A Verifier for Interactive, Data-driven Web Applications

Alin Deutsch Monica Marcus Liying Sui Victor Vianu Dayou Zhou

University of California, San Diego
Computer Science and Engineering
{deutsch,mmarcus,lsui,vianu,dzhou}@cs.ucsd.edu

ABSTRACT

We present WAVE, a verifier for interactive, database-driven Web applications specified using high-level modeling tools such as WebML. WAVE is complete for a broad class of applications and temporal properties. For other applications, WAVE can be used as an incomplete verifier, as commonly done in software verification. Our experiments on four representative data-driven applications and a battery of common properties yielded surprisingly good verification times, on the order of seconds. This suggests that interactive applications controlled by database queries may be unusually well suited to automatic verification. They also show that the coupling of model checking with database optimization techniques used in the implementation of WAVE can be extremely effective. This is significant both to the database area and to automatic verification in general.

1. INTRODUCTION

Web applications interacting with users or programs while accessing an underlying database are increasingly common. They include e-commerce sites, scientific and other domain-specific portals, e-government, and data-driven Web services. The spread of such applications has been accompanied by the emergence of tools for their high-level specification. A representative, commercially successful example is WebML [9, 8], which allows to specify a Web application using an interactive variant of the E-R model augmented with a workflow formalism. The code for the Web application is automatically generated from the WebML specification. This not only allows fast prototyping and improves programmer productivity but, as we argue in this paper, provides new opportunities for the automatic verification of Web applications. Indeed, we describe a verifier we have implemented that can check temporal properties of WebML-style specifications and is complete under reasonable restrictions. Such verification leads to increased confidence in the correctness of database-driven Web applications generated from high-level specifications, by addressing the most likely source of errors (the application's specification, as opposed to the less likely errors in the automatic generator's implementation).

We focus on interactive Web sites generating Web pages dynamically by queries on an underlying database. The Web site accepts input from external users or programs. It responds by taking some action, updating its internal state database, and moving to a new Web page determined by yet another query. A *run* is a sequence of inputs together with the Web pages, states, and actions generated by the Web site. The properties we wish to verify range from basic soundness of the specification (e.g. the next Web page to be displayed is always uniquely defined) to semantic properties (e.g. no order is shipped before a payment in the right amount is received). Such properties are expressed using an extension of *linear-time temporal logic* (LTL). The task of a verifier is to check that all runs of the Web site satisfy the given property (as usual in verification, runs are considered to be infinite). Verifiers search for counter-examples to the desired property, i.e. runs leading to a violation. A verifier is *complete* if it is guaranteed to find a counter-example whenever one exists.

In the broader context of verification, a database-driven Web application is an *infinite-state* system, because the underlying database queried by the application is not fixed in advance. This poses an immediate and seemingly insurmountable challenge. Classical verification deals with finite-state systems, modeled in terms of propositions. For more expressive specifications, the traditional approach suggests the following strategy: first abstract the specification to a fully propositional one and next apply an existing model checker such as SPIN [20] to verify LTL properties of the abstracted model. This approach is unsatisfactory when the data values are first-class citizens, as in data-driven Web applications. For example, abstraction would allow checking that *some* order was shipped only after *some* payment was completed. However, we could not inspect the payment and order data values to verify that the payment was for the shipped item, and in the correct amount.

Conventional wisdom holds that, short of using abstraction, it is hopeless to attempt complete verification of infinite-state systems. In this respect, WAVE represents a significant departure because it is complete for a practically relevant class of infinite-state specifications. As far as we know, this is the first implementation of such a verifier. Moreover, our experiments measuring verification times for a battery of typical properties of four different Web applications are extremely positive. These results suggest that complete verification of a significant range of Web applications is well within reach.

Completeness of verification is only guaranteed under certain restrictions described shortly. To show that these restrictions cover a large class of applications, we have modeled a computer shopping Web site similar to the Dell site, an airline reservation application similar to Expedia, an online bookstore in the spirit of Barnes & Noble, and a sports Web site on the Motorcycle Grand Prix (all published at [1]). We used these applications in our experimental evaluation of WAVE. Note that if the specification and the property do not satisfy the restrictions needed for completeness, WAVE can still be used as an incomplete verifier, as typically done in software verification. The heuristics we developed remain just as effective in this case.

We now describe informally the restrictions on the Web site specifications and properties that guarantee completeness, called *input boundedness* [29, 14]. We model the queries used in the specification of the Web site as first-order queries (FO), also known as relational calculus. FO can be viewed as an abstraction of the data manipulation core of SQL. In a nutshell, input boundedness restricts the range of quantifications in FO formulas to values occurring in the input. This is natural, since interactive Web applications are input-driven. For example, to state that every payment received is in the right amount, one might use the input-bounded formula $\forall x \forall y[pay(x, y) \rightarrow price(x, y)]$, where $pay(x, y)$ is an input and *price* is a database relation providing the price for each item.

The theoretical results of [29, 14] show the decidability of model checking for input-bounded specifications and properties, by reduction to the finite satisfiability problem for the logic existential FO extended with a transitive closure operator (E+TC). The complexity of checking that a Web site specification $\mathcal{W}$ satisfies a property φ is shown to be PSPACE. This upper bound is a positive starting point, but provides no indication of whether verification is actually feasible in practice. The WAVE tool demonstrates that this is in fact the case, using a fruitful coupling of novel verification and database optimization techniques. We briefly outline the main difficulties overcome in implementing WAVE.

In our scenario, a first difficulty facing a verifier is that exhaustive exploration of all possible runs of a Web site $\mathcal{W}$ on all databases is impossible since there are infinitely many possible databases and the length of runs is infinite. A fundamental consequence of results in [29] is that, for input-bounded specifications $\mathcal{W}$ and properties φ, it is sufficient to consider databases and runs of size bounded by an exponential in $\mathcal{W}$ and φ. However, this yields a doubly exponential state space, which is impossible to explore even for very small specifications. Therefore, we need a qualitatively different approach. The solution lies in avoiding explicit exploration of the state space. Instead of materializing a full initial database and exploring the possible runs on it, we generate runs by lazily making at each point in the run just the assumptions needed to obtain the next configuration. Specifically, for input-bounded $\mathcal{W}$ and φ, this can be done as follows:

(i) explicitly specify the tuples in the database that use only a small set of relevant constants C computed from $\mathcal{W}$ and φ; this is called the *core* of the database and remains unchanged throughout the run.

