Specification and Verification of Data-driven Web Services

Alin Deutsch Liying Sui* Victor Vianu*

University of California San Diego
Computer Science and Engineering
9500 Gilman Drive
La Jolla, CA 92093, U.S.A.
{deutsch, lsui, vianu}@cs.ucsd.edu

ABSTRACT

We study data-driven Web services provided by Web sites interacting with users or applications. The Web site can access an underlying database, as well as state information updated as the interaction progresses, and receives user input. The structure and contents of Web pages, as well as the actions to be taken, are determined dynamically by querying the underlying database as well as the state and inputs. The properties to be verified concern the sequences of events (inputs, states, and actions) resulting from the interaction, and are expressed in linear or branching-time temporal logics. The results establish under what conditions automatic verification of such properties is possible and provide the complexity of verification. This brings into play a mix of techniques from logic and automatic verification.

1. INTRODUCTION

Web services, viewed broadly as interactive Web applications providing access to information as well as transactions, are typically powered by databases. They have a strong dynamic component and are governed by protocols of interaction with users or programs, ranging from the low-level input-output signatures used in WSDL [31], to high-level workflow specifications (e.g., see [8, 6, 11, 30, 32, 19]). One central issue is to develop static analysis techniques to increase confidence in the robustness and correctness of complex Web services. This paper presents new verification techniques for Web services, and investigates the trade-off between the expressiveness of the Web service specification language and the feasibility of verification tasks.

In the scenario we consider, a Web service is provided by an interactive Web site that posts data, takes input from the user, and responds to the input by posting more data and/or taking some actions. The Web site can access an underlying database, as well as state information updated as the interaction progresses. The structure of the Web page the user sees at any given point is described by a Web page schema. The contents of a Web page is determined dynamically by querying the underlying database as well as the state. The actions taken by the Web site, and transitions from one Web page to another, are determined by the input, state, and database.

The properties we wish to verify about Web services involve the sequences of inputs, actions, and states that may result from interactions with a user. This covers a wide variety of useful properties. For example, in a Web service supporting an e-commerce application, it may be desirable to verify that no product is delivered before payment of the right amount is received. Or, we may wish to verify that the specification of Web page transitions is unambiguous, (the next Web page is uniquely defined at each point in a run), that each Web page is reachable from the home page, etc. To express such properties, we rely on temporal logic. Specifically, we consider two kinds of properties. The first requires that all runs must satisfy some condition on the sequence of inputs, actions, and states. To describe such properties we use a variant of linear-time temporal logic. Other properties involve several runs simultaneously. For instance, we may wish to check that for every run leading to some Web page, there exists a way to return to the home page. To capture such properties, we use variants of the branching-time logics CTL and CTL*.

Our results identify classes of Web services for which temporal properties in the above temporal logics can be checked, and establish their complexity. As justification for the choice of these classes, we show that even slight relaxations of our restrictions lead to undecidability of verification. Thus, our decidability results are quite tight.

Related work Our notion of Web service is a fairly broad one. It encompasses a large class of data-intensive Web applications equipped (implicitly or explicitly) with workflows that regulate the interaction between different partners who can be users, WSDL-style Web services, Web sites, programs and databases. We address the verification of properties pertaining to the runs of these workflows. Our model is related to WebML [8], a high-level conceptual model for data-driven Web applications, extended in [7] with workflow concepts to

*Supported in part by the NSF under grant number ITR-0225676 (SEEK).

PODS 2004 June 14-16, 2004, Paris, France.

support process modeling. It is also related to other high-level workflow models geared towards Web applications (e.g. [6, 11, 32]), and ultimately to general workflows (see [30, 17, 18, 12, 4, 29]), whose focus is however quite different from ours. Non-interactive variants of Web page schemas have been proposed in prior projects such as Strudel [14], Araneus [23] and Weave [15], which target the automatic generation of Web sites from an underlying database.

More broadly, our research is related to the general area of automatic verification, and particularly reactive systems [22, 21]. Directly relevant to us is Spielmann's work on Abstract State Machine (ASM) transducers [26, 28], and earlier variations of this work [27]. Similarly to the earlier relational transducers [3] these model database-driven reactive systems that respond to input events by taking some action, and maintain state information in designated relations. Our model of Web services is considerably more complex than ASM transducers. The techniques developed in [26, 28] remain nonetheless relevant, and we build upon them to obtain our decidability results on the verification of linear-time properties of Web services (by reducing the verification problem to finite satisfiability of existential first-order logic augmented with a transitive closure operator (denoted E+TC). For the results on branching-time properties we use a mix of techniques from finite-model theory and temporal logic (see [13]), as well as automata-theoretic model-checking techniques developed by Kupferman, Vardi, and Wolper [20].

Other work on formal verification of Web services was done by Narayanan and McIlraith [24]. They consider Web services described in a fragment of the DAML-S ontology [11]. DAML-S specifications are formalized using situation calculus [25] and given an operational semantics using 1-safe Petri nets. The results concern the complexity of verification tasks expressed as reachability problems in the Petri nets. While differences in the models and formalisms render a comparison difficult, it appears that the approach of [24] would allow, in our framework, the verification of a fragment of linear-time temporal logic properties of runs of Web services on a *fixed* database. In contrast, our results do not assume a fixed database and apply to an orthogonal class of linear-time temporal properties.

More broadly, note that classical finite-state verification techniques, including model checking, are not directly applicable to our framework, since Web services are not finite-state systems. This is due to the fact that we consider runs on arbitrary databases, whose domains are not statically bounded.

The verification of *e-compositions* of collections of individual web service peers is discussed in [19]. E-compositions are specified using the standard BPEL4WS [10]. A run of an e-composition is a sequence of message types passed between peers during a valid execution. It is observed in [19] that the set of all runs can be modeled as the language generated by a Mealy machine. This in turn enables PTIME verifiability of LTL properties of an e-composition. One of the open problems listed in [19] is the verification of properties that pertain not just to the type of the messages but also to the data, as done in our framework.

The paper is organized as follows. Section 2 introduces our model and specification language for Web sites. Section 3 considers the verification of linear-time temporal properties, and Section 4 focuses on branching-time properties. The main body of the paper is self contained. Due to space limitations, it emphasizes informal presentation of the results and extensive examples intended to facilitate understanding of the formal framework. A demo Web site implementing our running example is available at `http://www.cs.ucsd.edu/~lsui/project/index.html`.

2. WEB SERVICE SPECIFICATIONS

In this section we provide our model and specification language for data-driven Web services. Our model of a Web service captures the interaction of an external user[1] with the Web site, referred to as a "run". Informally, a Web service specification has the following components:

- A database that remains fixed throughout each run;
- A set of state relations that change throughout the run in response to user inputs;
- A set of Web page schemas, of which one is designated as the "home page", and another as an "error page";
- Each Web page schema defines how the set of current input choices is generated as a query on the database and states. In addition, it specifies the state transition in response to the user's input, the actions to be taken, as well as the next Web page schema.

Intuitively, a run proceeds as follows. First, the user accesses the home page, and the state relations are initialized to empty. When accessed, each Web page generates a choice of inputs for the user, by a query on the database and states. All input options are generated by the system except for a fixed set that represents specific user information (e.g. name, password, credit card number, etc). These are represented as constants in the input schema, whose interpretations are provided by the user throughout the run as requested. The user chooses at most one tuple among the options provided for each input. In response to this choice, a state transition occurs, actions are taken, and the next Web page schema is determined, all according to the rules of the specification. As customary in verification, we assume that all runs are infinite (finite runs can be easily represented as infinite runs by fake loops).

We now formalize the above notion of Web service. We assume fixed an infinite set of elements $\mathbf{dom}_\infty$. A *relational schema* is a finite set of relation symbols with associated arities, together with a finite set of constant symbols. Relation symbols with arity zero are also called propositions. A relational instance over a relational schema consists of a finite subset *Dom* of $\mathbf{dom}_\infty$, and a mapping associating to each relation symbol of positive arity a finite relation of the same arity, to each propositional symbol a truth value, and to each constant symbol an element of *Dom*. We use several kinds of relational schemas, with different roles in the specification of the Web service.

