **Reduction:** Implication for FDs and IDs

- Allowing variables to be bounded on state relations that hold all past input input (rather than the most recent one).
  
  **Reduction:** Trakhtenbrot’s Theorem

- Extend the LTL-FO formula with path quantification.
  
  **Reduction:** validity of \( \exists^* \forall^* \) FO formulas
Branching-time Temporal Logic

• CTL (Computation Tree Logic) introduces path quantifiers:
  - $A$: “for every path”
  - $E$: “there exists a path”

• Proposed for Web page schemas (to express properties of transition between pages)

• Example
  - From every page (i.e., the target of every path) there always exists a path that eventually reaches HP:
    $A \forall \exists F HP$
Verification of CTL-FO

- Input-boundedness is not enough for decidability of CTL-FO verification
- Further restriction: Propositional input-bounded Web Services
  - Actions and states are propositional (no data values)
  - Input rules need not be propositional
- Propositional CTL formulas
  - Use input, action, state relation names and WP symbols (viewed as propositions)
Input-bounded propositional CTL-FO

- Verification of $W \models \varphi$ for CTL(*) formulas for propositional Web services:
  - \textit{CO-NEXPTIME} for CTL
  - \textit{EXPSPACE} for CTL*

- But propositional Web services are not very expressive
  - Can be viewed as an abstraction
  - Can express property in previous slide
  - Cannot express navigational properties that depend on data, e.g.:
    $$\forall \text{pid}, \forall \text{pr} \ [A(G(\xi(\text{pid},\text{pr}) \rightarrow A((EF \text{ cancel}(\text{user},\text{pid}))U(\text{ship}(\text{user},\text{pid}))))]$$
Outline

- High-level Specification models
  - Relational Transducers
  - ASM Transducers
  - Web Page Schemas
- Automatic Verification
  - Past input
  - FO-LTL
  - CTL/CTL*
- Conclusions
Conclusions

- Spocus transducers are probably too restrictive
- ASM transducers more appropriate for practical use
- Interesting properties require temporal operators
  - Decidable for input-bounded specifications/formulas
- Navigational properties require further restrictions
  - Trade-off between more expressiveness of model and of property language
  - Questionable whether the model for which verification is decidable is expressive enough to represent interesting applications
Conclusions

- Static analysis is decidable under restrictions, but we need practical verification algorithms to build applications
  - Heuristics could help in achieving good performance
  - Attempt to implement real-life applications could uncover more limitations of the models
  - Performance of algorithms for such applications will determine whether this work can be widely used in practice
Extensions

- Allow updates in database relations
  - During runs, or
  - Sessions: verify properties within a session and allow updates between sessions

- Interacting Web services
  - Extend models with communication primitives to model web service composition
  - Discover important properties for composite web services
  - Devise appropriate restrictions under which verification of those properties is decidable