(ii) at each step in the run, make additional assumptions about the content of the database, needed to determine the next possible configurations. The assumptions involve only a small set of additional values.

The key point is that the local assumptions made in (ii) at each step need not be checked for global consistency. Indeed, a non-obvious consequence of the input-bounded restriction is that these assumptions are guaranteed to be globally consistent with *some* very large database which is however never explicitly constructed. This dramatically cuts down the space explored by the verifier. However, verification becomes practical only in conjunction with an array of heuristics and optimization techniques. This yields critical improvements, bringing the verification times in our experiments down to seconds.

In summary, the main contribution of our work is an extension of finite-state model checking techniques to data-aware reactive systems in general, and data-driven interactive Web applications in particular. This resulted in the implementation of WAVE (Web Application VErifier).

Paper Outline. The language of Web site specifications, and the temporal logic used to express properties, are presented in Section 2.1. Section 2.2 provides some background on classical model checking. Our verification algorithm is presented in Section 3. In particular, Section 3.2 addresses optimizations exploiting the structure of the database and specification rules. Section 4 details how our implementation exploits the capabilities of a main-memory database management system and Section 5 reports on the experimental evaluation of WAVE. We conclude with related work (Section 6) and a discussion (Section 7).

2. PRELIMINARIES

2.1 The Model

We use a model for high-level specifications of Web applications that was first introduced and studied in [14]. The model is similar in flavor to WebML. For the reader's convenience, we informally summarize the model and theoretical results of [14] that are relevant to our implementation. A Web site specification (spec) $\mathcal{W}$ consists of a finite set of Web page schemas, of which one is designated as the home page, together with a database relational schema $\mathcal{D}$ and a state relational schema $\mathcal{S}$. Each Web page schema serves as a template for dynamically generating Web pages. A Web page schema W specifies the following:

- The types of inputs accepted by W. Users can provide input in two ways: as text input requested by the Web site (e.g. *user-name, password, credit-card-no*, etc) or as a choice from one or several option lists (modeling pull-down menus, radio buttons, scroll-down lists, etc.) dynamically generated by the Web page. Formally, W provides an input schema consisting of constants and relations. The constants represent text input requests (such as the *credit-card-no* above). Their value is defined once provided by the user, and undefined otherwise. The relations represent input option lists. For each input relation R, the options generated by the Web page are defined by an input rule of the form $Options_R(\bar{x}) \leftarrow \varphi(\bar{x})$, where φ is an FO query on the database, state relations, and inputs provided by the user at the *previous* step. Note that clicking html links and buttons can be easily modeled as choices from a list of options.

- State update rules specifying the tuples to be inserted or deleted from state relations of $\mathcal{S}$. Insertions in a state S are specified by rules of the form $S(\bar{x}) \leftarrow \varphi(\bar{x})$ and deletions by rules $\neg S(\bar{x}) \leftarrow \varphi(\bar{x})$ where φ is an FO query on the database, current state relations, and the current or previous user input. Conflicts between insertions and deletions are treated as no-ops. States for which no rule is specified remain unchanged.
- Actions taken in response to user input. Actions (such as sending an email, an invoice, or shipping a product) are modeled as insertions into action relations associated with the Web page. They are specified by rules of the same shape as state insertion rules.
- Target Web page rules. These specify, for each possible next Web page, a condition under which the transition occurs. The conditions are FO queries on the database, current state, and current or previous user inputs.

The Web site defined by a spec $\mathcal{W}$ produces a sequence of Web pages in response to user inputs, starting at the home page. Transitions occur as follows. Each Web page first generates the input options specified by the rules and requests values for the input constants in its input schema. The user responds by making at most one choice from among the input options for each input relation, and providing values for the required input constants. In response to the user's inputs, the Web site takes the actions defined by the action rules, updates the state relations as specified by the state insertion and deletion rules, and moves to the Web page whose associated condition in its target rule evaluates to true (if several conditions are true, no transition occurs). The content of the database, state relations, current Web page, current input choices, and actions computed in response to inputs, form a *configuration* of $\mathcal{W}$. A *run* over a database instance D is an infinite sequence of configurations $\{C_i\}_{i\geq 0}$ where C_0 is the initial configuration of the home page (the database is D and all states and previous inputs are empty) and C_{i+1} is obtained from C_i as described above. The database remains unchanged within a run, unlike state relations.

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.

Example 2.1 We use as a running example throughout the paper the e-commerce Web site for online computers shopping first described in [14]. A demo Web site implementing this example, together with its full specification, is provided at [1].

New customers can register a name and password (modeled as constants in the input schema), 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. 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
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

We illustrate only the search functionality of the laptop search page LSP (see the online demo of [1] for the full version, which also allows users to search for desktops).

`Page` LSP
`Inputs:` $\mathsf{laptopsearch}(ram, hdisk, display), \mathsf{button}(x)$
`Input Rules:`
$\text{Options}_{\mathsf{button}}(x) \leftarrow \quad x = \text{“}search\text{”} \vee x = \text{“}view\ cart\text{”} \vee x = \text{“}logout\text{”}$
$\text{Options}_{\mathsf{laptopsearch}}(r,h,d) \leftarrow \underline{\mathsf{criteria}}(\text{“}laptop\text{”}, \text{“}ram\text{”}, r) \wedge \underline{\mathsf{criteria}}(\text{“}laptop\text{”}, \text{“}hdd\text{”}, h) \wedge \underline{\mathsf{criteria}}(\text{“}laptop\text{”}, \text{“}display\text{”}, d)$
`State Rules:`
$\text{userchoice}(\mathsf{r,h,d}) \leftarrow \mathsf{laptopsearch}(r,h,d) \wedge \mathsf{button}(\text{“}search\text{”})$
`Target Web Page Rules:`
$\text{HP} \leftarrow \mathsf{button}(\text{“}logout\text{”})$
$\text{PIP} \leftarrow \exists r \exists h \exists d\ \mathsf{laptopsearch}(r,h,d) \wedge \mathsf{button}(\text{“}search\text{”})$
$\text{CC} \leftarrow \mathsf{button}(\text{“}view\ cart\text{”})$
`End Page` SP

Notice how the three buttons *search*, *view-cart*, and *logout* are modeled by a single input relation button, whose argument specifies the clicked button. The corresponding input rule restricts it to a search, view-cart, or logout button only. Since the user chooses at most one tuple among the displayed options, no two buttons may be clicked simultaneously.