[1]We use the term "user" generically to stand for any partner interacting with the Web site, be it an actual user, program, another Web service, etc.

We assume familiarity with first-order logic (FO) over relational vocabularies. We adopt here an active domain semantics for FO formulas, as commonly done in database theory (e.g., see [2]).

DEFINITION 2.1. *A* Web service $\mathcal{W}$ *is a tuple* $\langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, W_0, W_\epsilon \rangle$*, where:*

- **D, S, I, A** *are relational schemas called database, state, input, and action schemas, respectively. The sets of relation symbols of the schemas are disjoint (but they may share constant symbols). We refer to constants in* $\mathbf{I}$ *as* input constants*, and denote them by* $\mathbf{const(I)}$.
- $\mathbf{W}$ *is a finite set of Web page schemas.*
- $W_0 \in \mathbf{W}$ *is the* home page *schema, and* $W_\epsilon \notin \mathbf{W}$ *is the* error page *schema.*

We also denote by $\mathbf{Prev_I}$ *the relational vocabulary* $\{prev_I \mid I \in \mathbf{I} - \mathbf{const(I)}\}$*, where* $prev_I$ *has the same arity as* I *(intuitively,* $prev_I$ *refers to the input* I *at the previous step in the run).*

A Web page schema W *is a tuple* $\langle \mathbf{I}_W, \mathbf{A}_W, \mathbf{T}_W, \mathcal{R}_W \rangle$ *where* $\mathbf{I}_W \subseteq \mathbf{I}, \mathbf{A}_W \subseteq \mathbf{A}$*,* $\mathbf{T}_W \subseteq \mathbf{W}$*. Then* $\mathcal{R}_W$ *is a set of rules containing the following:*

- *For each input relation* $I \in \mathbf{I}_W$ *of arity* $k > 0$*, an* input rule $\text{Options}_I(\bar{x}) \leftarrow \varphi_{I,W}(\bar{x})$ *where* Options_I *is a relation of arity* k*,* $\bar{x}$ *is a* k*-tuple of distinct variables, and* $\varphi_{I,W}(\bar{x})$ *is an FO formula over schema* $\mathbf{D} \cup \mathbf{S} \cup \mathbf{Prev_I} \cup \mathbf{const(I)}$*, with free variables* $\bar{x}$*.*
- *For each state relation* $S \in \mathbf{S}$*, one, both, or none of the following* state rules*:*
  - *an insertion rule* $S(\bar{x}) \leftarrow \varphi^+_{S,W}(\bar{x})$*,*
  - *a deletion rule* $\neg S(\bar{x}) \leftarrow \varphi^-_{S,W}(\bar{x})$*,*

  where the arity of S *is* k*,* $\bar{x}$ *is a* k*-tuple of distinct variables, and* $\varphi^\epsilon_{S,W}(\bar{x})$ *are FO formulas over schema* $\mathbf{D} \cup \mathbf{S} \cup \mathbf{Prev_I} \cup \mathbf{const(I)} \cup \mathbf{I}_W$*, with free variables* $\bar{x}$*.*
- *For each action relation* $A \in \mathbf{A}_W$*, an* action rule $A(\bar{x}) \leftarrow \varphi(\bar{x})$ *where the arity of* A *is* k*,* $\bar{x}$ *is a* k*-tuple of distinct variables, and* $\varphi(\bar{x})$ *is an FO formula over schema* $\mathbf{D} \cup \mathbf{S} \cup \mathbf{Prev_I} \cup \mathbf{const(I)} \cup \mathbf{I}_W$*, with free variables* $\bar{x}$*.*
- *for each* $V \in \mathbf{T}_W$*, a* target rule $V \leftarrow \varphi_{V,W}$ *where* $\varphi_{V,W}$ *is an FO sentence over schema* $\mathbf{D} \cup \mathbf{S} \cup \mathbf{Prev_I} \cup \mathbf{const(I)} \cup \mathbf{I}_W$*.*

Finally, $W_\epsilon = \langle \emptyset, \emptyset, \{W_\epsilon\}, \mathcal{R}_{W_\epsilon} \rangle$ *where* $\mathcal{R}_{W_\epsilon}$ *consists of the rule* $W_\epsilon \leftarrow true$*.*

Intuitively, the action rules of a Web page specify the actions to be taken in response to the input. The state rules specify the tuples to be inserted or deleted from state relations (with conflicts given no-op semantics, see below). If no rule is specified in a Web page schema for a given state relation, the state remains unchanged. The input rules specify a set of options to be presented to users, from which they can pick at most one tuple to input (this feature corresponds to menus in user interfaces). At every point in time, I contains the current input tuple, and $prev_I$ contains the input to I in the previous step of the run (if any). The choice of this semantics for $prev_I$ relations is somewhat arbitrary, and other choices are possible without affecting the results. For example, another possibility is to have $prev_I$ hold the *most recent* input to I occurring anywhere in the run, rather than in the previous step. Also note that $prev_I$ relations are really state relations with very specific functionality, and are redundant in the general model. However, they are very useful when defining tractable restricitons of the model.

Notation For better readability of our examples, we use the following notation: relation R is displayed as R if it is a state relation, as R if it is an input relation, as $\underline{\mathsf{R}}$ if it is a database relation, and as $\overline{\mathsf{R}}$ if it is an action relation. In Example 2.2 below, error $\in \mathbf{S}$, $\underline{\mathsf{user}} \in \mathbf{D}$ and $\mathsf{name}, \mathsf{password}, \mathsf{button} \in \mathbf{I}$.

Example 2.2 We use as a running example throughout the paper an e-commerce Web site selling computers online. New customers can register a name and password, while returning customers can login, search for computers fulfilling certain criteria, add the results to a shopping cart, and finally buy the items in the shopping cart. A demo Web site implementing this example, together with its full specification, is provided at http://www.cs.ucsd.edu/~lsui/project/index.html. We only list here a subset of the pages in the demo that are used in the running example:

HP	the home page
RP	the new user registration page
CP	the customer page
AP	the administrator page
LSP	a laptop search page
PIP	displays the products returned by the search
CC	allows the user to view the cart contents and order items in it
MP	an error message page

The following describes the home page HP which contains two text input boxes for the customer's user name and password respectively, and three buttons, allowing customers to register, login, respectively clear the input.

`Page` HP
`Inputs` $\mathbf{I}_{HP}$: $\mathsf{name}, \mathsf{password}, \mathsf{button}(x)$
`Input Rules:`

$$\text{Options}_{\mathsf{button}}(x) \leftarrow x = \text{“}login\text{”} \vee x = \text{“}register\text{”} \vee x = \text{“}clear\text{”}$$

`State Rules:`

$$\text{error}(\text{“}failed\ login\text{”}) \leftarrow \neg\underline{\mathsf{user}}(\mathsf{name}, \mathsf{password}) \wedge \mathsf{button}(\text{“}login\text{”})$$

Target Web Pages $\mathbf{T}_{HP}$: HP, RP, CP, AP, MP
Target Rules:
HP ← button("*clear*")
RP ← button("*register*")
CP ← user(name, password) ∧ button("*login*") ∧name ≠ "*Admin*"
AP ← user(name, password) ∧ button("*login*") ∧name = "*Admin*"
MP ← ¬user(name, password) ∧ button("*login*")
End Page HP

Notice how the three buttons are modeled by a single input relation button, whose argument specifies the clicked button. The corresponding input rule restricts it to a login, clear or register button only. As will be seen shortly (Definition 2.3), each input relation may contain at most one tuple at any given time, corresponding to the user's pick from the set of tuples defined by the associated input rule. This guarantees that no two buttons may be clicked simultaneously. The user name and password are modeled as input constants, as their value is not supposed to change during the session. If the login button is clicked, the first state rule looks up the name/password combination in the database table user. If the lookup fails, the second state rule records the login failure in the state relation error, and the last target rule fires a transition to the message page MP. Notice how the "*Admin*" user enjoys special treatment: upon login, she is directed to the admin page AP, whereas all other users go to the customer page CP. Assume that the CP page allows users to follow either a link to a desktop search page, or a laptop search page LSP. We illustrate only the laptop search functionality of the search page LSP (see the online demo for the full version, which also allows users to search for desktops).

Page LSP
Inputs $\mathbf{I}_{SP}$: laptopsearch$(ram, hdisk, display)$, button(x)
Input Rules:
Options$_{\text{button}}(x)$ ← $x =$ "*search*" ∨ $x =$ "*view cart*" ∨$x =$ "*logout*"
Options$_{\text{laptopsearch}}(r, h, d)$ ← criteria("*laptop*", "*ram*", r) ∧criteria("*laptop*", "*hdd*", h) ∧ criteria("*laptop*", "*display*", d)
State Rules:
userchoice(r,h,d) ← laptopsearch(r, h, d) ∧ button("*search*")
Target Web Pages $\mathbf{T}_{SP}$: HP, PIP, CC
Target Rules:
HP ← button("*logout*")
PIP ← $\exists r \exists h \exists d$ laptopsearch(r, h, d) ∧ button("*search*")
CC ← button("*view cart*")
End Page SP

Notice how the second input rule looks up in the database the valid parameter values for the search criteria pertinent to laptops. This enables users to pick from a menu of legal values instead of providing arbitrary ones. □

We next define the notion of "run" of a Web service. Essentially, a run specifies the fixed database and consecutive Web pages, states, inputs, and actions. Thus, a run over database instance D is an infinite sequence $\{\langle V_i, S_i, I_i, P_i, A_i\rangle\}_{i\geq 0}$, where $V_i \in \mathbf{W}, S_i$ is an instance of $\mathbf{S}$, I_i is an instance of $\mathbf{I}_{V_i}$, P_i is an instance of $\mathbf{prev_I}$, and A_i is an instance of $\mathbf{A}_{V_i}$. In particular, the input constants play a special role. Their interpretation is not fixed a priori, but is instead provided by the user as the run progresses. We will need to make sure this occurs in a sound fashion. For example, a formula may not use an input constant before its value has been provided. We will also prevent the Web service from asking the user repeatedly for the value of the same input constant. To formalize this, we will use the following notation. For each $i \geq 0$, κ_i denotes the set of input constants occurring in some $\mathbf{I}_{V_j}$ in the run, $j \leq i$, and σ_i denotes the mapping associating to each $c \in \kappa_i$ the unique $I_j(c)$ where $j \leq i$ and $c \in \mathbf{I}_{V_j}$.

DEFINITION 2.3. *Let $\mathcal{W} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, W_0, W_\epsilon\rangle$ be a Web service and D a database instance over schema $\mathbf{D}$. A* run *of $\mathcal{W}$ for database D is an infinite sequence $\{\langle V_i, S_i, I_i, P_i, A_i\rangle\}_{i\geq 0}$ where $V_i \in \mathbf{W}, S_i$ is an instance of $\mathbf{S}$, I_i is an instance of $\mathbf{I}_{V_i}$, P_i is an instance of $\mathbf{prev_I}$, A_i is an instance of $\mathbf{A}_{V_i}$ and:*

- *$V_0 = W_0$, and S_0, A_0, P_0 are empty;*
- *for each $i \geq 0$, $V_{i+1} = W_\epsilon$ if one of the following holds:*
  - *(i) some formula used in a rule of V_i involves a constant $c \in \mathbf{I}$ that is not in κ_i;*
  - *(ii) $\mathbf{I}_{V_i} \cap \kappa_{i-1} \neq \emptyset$;*
  - *(iii) there are distinct $W, W' \in \mathbf{T}_{V_i}$ for which φ_{W,V_i} and φ_{W',V_i} are both true when evaluated on D, S_i, I_i and P_i, and interpretation σ_i for the input constants occurring in the formulas;*

  Otherwise, V_{i+1} is the unique $W \in \mathbf{T}_{V_i}$ for which φ_{W,V_i} is true when evaluated on D, S_i, I_i, P_i and σ_i if such W exists; if not, $V_{i+1} = V_i$.
- *for each $i \geq 0$, and for each relation R in $\mathbf{I}_{V_i}$ of arity $k > 0$, $I_i(R) \subseteq \{v\}$ for some $v \in Options_R$, where $Options_R$ is the result of evaluating φ_{R,V_i} on D, S_i, P_i and σ_i;*
- *for each $i \geq 0$, and for each proposition R in $\mathbf{I}_{V_i}$, $I_i(R)$ is a truth value;*
- *for each $i \geq 0$, and for each constant c in $\mathbf{I}_{V_i}$, $I_i(c)$ is an element in $\mathbf{dom}_\infty$;*
- *for each $i \geq 0$, and for each relation $prev_I$ in $\mathbf{prev_I}$, $P_i(prev_I) = I_{i-1}(I)$ if $I \in \mathbf{I}_{V_{i-1}}$ and $P_i(prev_I)$ is empty otherwise.*
- *for each $i \geq 0$, and relation S in $\mathbf{S}$, $S_{i+1}(S)$ is the result of evaluating*

$$(\varphi^+_{S,V_i}(\bar{x}) \wedge \neg\varphi^-_{S,V_i}(\bar{x})) \vee$$
$$(S(\bar{x}) \wedge \varphi^-_{S,V_i}(\bar{x}) \wedge \varphi^+_{S,V_i}(\bar{x})) \vee$$
$$(S(\bar{x}) \wedge \neg\varphi^-_{S,V_i}(\bar{x}) \wedge \neg\varphi^+_{S,V_i}(\bar{x}))$$

  on D, S_i, I_i, P_i and σ_i, where $\varphi^\epsilon_{S,V_i}(\bar{x})$ is taken to be false *if it is not provided in the Web page schema ($\epsilon \in \{+,-\}$). In particular, S remains unchanged if no insertion or deletion rule is specified for it.*

- *for each $i \geq 0$, and relation A in $\mathbf{A}_{V_{i+1}}$, $A_{i+1}(A)$ is the result of evaluating φ_{A,V_i} on D, S_i, I_i, P_i and σ_i.*

Note that the state and actions specified at step $i+1$ in the run are those triggered at step i. This choice is convenient for technical reasons. As discussed above, input constants are provided an interpretation as a result of user input, and need not be values already existing in the database. Once an interpretation is provided for a constant, it can be used in the formulas determining the run. For example, such constants might include **name**, **password**, **credit-card**, etc. The error Web page serves an important function, since it signals behavior that we consider anomalous. Specifically, the error Web page is reached in the following situations: (i) the value of an input constant is required by some formula before it was provided by the user; (ii) the user is asked to provide a value for the same input constant more than once; and, (iii) the specification of the next Web page is ambiguous, since it produces more than one Web page. Once the error page is reached, the run loops forever in that page. We call a run *error free* if the error Web page is not reached, and we call a Web service error-free if it generates only error-free runs. Clearly, it would be desirable to verify that a given Web service is error-free. As we will see, this can be expressed in the temporal logics we consider and can be checked for restricted classes of Web services.

3. VERIFYING LINEAR-TIME TEMPORAL PROPERTIES

In this section we consider the verification of properties that must be satisfied by *all* runs of a Web service. Such properties are expressed using a variant of linear-time temporal logic, adapted from [13, 1, 28]. We begin by defining this logic, that we denote LTL-FO (first-order linear temporal logic).

DEFINITION 3.1. *[13, 1, 28] The language LTL-FO (first-order linear-time temporal logic) is obtained by closing FO under negation, disjunction, and the following formula formation rule: If φ and ψ are formulas, then $\mathbf{X}\varphi$ and $\varphi\mathbf{U}\psi$ are formulas. Free and bound variables are defined in the obvious way. The* universal closure *of an LTL-FO formula $\varphi(\bar{x})$ with free variables $\bar{x}$ is the formula $\forall\bar{x}\varphi(\bar{x})$. An LTL-FO sentence is the universal closure of an LTL-FO formula.*

Note that quantifiers cannot be applied to formulas containing temporal operators, except by taking the universal closure of the entire formula, yielding an LTL-FO sentence.