Observe 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.

If the search button is clicked, the state rule records the user's pick of search criteria in the userchoice table. If this pick is non-empty, the second target rule fires and the Web site transitions to the PIP page. □

The properties we wish to verify range from basic soundness of the spec (e.g. the next Web page is always uniquely defined) to semantic properties (e.g. no product is shipped before the correct payment is received). Such properties are expressed in a variant of linear-time temporal logic, denoted LTL-FO. Properties of runs of a Web site $\mathcal{W}$ are defined by formulas using temporal operators such as **G, F, X, U, B**. For example, $\mathbf{G}p$ means that property p always holds; $\mathbf{F}p$ means that p eventually holds; $\mathbf{X}p$ holds at a given point in the run if p holds in the next configuration of the run; $p\ \mathbf{U}\ q$ holds at a given point if q holds sometime in the future and p holds until q becomes true; and $p\ \mathbf{B}\ q$ holds[1] if either q never holds, or it eventually does and p must hold sometime before q becomes true. Classical LTL formulae are built from propositional variables, using temporal and Boolean operators. The language LTL-FO describing properties of a spec $\mathcal{W}$ uses as building blocks FO formulas evaluated in a given configuration. Specifically, an LTL-FO formula is obtained by combining FO formulas by temporal and Boolean operators (but no further quantifications). The remaining free variables in the resulting formula are universally quantified at the very end. For example, the LTL-FO formula

$$(\dagger) \forall x \forall y \forall id [(\mathsf{pay}(id,x,y) \wedge \underline{\mathsf{price}}(x,y))\ \mathbf{B}\ \overline{\mathsf{ship}}(id,x)]$$

states that whenever item x is shipped to customer id, a payment for x in the correct amount must have been previously received from customer id.

[1] Our definition of **B** differs slightly from [29, 14].

Results in [29, 14] show that it is decidable in PSPACE whether a Web site spec $\mathcal{W}$ satisfies an LTL-FO formula φ, under a restriction called *input boundedness*. Input boundedness requires that all quantified variables range over values from user inputs, in all formulas used in the rules of the spec. Specifically, existential quantifications must be of the form $\exists x(\mathsf{R}(x, \bar{y}) \wedge \varphi)$ and universal quantifications of the form $\forall x(\mathsf{R}(x, \bar{y}) \rightarrow \varphi)$, where R is an input relation and φ a formula where x does not occur in state or action relations. Restricting the quantification range to inputs is quite natural, since the Web site is driven by user input. For example, the formula $\forall x \forall y[\mathsf{pay}(x, y) \rightarrow \underline{\mathsf{price}}(x, y)]$, stating that every payment received is in the right amount, is input bounded. The same restriction applies to the FO components of the LTL-FO property (but not to the last universal quantification applied to the entire formula). For instance, (†) above is input bounded. Finally, there is a restriction on input option definitions: these must be FO formulas using only existential quantifications, and state atoms cannot contain any variables. The Web page definitions in Example 2.1 are all input bounded, as is the entire Web site of the demo of [1]. We later exhibit other natural examples of input-bounded Web sites and properties.

The main theoretical result of [29, 14] that is relevant to us is the following.

THEOREM 2.2. *It is decidable in* PSPACE *if an input-bounded spec $\mathcal{W}$ satisfies an input-bounded LTL-FO formula φ.*

The proof is based on an ingenious reduction of the problem of whether $\mathcal{W}$ satisfies φ to the finite satisfiability problem for sentences in the logic E+TC (existential FO augmented with a transitive closure operator), which is in turn shown to be in PSPACE [29, 14].

2.2 Propositional LTL Model Checking

Our work builds upon on verification techniques developed in the mature field of computer-aided software verification [26]. Existing verifiers and model checkers apply to transition systems described by propositional predicates, and they check properties expressed in propositional temporal logics such as LTL. In particular, configurations of a transition system S are described by a set of propositional variables $\mathcal{P} = \{P_1, \ldots, P_n\}$. Each configuration corresponds to a truth assignment for $\mathcal{P}$. The system can transition from the current configuration to one of several successor configurations, according to a transition relation $\mathcal{T}_S$. $\mathcal{T}_S$ may be specified using various formalisms: either as a nondeterministic finite-state automaton A_S, or a propositional formula involving the current and successive values of $\mathcal{P}$, or a Kripke structure [26]. A run of S is an infinite sequence of configurations $C_0, C_1, \ldots$ such that $(C_i, C_{i+1}) \in \mathcal{T}_S$ holds for each $i \geq 0$.

Given a propositional LTL property φ and a transition system S, the associated model checking problem consists in checking that every run of S satisfies φ, or equivalently, that no run of S satisfies $\neg\varphi$. Pragmatic solutions were enabled by the seminal result of [30], which shows that each LTL formula ϕ can be compiled into a Büchi automaton A_ϕ which accepts precisely the runs that satisfy ϕ. This reduces the model checking problem to checking the existence of a run ρ of A_S which is accepted by $A_{\neg\varphi}$. To find ρ, one can employ the so-called *nested depth-first search (ndfs) algorithm* [10, 20]. Conceptually, the *ndfs* algorithm performs a systematic construction of runs of A_S. It begins in the start configuration of A_S and at each subsequent step it extends the run constructed so far by following possible transitions in A_S in a depth-first fashion. Run extensions leading to non-acceptance in $A_{\neg\varphi}$ are pruned. When no possible run extension remains, the algorithm backtracks. This algorithm is implemented in the widely used SPIN model checker [20]. We detail it next.