Let $\mathcal{W} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, W_0, W_\epsilon \rangle$ be a Web service. To express properties of runs of $\mathcal{W}$, we use LTL-FO sentences over schema $\mathbf{D} \cup \mathbf{S} \cup \mathbf{I} \cup \mathbf{Prev_I} \cup \mathbf{A} \cup \mathbf{W}$, where each $W \in \mathbf{W}$ is used as a propositional variable. The semantics of LTL-FO formulas is standard, and we describe it informally. Let $\forall\bar{x}\varphi(\bar{x})$ be an LTL-FO sentence over the above schema. The Web service $\mathcal{W}$ satisfies $\forall\bar{x}\varphi(\bar{x})$ iff every run of $\mathcal{W}$ satisfies it. Let $\rho = \{\alpha_i\}_{i\geq 0}$ be a run of $\mathcal{W}$ for database D, and let ρ_j denote $\{\alpha_i\}_{i\geq j}$, for $j \geq 0$. Note that $\rho = \rho_0$. Let $\mathrm{Dom}(\rho)$ be the active domain of ρ, i.e. the set of all elements occurring in relations or as constants in ρ. The run ρ satisfies $\forall\bar{x}\varphi(\bar{x})$ iff for each valuation ν of $\bar{x}$ in $\mathrm{Dom}(\rho)$, ρ_0 satisfies $\varphi(\nu(\bar{x}))$. The latter is defined by structural induction on the formula. An FO sentence ψ is satisfied by $\alpha_i = \langle V_i, S_i, I_i, P_i, A_i \rangle$ if the following hold:

- the set of input constants occurring in ψ is included in κ_i;
- the structure α_i' satisfies ψ, where α_i' is the structure obtained by augmenting α_i with interpretation σ_i for the input constants. Furthermore, α_i assigns *true* to V_i and *false* to all other propositional symbols in $\mathbf{W}$.

The semantics of Boolean operators is the obvious one. The meaning of the temporal operators **X, U** is the following (where $\models$ denotes satisfaction and $j \geq 0$):

- $\rho_j \models \mathbf{X}\varphi$ iff $\rho_{j+1} \models \varphi$,
- $\rho_j \models \varphi\mathbf{U}\psi$ iff $\exists k \geq j$ such that $\rho_k \models \psi$ and $\rho_l \models \varphi$ for $j \leq l < k$.

Observe that the above temporal operators can simulate all commonly used operators, including **B** (before), **G** (always) and **F** (eventually). Indeed, $\varphi\mathbf{B}\psi$ is equivalent to $\neg(\neg\varphi\mathbf{U}\neg\psi)$, $\mathbf{G}\varphi \equiv \mathit{false}\ \mathbf{B}\ \varphi$, and $\mathbf{F}\varphi \equiv \mathit{true}\ \mathbf{U}\ \varphi$. We use the above operators as shorthand in LTL-FO formulas whenever convenient.

LTL-FO sentences can express many interesting properties of a Web service. A useful class of properties pertains to the navigation between pages.

Example 3.2 The following property states that whenever page P is reached in a run, page Q will be eventually reached as well:

$$\mathbf{G}(\neg\ \mathrm{P}) \vee \mathbf{F}(\ \mathrm{P} \wedge \mathbf{F}\ \mathrm{Q}) \tag{1}$$

□

Another important class of properties describes the flow of the interaction between user and service.

Example 3.3 Assume that the Web service in Example 2.2 allows the user to pick a product and records the pick in a state relation pick($product\_id, price$). There is also a payment page PP, with input relation $\mathsf{pay}(amount)$ and "*authorize payment*" button. Clicking this button authorizes the payment of *amount* for the product with identifier recorded in state pick, on behalf of the user whose name was provided by the constant name (recall page HP from Example 2.2). Also assume the existence of an order confirmation page OCP, containing the actions $\overline{\mathsf{conf}}(user\_id, price)$ and $\overline{\mathsf{ship}}(user\_id, product\_id)$. The following property involving state, action, input and database relations requires that any shipped product be previously paid for:

$$\forall pid, price\ [\xi(pid, price)\ \mathbf{B}\ \neg(\overline{\mathsf{conf}}(\mathsf{name}, price) \wedge \overline{\mathsf{ship}}(\mathsf{name}, pid))\ ] \tag{2}$$

where $\xi(pid, price)$ is the formula

$$\text{PP} \wedge \mathsf{pay}(price) \wedge \mathsf{button}(\text{“}authorize\ payment\text{”}) \wedge \text{pick}(pid, price) \wedge \exists pname\ \underline{\mathsf{catalog}}(pid, price, pname) \quad (3)$$

□

We consider the following verification scenario. Given a Web service, we would first like to verify, as a minimum soundness check, that it is error free. If it is, we may wish to verify some additional temporal properties expressed by LTL-FO sentences.

It is easily seen that it is undecidable if a Web service is error free, or if it satisfies a LTL-FO formula, using Trakhtenbrot's theorem. To obtain decidability, we must restrict both the Web services and the LTL-FO sentences. We use a restriction proposed in [26, 28] for ASM transducers, limiting the use of quantification in state, action, and target rule formulas to "input-bounded" quantification, and limiting formulas of input rules to be existential. The restriction is formulated in our framework as follows. Let

$$\mathcal{W} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, W_0, W_\epsilon \rangle$$

be a Web service. The set of *input-bounded* FO formulas over the schema $\mathbf{D} \cup \mathbf{S} \cup \mathbf{I} \cup \mathbf{A} \cup \mathbf{W} \cup \mathbf{Prev_I}$ is obtained by replacing in the definition of FO the quantification formation rule by the following:

- if φ is a formula, α is a current or previous input atom using a relational symbol from $\mathbf{I} \cup \mathbf{Prev_I}$, $\bar{x} \subseteq free(\alpha)$, and $\bar{x} \cap free(\beta) = \emptyset$ for every state or action atom β in φ, then $\exists \bar{x}(\alpha \wedge \varphi)$ and $\forall \bar{x}(\alpha \rightarrow \varphi)$ are formulas.

A Web service is input-bounded iff all formulas in state, action, and target rules are input bounded, and all input rules use $\exists^*$FO formulas in which all state atoms are ground. An LTL-FO sentence over the schema of $\mathcal{W}$ is input-bounded iff all of its FO subformulas are input-bounded.

Example 3.4 All rules on pages HP,SP in Example 2.2 are input-bounded. Property (1) in Example 3.2 is trivially input-bounded, as it contains no quantifiers. Property (2) in Example 3.3, however, is not input-bounded because *pname* appears in no input atom. We turn this into an input-bounded property by modeling the $\underline{\mathsf{catalog}}$ database relation with two relations $\underline{\mathsf{prod\_prices}}(pid, price)$ and $\underline{\mathsf{prod\_names}}(pid, pname)$. We can now rewrite Property (2) to the input-bounded sentence

$$\forall pid, price\ [\xi'(pid, price)\ \mathbf{B}\ \neg(\overline{\mathsf{conf}}(\mathsf{name}, price) \wedge \overline{\mathsf{ship}}(\mathsf{name}, pid))\ ] \quad (4)$$

where $\xi'(pid, price)$ is short for

$$\text{PP} \wedge \mathsf{pay}(price) \wedge \mathsf{button}(\text{“}authorize\ payment\text{”}) \wedge \text{pick}(pid, price) \wedge \underline{\mathsf{prod\_prices}}(pid, price) \quad (5)$$

□

We can now state the main result of this section:

THEOREM 3.5. *The following are decidable:*

(i) given a Web service $\mathcal{W}$ with input-bounded rules, whether it is error free;

(ii) given an error-free Web service $\mathcal{W}$ with input-bounded rules and an input-bounded LTL-FO sentence φ over the schema of $\mathcal{W}$, whether $\mathcal{W}$ satisfies φ.

Furthermore, both problems are PSPACE*-complete for schemas with fixed bound on the arity, and in* EXPSPACE *for schemas with no fixed bound on the arity.*

In the remainder of the section, we outline the main steps in the proof of Theorem 3.5. We outline the main steps needed to prove Theorem 3.5. To begin, we note that (i) can be reduced to (ii).

LEMMA 3.6. *For each Web service $\mathcal{W}$ with input-bounded rules there exists an error-free Web service $\mathcal{W}'$ with input bounded rules, of size quadratic in $\mathcal{W}$, such that $\mathcal{W}$ is error free iff $\mathcal{W}' \models \varphi$, for some fixed input-bounded LTL-FO sentence φ.*

It follows from Lemma 3.6 that to prove Theorem 3.5 it is enough to establish PSPACE-hardness for (i), and inclusion in PSPACE for (ii). The PSPACE-hardness of (i) is shown by an easy reduction from Quantified Boolean Formula [16]. The upper bound for (ii) is much more involved, and is based on a reduction to the satisfiability problem for the logic E+TC (existential FO augmented with a transitive closure operator). This requires modifying and extending an analogous reduction used by Spielmann in [26, 28] for ASM transducers.

We next briefly review the relevant results from [26, 28]. Like Web services, and similarly to the earlier relational transducers [3], ASM transducers model database-driven reactive systems that respond to input events by taking some action, and maintain state information in designated relations. In terms of our framework, the ASM relational transducer can be viewed as a simplified Web service consisting of a single Web page. Like Web services, ASM transducers use database and state relations (called *memory* relations), as well as action and input relations. At each step, the transducer receives from the environment inputs consisting of arbitrary relations over the input vocabulary, whose elements come from the underlying database. The transducer reacts to the inputs with a state transition and by producing output relations. The control of the transducer is defined by rules similar to ours. The temporal properties to be verified are expressed by LTL-FO formulas.

In order to achieve decidability of verification, Spielmann considers several possible restrictions. The one of interest to us is *input-bounded ASM with bounded input flow*, denoted

ASM^I. This requires the following: (i) each input relation received in any single step has cardinality bounded by a constant, and (ii) the rules used in the specification, as well as the LTL-FO formula to be verified, are input bounded. The definition of input-bounded rule and formula are the same as ours, except that ASM rules use no $\mathbf{Prev_I}$ atoms.

In summary, the main differences between our input-bounded Web services and ASM^I transducers are:

- Web services use multiple Web pages and specify transitions among them,
- Web service inputs are restricted by input options defined by certain $\exists^*$FO formulas,
- The input vocabulary of Web services may contain input constants whose values are progressively supplied by users and need not come from the database, and
- Input-bounded Web services allow the use of $\mathbf{Prev_I}$ atoms, treated the same as input atoms.

Spielmann's proof of decidability of input-bounded LTL-FO properties of ASM^I transducers makes use of several logics for which finite satisfiability is decidable:

- FO^W, the *witness-bounded* fragment of FO;
- FO^W+ posTC, the extension of FO^W with the positive occurrences of the transitive closure operator, and
- E+TC, the existential fragment of FO+TC.

The main idea of the proof is to reduce the problem of checking the existence of a run of the transducer violating the desired property to that of checking finite satisfiability of a formula in one of the above logics. Specifically, this is done by an ingenious polynomial reduction to the finite satisfiability problem for FO^W+ posTC. Next, it is shown in [26] that finite satisfiability of FO^W+ posTC is polynomially reducible to the finite satisfiability of E+TC, and the latter is in PSPACE for fixed database arity and in EXPSPACE for arbitrary arity.

Our proof of containment in PSPACE (for fixed schema arity) requires adapting the proof of Spielmann's analogous result for ASM^I transducers. This is done in three steps:

1. We first extend the standard ASM model to allow for input option rules and the use of $prev_I$ atoms in all rules. We denote the resulting model, further restricted to be input-bounded with bounded input flow, by ASM^{IR}. We then show decidability of verification of input-bounded LTL-FO properties of ASM^{IR} transducers, via a direct reduction to finite satisfiability of E+TC (the reduction to FO^W+TC in [28] is no longer possible here).
2. Next, we define a special type of Web service, called "simple", that corresponds directly to ASM^{IR}. Simple Web services have a single Web page schema and are error free.
3. Finally, we reduce the general verification problem to verification of simple Web services.

REMARK 3.7. (**Sessions and Database Updates**) *Recall that our model prohibits database updates within a run. However, in practice it is very useful to update the database at various points in the interaction with users. In our running example, it makes sense to do so when a user logs out. In fact, the Web site implementing the full example specifies database update rules triggered at logout. This implicitly assumes that the runs to be verified consist of interactions of a single user between login and logout. Indeed, these are natural boundaries of* sessions *to be verified, and can be specified implicitly within the temporal formula to be verified. However, other definitions of sessions are possible (see also the discussion in the Conclusions).*

Boundaries of decidability One may wonder whether our restrictions can be relaxed without affecting decidability of verification. Unfortunately, even small relaxations of these restrictions can lead to undecidability. Specifically, we consider the following: (i) relaxing the requirement that state atoms be ground in formulas defining input options, by allowing state atoms with variables, (ii) relaxing the input-bounded restriction by allowing a very limited form of non input-bounded quantification in the form of state projections, (iii) allowing $prev_I$ relations to record *all* previous inputs to I rather than just the most recent one, and (iv) extending LTL-FO formulas with path quantification.

We begin with extension (i) and show undecidability even for a fixed LTL-FO formula and input options defined by quantifier-free FO formulas using just database and state relations.

THEOREM 3.8. *There exists a fixed input-bounded LTL-FO formula φ for which it is undecidable, given an input-bounded Web service $\mathcal{W}$ with input options defined by quantifier-free FO formulas over database and state relations, whether $\mathcal{W} \models \varphi$.*

Proof: The proof is by reduction of the question of whether a Turing Machine (TM) M halts on input ϵ. Let M be a deterministic TM with a left-bounded, right-infinite tape. We construct a Web service with a single page (excepting the error page). The idea is to represent configurations of M using a 4-ary state relation T. The first two coordinates of T represent a successor relation on a subset of the active domain of the database. A tuple $T(x, y, u, v)$ says that the content of the x-th cell is u, the next cell is y, and v is a state p iff M is in state p and the head is on cell x. Otherwise, v is some special symbol #. The moves of M are simulated by modifying T accordingly. M halts on input ϵ iff there exists a run of $\mathcal{W}$ on some database such that some halting state h is reached. Thus, M does not accept ϵ iff for every run, $T(x, y, u, h)$ does not hold for any x, y, u, that is, $\mathcal{W} \models \forall x \forall y \forall u \mathbf{G}(\neg T(x, y, u, h))$.

We now outline the construction of $\mathcal{W}$ in more detail. The database schema of $\mathcal{W}$ consists of a unary relation D and a constant *min*. The state relations are the following:

- T, a 4-ary relation;
- $Cell$, Max, and $Head$, unary relations;
- propositional states used to control the computation: *initialized*, *simul*

The input relations are I and H, both unary.

The first phase of the simulation constructs the initial configuration of M on input ϵ, and the tape that the current run will make available for the computation. This phase makes use of the unary input relation I. Intuitively, the role of I is to pick a new value from the active domain, that has not yet been used to identify a cell, and use it to identify a new cell of the tape. The state relation *Cell* keeps track of the values previously chosen, to prevent them from being chosen again. The state relation *Max* keeps track of the most recently inserted value. The rules implementing the initialization are the following (the symbol b denotes the blank symbol of M and q_0 is the start state):

$$
\begin{array}{lcl}
Options_I(y) & \leftarrow & D(y),\ y \neq min,\ \neg Cell(y),\ \neg simul \\
T(min, y, b, q_0) & \leftarrow & I(y),\ \neg initialized \\
Cell(min) & \leftarrow & \neg initialized \\
Head(min) & \leftarrow & \neg initialized \\
initialized & \leftarrow & \neg initialized \\
T(x, y, b, \#) & \leftarrow & I(y),\ Max(x) \\
Cell(y) & \leftarrow & I(y) \\
\neg Max(x) & \leftarrow & Max(x) \\
Max(y) & \leftarrow & I(y) \\
simul & \leftarrow & \forall y \neg I(y)
\end{array}
$$