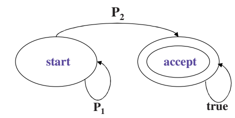

Figure 1: Büchi automaton for $\varphi^{aux} = P_1 \mathbf{U} P_2$

Büchi Automata. We present here the flavor of Büchi automata used in SPIN. A Büchi automaton A is a nondeterministic finite state automaton (NFA) with a special acceptance condition for infinite input sequences. The input alphabet consists of truth assignments for some given set of propositional variables $P_1, \ldots, P_n$. The transition relation $\mathcal{T}$ specifies triples (s_1, δ, s_2) where s_1, s_2 are states and δ is a propositional formula over $P_1, \ldots, P_n$. Intuitively (s_1, δ, s_2) states that A may transition from s_1 to s_2 if the current input is a satisfying assignment for δ. A run of A on a given infinite input sequence $a_0, a_1, a_2, \ldots$ is a sequence of states $s_0, s_1, s_2, \ldots$ such that s_0 is the start state and for each $i \geq 0$, there is some formula δ_i such that $(s_i, \delta_i, s_{i+1}) \in \mathcal{T}$ and a_i is a satisfying assignment for δ_i. A accepts an infinite input sequence IS if and only if there is a run of A on IS which visits some final state s_f infinitely often.

Example 2.3 Figure 1 shows the Büchi automaton for $P_1 \mathbf{U} P_2$. Notice that the accepted infinite input sequences consist of an arbitrary-length prefix of satisfying assignments for P_1, followed by a satisfying assignment for P_2 and continued with an arbitrary infinite suffix. □

Notice that any run for which some final state s_f is reached infinitely often must correspond to a path in A which starts at the initial state s_0, reaches s_f, and proceeds back to s_f. We shall call such a path a *"lollipop"* path, referring to its prefix from s_0 to s_f as the "stick", and to the cycle through s_f as the "candy" part. [10] introduces the *ndfs (nested depth-first search)* below, which searches for runs ρ of A_S that determine a lollipop path in $A_{\neg\varphi}$. $\mathcal{T}_{\neg\varphi}$ denotes the transition relation of $A_{\neg\varphi}$.

algorithm *ndfs*
stick(s_0, C_0) // s_0 is the start state of $A_{\neg\varphi}$,
        C_0 the start configuration of A_S

procedure *stick*(s, C_s)
record $< (s, C_s), 0 >$ as visited
for each successor C_t of C_s in A_S
    for each $(s, \delta, t) \in \mathcal{T}_{\neg\varphi}$ such that C_s satisfies δ
        if $< (t, C_t), 0 >$ not yet visited then *stick*(t, C_t)
        if t is final then
            base $:= (t, C_t)$;
            candy(t, C_t)

procedure *candy*(s, C_s)
record $< (s, C_s), 1 >$ as visited
for each successor C_t of C_s in A_S

for each $(s, \delta, t) \in \mathcal{T}_{\neg\varphi}$ such that C_s satisfies δ
 if $< (t, C_t), 1 >$ not yet visited then $candy(t, C_t)$
 else if $t = base$ then report run

Procedure *stick* performs a depth-first search for a prefix of a run in A_S which corresponds to the "stick" prefix of a lollipop path in $A_{\neg\varphi}$. When the search reaches a configuration C_t of A_S and a final state t in $A_{\neg\varphi}$, (the candidate for the *base* of the candy), it is suspended and a nested search is initiated to find an extension of the run in A_S which corresponds in $A_{\neg\varphi}$ to a cycle through t (the "candy" part of the lollipop path). If the nested search fails, the suspended search is resumed. The 0 and 1 flags serve to record that *stick*, respectively *candy* have already been called on arguments (s, C_s) unsuccessfully so the search can be pruned.

The remarkable achievement of algorithm *ndfs* is to check whether some *infinite* run satisfies $\neg\varphi$ by constructing only finitely many finite-length runs of A_S. These are precisely the runs of length upper bounded by $2N$, with N the product between the number of states of A_S and of A_φ. Indeed, observe that once the length of a run exceeds N, *stick* is invoked the second time with the same arguments. Since the search failed at the first invocation, it is guaranteed to fail at the second, and it can therefore be pruned. Similar reasoning yields that *candy* can extend the run unsuccessfully for at most another N steps until it calls itself with the same arguments.

3. DATA-AWARE VERIFICATION

Given the success of model checking techniques, it is natural to consider solving the Web application verification problem using existing model checkers. At first glance, there is a direct analogy between the two problems. Notice that all runs of a Web application $\mathcal{W}$ satisfy a property φ_0 if and only if there is no run ρ of $\mathcal{W}$ that satisfies $\varphi := \neg\varphi_0$. As in model checking, we could attempt to find ρ by using the *ndfs* algorithm. Upon closer inspection, the analogy fails.

Recall from Section 2.2 that in the propositional case, the *ndfs* search relies crucially on the fact that it suffices to inspect only finitely many runs of A_S. These are runs of length upper bounded by a constant, over configurations from a finite set. Both the length upper bound and the set of possible configurations are finite because the transition system has finitely many distinct configurations. This argument breaks down for Web applications given by FO specifications, since they have infinitely many possible configurations: recall that a part of the configuration is the underlying database, which is not known in advance and can be arbitrarily large.

We solve this problem via a series of successive refinements to an initial solution, spanning the spectrum from "decidable but impractical" to "feasible, with running times within seconds".

The first cut is based on results of [29, 14] implying that it is sufficient to inspect only finitely many runs of $\mathcal{W}$, namely those on a finite set of databases constructed over a domain **dom** which depends only on the specification and the property. This suggests a search along the lines of the *ndfs* algorithm: for each representative database over **dom**, the verifier would simply enumerate runs until configurations started repeating, while searching in parallel for a lollipop path in the property automaton. In the run construction, notice that for any given configuration, the next state, action, previous input and input options are uniquely determined by the appropriate rules. The only nondeterminism arises from the user's input choice. The algorithm can simply run these rules and generate a new successor configuration for each input choice. Unfortunately, the size of **dom** is exponential in the size of the specification and property, leading to a set of databases of doubly exponential cardinality. This is far removed from a practical solution: simply enumerating the necessary databases is infeasible, even ignoring the construction of the runs for each database.

Section 3.1 takes a crucial step towards a practical algorithm. It shows that it is not necessary to explicitly construct the entire underlying database in order to generate runs. Instead, at each step of the run it suffices to construct only those portions of the database, state and actions which can affect the page rules and property. We call the resulting sequence of partially specified configurations a *pseudorun*. The key advantage of pseudoruns is that their partially specified configurations have size polynomial in the application spec and property, thus yielding a PSPACE verification algorithm. While this is a significant improvement from the first cut algorithm which works in exponential space, it turns out to still be insufficient in practice.

The pseudorun-based search achieves practical relevance only with the aid of two heuristics (presented in Section 3.2) which dramatically improve the verification time without giving up soundness and completeness. The heuristics rely on a dataflow analysis to prune the partial configurations with tuples that are irrelevant to the rules and property. In our experimental evaluation, the new running times are of the order of a few seconds.

During the design of our pseudorun-based search we had to deal with the fact that LTL-FO differs from classical LTL by allowing FO rather than just propositional components. There are well-known public-domain tools such as `ltl2ba` (`www.spinroot.com`), that translate any propositional LTL property into a corresponding Büchi automaton, based on the algorithm described in [19]. To use such tools for LTL-FO formulas, we must first reduce them to propositional LTL properties. We do so by constructing, from an LTL-FO formula φ, a propositional LTL property φ^{aux} by replacing the FO components of φ with new propositional symbols. φ^{aux} can then be translated into a Büchi automaton $A_{\varphi^{aux}}$ using the `ltl2ba` tool. At every step of the search, we evaluate φ's FO components over the current configuration to determine the truth values of the propositional symbols in φ^{aux}, which yield the possible transitions in $A_{\varphi^{aux}}$.

Summarizing, the roadmap to our approach to verification is the following. Given Web application $\mathcal{W}$ and property $\varphi_0 \in$ LTL-FO, we check that all runs of $\mathcal{W}$ satisfy φ_0 by checking that no run satisfies $\varphi := \neg\varphi_0$. This involves the following steps.

1. Construct $\varphi^{aux} \in$ LTL by replacing the FO components of φ with new propositional symbols.
2. Construct $A_{\varphi^{aux}}$, the Büchi automaton accepting precisely the runs which satisfy φ^{aux} (using `ltl2ba`).
3. Execute a nested depth-first search which constructs the pseudoruns of $\mathcal{W}$, simultaneously navigating in $A_{\varphi^{aux}}$ by evaluating the FO components of φ to obtain the truth values of the propositional symbols in φ^{aux}. If the search finds no lollipop path in $A_{\varphi^{aux}}$, then return `yes`, otherwise return `no` and report the counterexample pseudorun. Pseudoruns are pruned according to the heuristics exploiting the dataflow analysis of the specification and property.

We now illustrate the first two steps of the approach, detailing Step 3 in Sections 3.1 and 3.2.

We discuss the construction of φ^{aux} first. If φ_0 is input-bounded, φ has general form $\exists\bar{x}\ \varphi_1(\bar{x})$, where $\bar{x}$ are the free variables of φ_1, and φ_1 contains only input-bounded quantifiers.

The set of FO components of φ_1, denoted $fr_{FO}(\varphi_1)$, consists of the maximal FO subformulas of φ_1, i.e. subexpressions which contain no temporal operators and are not nested within any FO subexpression of φ_1. For each $\varphi_i \in fr_{FO}(\varphi_1)$ we invent a fresh auxiliary propositional action variable P_i^{aux}, and obtain φ^{aux} by substituting φ_i with P_i^{aux} in φ_1.

Example 3.1 The following LTL-FO property referring to Example 2.1 states that any confirmed product must have previously been paid for.

$$
\begin{array}{ll}
\forall & pid, category, name, ram, hdd, display, price \\
& (\ \text{UPP} \wedge \mathsf{button}(\text{“}submit\text{”}) \wedge \text{cart}(pid, price) \\
& \wedge \underline{\mathsf{products}}(pid, category, name, ram, hdd, display, price)) \\
\quad \mathbf{B} & \\
& \overline{\mathsf{conf}}(pid, category, name, ram, hdd, display, price)
\end{array}
\tag{1}
$$

Payment is detected by the user clicking the “*submit*” button on the user payment page UPP, when the product of id *pid* is in the cart (modeled as a state relation). Notice how the price is checked against the price in the $\underline{\mathsf{products}}$ database table. The confirmation is modeled by inserting the appropriate tuple into the $\overline{\mathsf{conf}}$ action table.

Property (1) is negated to

$$
\begin{array}{ll}
\exists & pid, category, name, ram, hdd, display, price \\
& \neg(\ \text{UPP} \wedge \mathsf{button}(\text{“}submit\text{”}) \wedge \text{cart}(pid, price) \\
& \wedge \underline{\mathsf{products}}(pid, category, name, ram, hdd, display, price)) \\
\quad \mathbf{U} & \\
& \overline{\mathsf{conf}}(pid, category, name, ram, hdd, display, price)
\end{array}
\tag{2}
$$

which yields the propositional property φ^{aux}

$$P_1 \mathbf{U} P_2 \tag{3}$$

where P_1, P_2 are the new propositional symbols introduced for the FO formulae to the left, respectively right of the temporal operator $\mathbf{U}$ in (2). We have already seen in Figure 1 the Büchi automaton corresponding to Property (3). $\Box$

3.1 Searching for Pseudoruns

In this section we introduce an algorithm circumventing the explicit enumeration of representative databases. The algorithm is based on the key insight that it is not necessary to first materialize a full database in order to generate runs. Instead, it is sufficient to generate sequences of partially specified configurations by lazily making at each step just the right assumptions needed to obtain the next partially specified configuration. Let us call the partially specified configurations *pseudoconfigurations* and the resulting sequences of pseudoconfigurations *pseudoruns* (to be described in detail shortly). Pseudoruns have two important properties for input-bounded $\mathcal{W}$ and φ:

(i) φ is satisfied by some genuine run of $\mathcal{W}$ if and only if it is satisfied by some pseudorun on $\mathcal{W}$. Hence the search for a satisfying run can be confined to pseudoruns only.