The state *simul* signals the transition to the simulation phase. Notice that this happens either if the input options for I become empty (because we have used the entire active domain) or because the input is empty at any point. In the simulation phase, T is updated to reflect the consecutive moves of M. The simulation is aborted if T runs out of tape. We illustrate the simulation with an example move. Suppose M is in state p, the head is at cell x, the content of the cell is 0, and the move of M in this configuration consists of overwriting 0 with 1, changing states from p to q, and moving right. The rules simulating this move are the following:

$$
Options_H(x, y, 0, p) \leftarrow simul,\ Head(x), T(x, y, 0, p)
$$

$$
\begin{array}{lcl}
\neg T(x, y, 0, p) & \leftarrow & simul,\ H(x, y, 0, p) \\
T(x, y, 1, \#) & \leftarrow & simul, H(x, y, 0, p) \\
\neg T(y, z, u, \#) & \leftarrow & simul, H(x, y, 0, p), T(y, z, u, \#) \\
T(y, z, u, q) & \leftarrow & simul, H(x, y, 0, p), T(y, z, u, \#) \\
\neg Head(x) & \leftarrow & simul, H(x, y, 0, p) \\
Head(y) & \leftarrow & simul, H(x, y, 0, p)
\end{array}
$$

Such rules are included for every move of M. It is easy to see that this correctly simulates the moves of M. Note that if the input H is empty, T does not change. Finally, if the head reaches the last value provided in T, the transducer goes into an infinite loop in which, again, T stays unchanged. Thus, $T(x, y, u, h)$ holds in some run iff the computation of M on ϵ is halting. Equivalently, M does not halt on ϵ iff the transducer satisfies the formula $\varphi = \forall x \forall y \forall u \mathbf{G}(\neg T(x, y, u, h))$. □ □

We next consider extension (ii): we relax input-boundedness of rules by allowing projections of state relations. We call a Web service *input-bounded with state projections* if all its formulas are input-bounded, excepting state rules that allow insertions of the form:

$$
S(\bar{x}) \leftarrow \exists \bar{y}\ S'(\bar{x}, \bar{y})
$$

where S and S' are state relations. We can show the following.

THEOREM 3.9. *It is undecidable, given a simple, input-bounded Web service $\mathcal{W}$ with state projections and input-bounded LTL-FO sentence φ, whether $\mathcal{W} \models \varphi$.*

Proof: Reduction of the implication problem for functional and inclusion dependencies, known to be undecidable [9]. □

We now deal with extension (iii). We say that a Web service has *lossless input* if the $prev_I$ relations record *all* previous inputs to I in the current run.

THEOREM 3.10. *It is undecidable, given an input-bounded Web service $\mathcal{W}$ with lossless input and an input-bounded LTL-FO formula φ, whether $\mathcal{W} \models \varphi$.*

Proof: Straightforward reduction of the undecidability of finite validity of FO formulas. □

The undecidability of extension (iv) is shown in the next section, after the notation on branching-time logics is introduced.

4. BRANCHING-TIME PROPERTIES

In this section we consider the verification of temporal properties of Web services involving quantification over runs. This allows expressing properties involving multiple runs of a Web service rather than just individual ones, such as "at any point in a run there is a way to return to the home page". To specify such properties, we use variants of the logics CTL and CTL* [13] extending the logic LTL-FO used in the previous section. These variants, denoted CTL-FO and CTL*-FO, extend LTL-FO with path quantifiers **E** and **A**, subject to certain syntactic restrictions. Informally, $\mathbf{E}\varphi$ stands for "there exists a continuation of the current run that satisfies φ" and $\mathbf{A}\varphi$ means "every continuation of the

current run satisfies φ". We refer the reader to [1, 28] for further details. Satisfaction of a CTL$^{(*)}$-FO sentence by a Web service $\mathcal{W}$ is defined using the tree corresponding to the runs of $\mathcal{W}$ on a given database D. The nodes of the tree consist of all prefixes of runs of $\mathcal{W}$ on D (the empty prefix is denoted *root* and is the root of the tree). A prefix π is a child of a prefix π' iff π extends π' with a single configuration. We denote the resulting infinite tree by $\mathcal{T}_{\mathcal{W},D}$. Note that $\mathcal{T}_{\mathcal{W},D}$ has only infinite branches (so no leafs) and each such infinite branch corresponds to a run of $\mathcal{W}$ on database D. Satisfaction of an CTL$^{(*)}$-FO sentence by $\mathcal{T}_{\mathcal{W},D}$ is the natural extension of the classical notion of satisfaction of CTL$^{(*)}$ formulas by infinite trees labeled with propositional variables (e.g., see [13]). The main difference is that propositional variables are not explicitly provided; instead, the relevant FO formulas have to be evaluated on the current configuration (last of the prefix defining the node). We say that a Web service $\mathcal{W}$ satisfies φ, denoted $\mathcal{W} \models \varphi$, iff $\mathcal{T}_{\mathcal{W},D} \models \varphi$ for every database D.

The above notion of satisfaction of a CTL$^{(*)}$-FO formula by an infinite tree is closely related to the classical semantics of CTL$^{(*)}$ based on *Kripke structures*. Informally, a Kripke structure K is a finite graph labeled by sets of propositions from a finite set Σ, with a designated root vertex. The Kripke structure K satisfies a CTL$^{(*)}$ formula φ using propositions in Σ iff the infinite labeled tree obtained by unfolding K starting from the root satisfies φ. The complexity of checking whether a CTL$^{(*)}$ formula is satisfied by a Kripke structure (model checking) is in PTIME for CTL and PSPACE-complete for CTL*. The satisfiability problem for CTL$^{(*)}$ formulas is EXPTIME-complete for CTL and 2-EXPTIME complete for CTL*. See [13] for a concise survey on temporal logics, and further references.

Example 4.1 The following CTL*-FO sentence expresses the fact that in every run, whenever a product *pid* is bought by a user, it will eventually ship, but until that happens, the user can still cancel the order for *pid*.

$$\forall pid \forall price\ A\mathbf{G}(\xi'(pid, price) \rightarrow A((E\mathbf{F}\overline{\mathsf{cancel}}(\mathsf{name}, pid))\mathbf{U}(\overline{\mathsf{ship}}(\mathsf{name}, pid))))$$

where ξ' is the formula defined in Example 3.4 (5). □

As noted earlier, the decidability results of the previous section do not extend to CTL$^{(*)}$-FO sentences, even if they are restricted to be input bounded (by requiring every FO subformula to be input bounded). Indeed, the following can be shown.

THEOREM 4.2. *It is undecidable, given a simple, input-bounded Web service $\mathcal{W}$ and input-bounded CTL-FO sentence φ, whether $\mathcal{W} \models \varphi$.*

The proof further shows that a single alternation of path quantifiers is sufficient to yield undecidability, since one alternation is enough to express validity of FO sentences in the prefix class $\exists^*\forall^*$FO, known to be undecidable [5].

We next consider several restrictions leading to decidability of the verification problem for CTL$^{(*)}$-FO sentences.

Propositional input-bounded Web services The first restriction further limits input-bounded Web services by requiring all states and actions to be propositional. Furthermore, no rules can use $\mathbf{Prev_I}$ atoms. We call such Web services *propositional*. In a propositional Web service, inputs can still be parameterized in the Web service specification. The CTL* formulas we consider are propositional and use input, action, state, and Web page symbols, viewed as propositions. Satisfaction of such a CTL* formula by a Web service is defined as for CTL*-FO, where truth of propositional symbols in a given configuration $\langle V, S, I, A\rangle$ is defined as follows: a Web page symbol is true iff it equals V, a state symbol s is true iff $s \in S$, an input symbol I is true iff $I \in \mathbf{I}_V$, and an action symbol a is true iff $a \in A$.

Example 4.3 CTL$^{(*)}$-FO is particularly useful for specifying navigational properties of Web services. Note that these services do not necessarily have to be propositional; we could abstract their predicates to propositional symbols, thus concentrating only on reachability properties.[2] For our running example, abstracting all non-input atoms to propositions, we could ask whether from any page it is possible to navigate to the home page HP using the following CTL sentence:

$$A\mathbf{G}E\mathbf{F}(\text{ HP})$$

The following CTL property states that, after login, the user can reach a page where he can authorize payment for a product:[3]

$$A\mathbf{G}((\text{ HP} \wedge \mathsf{button}(\text{“}login\text{”})) \rightarrow E\mathbf{F}(\mathsf{button}(\text{“}authorize\ payment\text{”})))$$

where $\mathsf{button}(\text{“}login\text{”}), \mathsf{button}(\text{“}authorize\ payment\text{”})$ denote the corresponding propositions. In the specification of the abstracted service, we can still allow in the home page HP a state rule that checks successful login:

$$\text{logged\_in} \leftarrow \underline{\mathsf{users}}(\mathsf{name}, \mathsf{password}) \wedge \mathsf{button}(\text{“}login\text{”}).$$

□

For a given Web service $\mathcal{W} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, W_0, W_\epsilon\rangle$, we denote by $\Sigma_\mathcal{W}$ the propositional vocabulary consisting in all symbols in $\mathbf{S} \cup \mathbf{I} \cup \mathbf{A} \cup \mathbf{W}$. By abuse of notation, we use the same symbol for a relation R in the vocabulary of $\mathcal{W}$ and for the corresponding propositional symbol in $\Sigma_\mathcal{W}$. We first show the following.