(ii) Pseudoconfigurations can be constructed using a fixed domain of size polynomial in the size of the specification and property, yielding a PSPACE verification algorithm (as opposed to the first cut algorithm, which works in exponential space).

At each step, we construct pseudoconfigurations by picking an input, by assuming the presence of certain database tuples, and then computing the corresponding successor page, states and actions according to the page schema rules.

States and actions are only partially specified, in the sense that we only consider their tuples over a fixed domain, guided by the following intuition. Recall that the property φ has general form $\exists\bar{x}\ \varphi_1(\bar{x})$. To check that some run ρ of $\mathcal{W}$ satisfies φ, we need to check that we can assign to the existentially quantified variables $\bar{x}$ a vector of values $\mathbf{C}_\exists$, such that ρ satisfies $\varphi_1(\mathbf{C}_\exists)$. We denote by $\mathbf{C}_\mathcal{W}$ the set of constants occurring in $\mathcal{W}$. Since φ is input-bounded, all state and action atoms in $\varphi_1(\mathbf{C}_\exists)$ must be ground, i.e. they cannot contain variables, but only constants from $\mathbf{C}_\mathcal{W}$ or from $\mathbf{C}_\exists$. We denote $\mathbf{C} := \mathbf{C}_\mathcal{W} \cup \mathbf{C}_\exists$ and construct only pseudoconfigurations whose state and action relations contain only ground tuples over $\mathbf{C}$, since any other tuples cannot affect $\varphi_1(\mathbf{C}_\exists)$.

As is the case when constructing genuine runs, at every step we pick an input. For genuine runs, this input was drawn from the active domain of the underlying database (augmented with finitely many additional values accounting for the text input from users). In contrast, for pseudoruns, whenever we reach a page V, we pick the input from a fixed domain $\mathbf{C} \cup \mathbf{C}_V$ where $\mathbf{C}_V$ depends only on V, and is disjoint from $\mathbf{C} \cup \mathbf{C}_{V'}$ for all $V' \neq V$. The size of C_V is bounded by the total number of variables used in the input option rules of V (assuming the rules use disjoint sets of variables). Intuitively, this allows to represent one choice of input tuple from each input relation, together with witnesses to the existentially quantified variables in the input option rule satisfied by the tuple. It turns out (see Theorem 3.2 below) that we do not lose completeness by restricting our picks this way.

At step k of the pseudorun, we pick database tuples as follows. Since $\varphi_1(\mathbf{C}_\exists)$ is a sentence, all of its database atoms contain either constants from $\mathbf{C}$ or quantified variables. The input-boundedness restriction requires these variables to appear in some positive input or previous-input atom. Therefore, denoting with V_k the page at step k, we consider only database tuples over $\mathbf{C} \cup \mathbf{C}_{V_k} \cup \mathbf{C}_{V_{k-1}}$ as these are the only ones that may affect $\varphi_1(\mathbf{C}_\exists)$.

There is an important difference between $\mathbf{C}$ and the sets $\mathbf{C}_V$. The choice of database tuples using values in $\mathbf{C}$ must be consistent across pseudoconfigurations. Specifically, if at step k we assume that some tuple over $\mathbf{C}$ is present (or absent) in the database, we cannot assume the contrary at some other step. Intuitively, this is because the property φ can talk about such tuples and may therefore detect such inconsistencies. We therefore must fix the fragment of the database using values in $\mathbf{C}$ once and for all before the pseudorun is generated. We call this fragment the *core*, and denote by $\mathbf{cores}(\mathbf{C})$ the set of all instances using only constants in $\mathbf{C}$.

In contrast, it turns out that the assumptions we make about tuples outside the core that use constants in $\mathbf{C} \cup \mathbf{C}_{V_k} \cup \mathbf{C}_{V_{k-1}}$ do *not* have to be consistent across configurations. We call a subinstance containing only such tuples an *extension* to the core. The set $\mathbf{ext}(V_k)$ of possible extensions at page V_k is finite due to the finite domain. Extensions affect the property and rule atoms containing variables which also appear in input atoms (as is the case for input-boundedly quantified variables). Since extensions do not have to be consistent across pseudoconfigurations, the extension used at step k can be forgotten at step $k+1$, when a *new* extension can be picked. This non-obvious result is based on the following intuition. If for all k we replace the input values from $\mathbf{C}_{V_k} \cup \mathbf{C}_{V_{k-1}}$ with fresh values, the union of all database extensions and of the unique core yields some consistent, exponential-sized database D. Pseudoruns never explicitly materialize D. Instead, at every step they "slide" a polynomial-sized "window" over D.

Let $D_s, V_s, I_s, P_s, S_s, A_s$ be respectively the database, page, input, previous input, state and action of the current pseudoconfiguration C_s, and D_t the database of C_t, one of the successor pseudoconfigurations of C_s. To construct D_t, we keep the core of D_s, discard the extension of D_s, and pick an extension to complete D_t. The construction is detailed in procedure $succ_P$ below.

```
procedure succ_P
input: pseudoconfiguration C_s = ⟨D_s, V_s, I_s, P_s, S_s, A_s⟩
output: set of successor pseudoconfigurations of C_s
result := ∅
compute V_t by applying V_s's target rules on C_s
compute S_t by applying V_s's state rules on C_s
            and keeping only the tuples over C
P_t := I_s
// pick successor's partial database D_t :
let DBcore be the core of D_s
for each DBext ∈ ext(V_t)
    let D_t := DBcore ∪ DBext
    compute the input options by running
            V_t's input rules on D_t, P_t, S_t
    for each input choice I_t
        compute A_t by applying V_t's action rules on I_t, D_t, P_t, S_t
                    and keeping only the tuples over C
        result := result ∪{⟨D_t, V_t, P_t, I_t, S_t, A_t⟩}
return result
```

The following shows that it suffices to restrict the search for a run satisfying an input-bounded property to pseudoruns only. The proof is obtained by adapting to our framework the non-trivial proof of PSPACE complexity of model checking from [29, 14].

THEOREM 3.2. *If $\mathcal{W}$ and φ are input bounded, then φ is satisfied by some genuine run of $\mathcal{W}$ if and only if it is satisfied by some pseudorun of $\mathcal{W}$.*

Intuitively, we can think of a pseudorun as a concise representation of a large class of genuine runs. Working on pseudoruns speeds up the search since it amounts to inspecting the entire corresponding class at once rather than one run at a time.

Algorithm *ndfs-pseudo* below conducts a nested depth-first search for pseudoruns of $\mathcal{W}$ which determine a lollipop path in $A_{\varphi^{aux}}$. The algorithm enumerates all database cores and initiates an independent search for a satisfying pseudorun over each core. At each step of the search, both *stick* and *candy* attempt to extend the current pseudorun prefix and the current path prefix. In pseudoconfiguration C_s, the lollipop path prefix can be extended from state s to t only along a transition in $A_{\varphi^{aux}}$ i.e. only if there exists some propositional formula δ such that (s, δ, t) belongs to the transition relation $\mathcal{T}_{\varphi^{aux}}$ of $A_{\varphi^{aux}}$, and the truth values on C_s of φ's FO components satisfy δ.

Recall that since φ has the general form $\exists \bar{x}\ \varphi_1(\bar{x})$, these FO components may have free variables. Also recall that the domain of the cores and extensions depends on $\mathbf{C}_\exists$, the set of values assigned to the existentially quantified variables $\bar{x}$. These values need not necessarily be distinct from each other or from the ones in $\mathbf{C}_\mathcal{W}$. The *ndfs-pseudo* algorithm therefore considers all choices for $\mathbf{C}_\exists$, ranging from a subset of $\mathbf{C}_\mathcal{W}$ to a disjoint set of arbitrarily picked fresh constants.

```
algorithm ndfs-pseudo
// pick assignments for free variables in φ's FO components:
for each choice of C_∃
    instantiate the free variables of φ's FO components with C_∃
    C := C_W ∪ C_∃
    // construct the start pseudoconfigurations:
    let V_0 be the home page of W
    P_0 := ∅; S_0 := ∅
    for each DBcore ∈ cores(C)
        for each DBext ∈ ext(V_0)
            D_0 := DBcore ∪ DBext
            compute the input options by running
                    the input rules of V_0 on D_0, P_0, S_0.
            for each input choice I_0
                compute A_0 by running V_0's action rules on D_0, I_0,
                        P_0, S_0 and keeping only the tuples over C
                C_0 := ⟨D_0, V_0, I_0, P_0, S_0, A_0⟩
                let s_0 be the start state of A_φaux
                // search for pseudorun determining lollipop path:
                stick(s_0, C_0)

procedure stick(s, C_s)
record ⟨(s, C_s), 0⟩ as visited
evaluate φ's instantiated FO components on C_s
        to get truth values of auxiliary propositions P^aux
for each (s, δ, t) ∈ T_φaux such that P^aux satisfies δ
    for each C_t ∈ succ_P(C_s)
        if ⟨(t, C_t), 0⟩ not yet visited then stick(t, C_t)
        if t is final then
            base := (t, C_t); candy(t, C_t)

procedure candy(s, C_s)
record ⟨(s, C_s), 1⟩ as visited
evaluate φ's instantiated FO components on C_s
        to get truth values of auxiliary propositions P^aux
for each (s, δ, t) ∈ T_φaux such that P^aux satisfies δ
    for each C_t ∈ succ_P(C_s)
        if ⟨(t, C_t), 1⟩ not yet visited then candy(t, C_t)
        else if (t, C_t) = base then report pseudorun
```

THEOREM 3.3. *If $\mathcal{W}$ and φ are input-bounded, then algorithm* ndfs-pseudo *reports a pseudorun satisfying φ if and only if some run of $\mathcal{W}$ satisfies φ.*

The bound on the domains of the database cores and extensions picked by algorithm *ndfs-pseudo* enables the enumeration of pseudoruns in PSPACE. However, the resulting search space is exponential, and still too large in practice.

Example 3.4 In the online computer shopping example, the database schema contains 4 tables with arities 2, 3, 5 and 7. Even if the property had no prefix of universal quantifiers, thus yielding $\mathbf{C}_\exists = \emptyset$, $\mathbf{C}$ would contain 29 constants (page schema LSP from Example 2.1 alone features 7 constants). Algorithm *ndfs-pseudo* must therefore construct at least $2^{29^2+29^3+29^5+29^7} = 2^{17,270,412,688}$ cores. A similar analysis yields $2^{9,046,208,721}$ possible extensions. □

Algorithm *ndfs-pseudo* achieves practical relevance only in conjunction with the heuristics presented in Section 3.2.

3.2 Optimizations

As illustrated by Example 3.4, a major bottleneck in algorithm *ndfs-pseudo* is the construction of the numerous database cores and extensions. It turns out however that most of these are not needed. We have developed heuristics for pruning the sets of cores and extensions constructed by algorithm *ndfs-pseudo*. These heuristics slash the verification times to seconds while preserving the soundness and completeness of the algorithm.

The key intuitions behind our heuristics are the following.