THEOREM 4.4. *Given a propositional, input-bounded Web service $\mathcal{W}$ and a CTL* formula φ over $\Sigma_\mathcal{W}$, it is decidable whether $\mathcal{W} \models \varphi$. The complexity of the decision procedure is* CO-NEXPTIME *if φ is in CTL, and* EXPSPACE *if φ is in CTL*.*

[2]This is in the spirit of program verification, where program variables are first abstracted to booleans, in order to check CTL* properties such as liveness.

[3]The most important property in electronic commerce ☺

Proof: The proof has two stages. First, we show that there is a bound on the size of databases that need to be considered when checking for violations of φ (or equivalently, satisfaction of $\neg\varphi$). Second, we prove that for a given database D there exists a Kripke structure $\mathcal{K}_{\mathcal{W},D}$ over alphabet $\Sigma_{\mathcal{W}}$, of size exponential in $\Sigma_{\mathcal{W}}$, such that $\mathcal{T}_{\mathcal{W},D} \models \neg\varphi$ iff $\mathcal{K}_{\mathcal{W},D} \models \neg\varphi$. This allows us to use known model-checking techniques for CTL$^{(*)}$ on Kripke structures to verify whether $\mathcal{T}_{\mathcal{W},D} \models \neg\varphi$. □

The complexity of the decision problem of Theorem 4.4 can be decreased under additional assumptions. The following result focuses on verification of navigational properties of Web sites, expressed by CTL* formulas over alphabet $\mathbf{W}$.

COROLLARY 4.5. *Let* $\mathbf{S}$ *be a fixed set of state propositions and* $\mathbf{D}$ *a fixed database schema. Given a propositional, input-bounded, error-free Web service* $\mathcal{W}$ *with states* $\mathbf{S}$ *and database schema* $\mathbf{D}$*, and a CTL* formula* φ *over* $\mathbf{W}$*, it is decidable in* PSPACE *whether* $\mathcal{W} \models \varphi$*.*

Proof: The decision procedure is similar to that for Theorem 4.4. Since $\mathbf{S}$ is fixed and φ refers only to $\mathbf{W}$, it is enough to retain, in labels of $\mathcal{T}_{\mathcal{W},D}$ only the states and Web page names. Since $\mathcal{W}$ is error free, there is exactly one Web page name per label. With some work, this yields the PSPACE upper bound. □

Another special case of interest involves Web services that are entirely propositional. Thus, the database plays no role in the specification: inputs, states, and actions are all propositional, and the rules do not use the database. Let us call such a Web service *fully propositional*. We can show the following.

THEOREM 4.6. *Given a fully propositional Web service* $\mathcal{W}$ *and a CTL* formula* φ *over* $\Sigma_{\mathcal{W}}$*, it is decidable in* PSPACE *whether* $\mathcal{W} \models \varphi$*.*

Proof: In the case of a fully propositional Web service $\mathcal{W}$, the Kripke structure $\mathcal{K}_{\mathcal{W},D}$ is independent of D (let us denote it by $\mathcal{K}_{\mathcal{W}}$). However, unlike the Kripke structure used in the proof of Corollary 4.5, $\mathcal{K}_{\mathcal{W}}$ is exponential wrt $\mathcal{W}$ so cannot be constructed in PSPACE. We therefore need a more subtle approach, that circumvents the explicit construction of $\mathcal{K}_{\mathcal{W}}$. To do so, we adopt techniques developed in the context of model checking for concurrent programs (modeled by propositional transition systems). Specifically, the model checking algorithm developed by Kupferman, Vardi and Wolper in [20] can be adapted to fully propositional Web services. The algorithm uses a special kind of tree automaton, called *hesitant alternating tree automaton* (HAA) (see [20] for the definition). As shown in [20], for each CTL* formula φ one can construct an HAA A_φ accepting precisely the trees (with degrees in a specified finite set) that satisfy φ. In particular, for a given Kripke structure $\mathcal{K}$, one can construct a product HAA $\mathcal{K} \times A_\varphi$ that is nonempty iff $\mathcal{K} \models \varphi$. The nonemptiness test can be rendered efficient using the crucial observation that nonemptiness of $\mathcal{K} \times A_\varphi$ can be reduced to the nonemptiness of a corresponding word HAA over a 1-letter alphabet, which is shown to be decidable in linear time, unlike the general nonemptiness problem for alternating tree automata. Finally, it is shown that $\mathcal{K} \times A_\varphi$ need not be constructed explicitly. Instead, its transitions can be generated on-the-fly from $\mathcal{K}$ and φ, as needed in the nonemptiness test for the 1-letter word HAA corresponding to $\mathcal{K} \times A_\varphi$. This yields a model checking algorithm of space complexity polynomial in φ and polylogarithmic in $\mathcal{K}$.

In our case, $\mathcal{K}$ is $\mathcal{K}_{\mathcal{W}}$, and the input consists of φ and $\mathcal{W}$ instead of φ and $\mathcal{K}_{\mathcal{W}}$. The previous approach can be adapted by pushing further the on-the-fly generation of $\mathcal{K}_{\mathcal{W}} \times A_\varphi$ by also generating on-the-fly the relevant edges of $\mathcal{K}_{\mathcal{W}}$ from $\mathcal{W}$ when needed. This yields a polynomial space algorithm for checking whether $\mathcal{W} \models \varphi$, similar to the algorithm with the same complexity obtained in [20] for model checking of concurrent programs. □

One may wonder if the restrictions of Theorem 4.4 can be relaxed without compromising the decidability of verification. In particular, it would be of interest if one could lift some of the restrictions on the propositional nature of states and actions. Unfortunately, we have shown that allowing parameterized actions leads to undecidability of verification, even for CTL formulas whose only use of action predicates is to check emptiness. The proof is by reduction of the implication problem for functional and inclusion dependencies. We omit the details.

Web services with input-driven search The restrictions considered so far require states of a Web service to be propositional, and do not allow the use of $\mathbf{Prev_I}$ atoms. Although adequate for some verification tasks, this is a serious limitation in many situations, since no values can be passed on from one Web page to another. We next alleviate some of this limitation by considering Web services that allow limited use of $\mathbf{Prev_I}$ atoms. This can model commonly arising applications involving a user-driven search, going through consecutive stages of refinement. More formally:

DEFINITION 4.7. *A* Web service with input-driven search *is an input-bounded Web service*

$$\mathcal{W} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, W_0, W_\epsilon \rangle$$

where:

- $\mathbf{I}$ *consists of a single unary relation* I
- $\mathbf{S}$ *consists of propositional states including* not-start
- $\mathbf{A}$ *is propositional*
- $\mathbf{D}$ *includes a constant* i_0 *and a designated binary relation* R_I
- *the state rule for* not-start *is not-start* $\leftarrow \neg$*not-start*

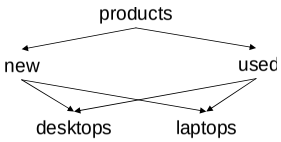

Figure 1: Fragment of R_I for Example 4.8

- *the input option rule for I is in all Web pages of the form*

  $$Options_I(y) \leftarrow (\neg not\text{-}start \ \wedge \ y = i_0) \\ \vee (not\text{-}start \ \wedge \ \exists x(prev_I(x) \ \wedge \ R_I(x,y)) \ \wedge \ \varphi(y))$$

  where $\varphi(y)$ is a quantifier-free formula over $\mathbf{D} \cup \mathbf{S}$ with free variable y.

Note that *not-start* is false at the start of the computation and true thereafter. To initialize the search, the first input option is the constant i_0. Subsequently (when *not-start* is true), if x was the previously chosen input, the allowed next inputs are the y's for which $R_I(x,y) \ \wedge \ \varphi(y)$ holds, where R_I is the special input search relation and φ places some additional condition on y involving the database and the propositional states.

Example 4.8 Consider a variation of a computer-selling Web site which doesn't just partition its products into desktops and laptops, but rather uses the more complex classification depicted in Figure 1. The user can search the hierarchy of categories, and will only see a certain category if it is currently in stock, as reflected by the database. The propositional state new is set on the page which offers the choice between new and used products. The page schemas for new and old computers are reused, so when generating the options, the Web site must consult state new to distinguish among new and old products. We can abstract this Web site as a Web service with input-driven search, in which the binary database relation R_I is a graph which contains as a subgraph the one in Figure 1, and in which the unary database relations such as newDesktop,usedDesktop,usedLaptop contain the in-stock products. Here is the input rule corresponding to the desktop search page:

$$\begin{aligned} Options_I(y) \leftarrow & (\neg not\text{-}start \wedge y = i_0) \vee \\ & not\text{-}start \wedge \exists x(prev_I(x) \wedge R_I(x,y)) \wedge \\ & (\text{new} \wedge \underline{\text{newDesktop}}(y) \vee \neg \text{new} \\ & \wedge \underline{\text{usedDesktop}}(y)) \end{aligned} \tag{6}$$

□

We can show the following.

THEOREM 4.9. *Given a Web service with input-driven search $\mathcal{W}$ and a CTL* formula φ, it is decidable whether $\mathcal{W} \models \varphi$ in* EXPTIME *if φ is in CTL, and* 2-EXPTIME *if φ is in CTL*.*

Proof: We reduce the problem of checking whether $\mathcal{W} \models \varphi$ to the satisfiability problem for CTL$^{(*)}$ formulas, known to be EXPTIME-complete for CTL and 2-EXPTIME complete for CTL* [13]. □

5. CONCLUSIONS

We have identified a practically appealing and fairly tight class of Web services and linear-time temporal formulas for which verification is decidable. The complexity of verification is PSPACE-complete (for fixed database arity). This is quite reasonable as static analysis goes[4]. For branching-time properties, we identified decidable restrictions for which the complexity of verification ranges from PSPACE to 2-EXPTIME. To obtain these results, we used a mix of techniques from logic and automatic verification.

Other interesting aspects of Web service verification could not be addressed in this paper and are left for future work. We mention a few of them.

Specifying and verifying sessions As discussed in Section 3, in practical Web service applications it is not always realistic to assume that verification applies to *all* possible runs of the service. This may be due to various reasons: there may be a need to verify properties of complex services in a modular fashion, the restrictions needed for decidability may only hold for certain portions of runs, etc. Let us call portions of runs to be verified *sessions*. Some sessions can be specified implicitly within the temporal formula to be verified, while others may require explicit definition by other means. It is of interest to understand what types of sessions can be verified by our approach. For instance, in our running example, the default assumption is that sessions consist of single-user runs beginning at login and ending at logout. However, other types of sessions can be fit to our restrictions, including multi-user sessions (as long as no database updates occur within the session and only a bounded number of new users register).

Interacting Web services An important aspect of Web services is the interaction of multiple services and their composition into more complex services (as in e-service composition, see [19]). On the face of it, our model concerns the behavior of a single Web service interacting with its environment. However, it can also capture to some extent the interaction and composition of multiple Web services. For example, external calls to a service viewed as a black box can be modeled simply by an extra database relation with a limited access pattern. In terms of verification, certain properties of the sequence of messages exchanged by Web services (called *conversations* in the framework of WSDL [31]) can be specified using our temporal formulas. We plan to further explore to what extent interacting Web services can be modeled in our framework and their properties verified by our techniques.

Algorithms and heuristics for verification While our positive results provide complexity-theoretic upper bounds on Web service verification, significant work is still needed in order to obtain practical algorithms and heuristics. We plan

[4]Recall that even testing inclusion of two conjunctive queries is NP-complete!

to explore this issue, including the use of theorem-proving techniques in conjunction with the logic and automata-based approaches suggested in the paper.

Acknowledgement

We wish to thank Caroline Cruz and Dayou Zhou for their help in implementing the demo Web site for our running example, and Ioana Manolescu and Marc Spielmann for useful comments on the paper.

6. REFERENCES

[1] S. Abiteboul, L. Herr, and J. V. den Bussche. Temporal versus first-order logic to query temporal databases. In *Proc. ACM PODS*, pages 49–57, 1996.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *JCSS*, 61(2):236–269, 2000. Extended abstract in PODS 98.

[4] A. J. Bonner and M. Kifer. An overview of transaction logic. *Theor. Comput. Sci.*, 133(2):205–265, 1994.

[5] E. Borger, E. Gradel, and Y. Gurevich. *The Classical Decision Problem*. Springer, 1997.

[6] BPML.org. Business process modeling language. http://www.bpmi.org.

[7] M. Brambilla, S. Ceri, S. Comai, P. Fraternali, and I. Manolescu. Specification and design of workflow-driven hypertexts. *Journal of Web Engineering*, 1(1), 2002.

[8] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing data-intensive Web applications*. Morgan-Kaufmann, 2002.

[9] A. K. Chandra and M. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM J. Comp.*, 14(3):671–677, 1985.

[10] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for Web services. http://dev2dev.bea.com/techtrack/BPEL4WS.jsp.

[11] DAML-S Coalition (A. Ankolekar et al).DAML-S: Web service description for the semantic Web. In *The Semantic Web - ISWC*, pages 348–363, 2002.

[12] H. Davulcu, M. Kifer, C. R. Ramakrishnan, and I. V. Ramakrishnan. Logic based modeling and analysis of workflows. In *Proc. ACM PODS*, pages 25–33, 1998.

[13] E. A. Emerson. Temporal and modal logic. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 995–1072. North-Holland Pub. Co./MIT Press, 1990.

[14] M. F. Fernández, D. Florescu, A. Y. Levy, and D. Suciu. Declarative specification of web sites with Strudel. *VLDB Journal*, 9(1):38–55, 2000.

[15] D. Florescu, K. Yagoub, P. Valduriez, and V. Issarny. WEAVE: A data-intensive web site management system(software demonstration). In *Proc. of the Conf. on Extending Database Technology (EDBT)*, 2000.

[16] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, 1979.

[17] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[18] D. Harel. On the formal semantics of statecharts. In *Proc. LICS*, pages 54–64, 1987.

[19] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: a look behind the curtain. In *Proc. ACM PODS*, pages 1–14, 2003.

[20] O. Kupferman, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *J. of ACM*, 47(2):312–360, 2000.

[21] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1991.

[22] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer Verlag, 1995.

[23] G. Mecca, P. Merialdo, and P. Atzeni. Araneus in the era of XML. *IEEE Data Engineering Bulletin*, 22(3):19–26, 1999.

[24] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of Web services. In *Proc. WWW*, pages 77–88, 2002.

[25] R. Reiter. *Knowledge in action: logical foundations for specifying and implementing dynamical systems*. MIT Press, 2001.

[26] M. Spielmann. Abstract State Machines: Verification problems and complexity. Ph.D. thesis, RWTH Aachen, 2000.

[27] M. Spielmann. Automatic verification of Abstract State Machines. In *Proc. CAV*, pages 431–442, 1999.

[28] M. Spielmann. Verification of relational transducers for electronic commerce. *JCSS.*, 66(1):40–65, 2003. Extended abstract in PODS 2000.

[29] D. Wodtke and G. Weikum. A formal foundation for distributed workflow execution based on state charts. In *Proc. ICDT*, pages 231–246, 1997.

[30] Workflow management coalition, 2001. http://www.wfmc.org.

[31] Web Services Description Language(WSDL 1.1), 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[32] Web Services Flow Language(WSFL 1.0), 2001. http://www-3.ibm.com/ software/ solutions /webservices/pdf/WSFL.pdf.