Database cores keep track of the ground tuples whose presence or absence is checked by page rules and by the property. Ground tuples consist exclusively of constants and are detected by comparing all their attributes with constants. For instance, the home page schema HP of the demo site [1] authenticates users by testing for the presence of ground tuple user(name,password) in the database, where name and password are input constants provided by the user at login. However, the user attributes are never compared to other constants from the spec, such as "*login*", "*cancel*", "*logout*", etc. which play the role of button names. We developed a dataflow analysis which provides an upper bound on the potential comparisons to constants that may be performed throughout any run, explicitly or implicitly. Ground tuples which do not satisfy any potential comparison can satisfy neither membership tests nor absence tests. Therefore, they remain undetected and can be pruned from the core in the first place, leading to fewer cores to be inspected.

Similar observations apply to tuples in the extensions. The only way for rules or properties to check the presence/absence of these tuples is by comparing their attributes to constants or input values. Again, by means of dataflow analysis we identify all potential comparisons that may be performed during any run, and tuples which satisfy none of these comparisons can be safely dropped from the extension. This in turn restricts the number of extensions we need to construct in the first place. We detail our techniques next.

Heuristic 1 (Core Pruning) *Consider only core tuples for which each attribute A contains constants to which A is compared by the page rules or property.*

Example 3.5 Assume we want to verify Property (1) on the computer shopping application of Example 2.1. It turns out that among the underlying four database tables, two have at least one attribute which is compared to no constant whatsoever. For example, the third attribute of criteria, used on page LSP. By Heuristic 1, there are no tuples to consider for the cores of these tables, leaving only one choice, namely the empty core. A third table is products for which Property (2) of Example 3.1 compares the attributes to the constants in $\mathbf{C}_\exists$. Since there are no other comparisons in the specification, Heuristic 1 allows only at most one tuple for the core of products, yielding two cores: the empty core and the single-tuple core. Further analysis yields only four possible user cores, which together with the two products cores results in a total of 8 database cores, as opposed to the $2^{17,270,412,688}$ cores obtained without Heuristic 1. □

Dataflow Analysis for Potential Comparisons. We overestimate all potential comparisons of the A attribute of R-tuples to a constant c by performing the following straightforward dataflow analysis. Comparisons can be explicit, i.e. due to the occurrence in some rule or in the property of an R-atom containing c in the column corresponding to A. Comparisons can also be implicit. On one hand, they are due to the occurrence in an R-atom of a variable x in the A column, such that the equality $x = c$ follows by transitivity from the equality atoms in the rule or property. On the other hand, they are due to the A column of an R-tuple being copied to the B column of an S-tuple (S is a state table), such that the B attribute is itself (recursively) compared to c, explicitly or implicitly. This analysis is easily implemented by a recursive function which runs in linear time in the size of the property and specification.

Example 3.6 For an explicit comparison, see the second input rule of page LSP which compares the attributes of tuples in criteria to constants like "*laptop*", "*ram*", etc. To illustrate an implicit comparison, assume that the property contains the state atom userchoice("*1GB*", "*60GB*", "*21in*"). This results in a potential implicit comparison of the third attribute of criteria tuples to the constants "*1GB*","*60GB*" and "*21in*". This is because, by the input rule of page LSP, the laptopsearch input corresponds to the third attribute of several criteria tuples. These values are then copied by the state rule of page LSP into state userchoice, where they are finally compared by the property to the three constants. □

Example 3.4 shows that, even if we reduce the set of cores to a manageable size, we face a huge number of database extensions at each page schema. Fortunately, extensions can be pruned as well, using the following heuristic.

Heuristic 2 (Extension Pruning) *At page W, consider only extension tuples for which each attribute A contains constants or values of input tuple attributes to which A is compared by W's rules and by the property.*

Notice that by Heuristic 2, extensions are always empty for database tables not mentioned by the rules of page W.

Example 3.7 We consider the extensions at page LSP from Example 2.1. By Heuristic 2, for a database tuple to be in some extension, one of its attributes must be compared to the attribute of the button or laptopsearch input relation. This is not the case for any of the four database tables (three are not even mentioned by the rules of LSP, while criteria is not involved in comparisons to input variables). Heuristic 2 therefore leaves only one possible extension, namely the empty instance. Contrast this with the $2^{9,046,208,721}$ extensions obtained in Example 3.4 without Heuristic 2. □

We refer to our pruning strategies as "heuristics" because in the worst case they may not prune any cores or extensions. This would happen if all database attributes were compared to all constants and input attributes. However, we have observed that in practice, the opposite scenario prevails: each database attribute is compared to only a handful of constants, if any, and the impact of the heuristics is spectacular, indeed crucial in rendering algorithm *ndfs-pseudo* practical. By Theorem 3.8 below, this comes at no sacrifice of completeness.

THEOREM 3.8. *If we prune the database cores and extensions according to Heuristics 1 and 2, algorithm* ndfs-pseudo *remains sound and complete for input-bounded Web applications and properties.*

4. IMPLEMENTATION DETAILS

We designed our implementation to satisfy the following desiderata.

- Enumerate cores and extensions on demand, reusing space, to avoid exponential space consumption.
- Avoid to first enumerate a core or extension and to only afterwards check that it satisfies the pruning heuristics. Instead, directly generate only cores and extensions which are allowable under Heuristics 1 and 2.
- The representation of cores and extensions should be compatible with efficient checks of whether the search has previously visited a pseudoconfiguration.
- The page rules and property FO components should be efficiently evaluated over the pseudoconfigurations.

As detailed shortly, we decided to simultaneously use two distinct pseudoconfiguration representations in order to satisfy all requirements. We achieved the first three goals by representing pseudoconfigurations as bitmaps. For the fourth goal, we inserted pseudoconfigurations in a database management system and implemented the FO rules as SQL queries. This approach required extra attention to the efficient translation between representations.

Core and extension enumeration. We address the first two requirements as follows. Consider database cores first. For each k-ary table R, each attribute A_i can take a finite set of n_i constant values, as provided by the dataflow analysis. There are therefore $n_R := \Pi_{i=1}^{k} n_i$ possible tuples ($n_R = 0$ if some attribute A_j is compared to no constants, i.e. $n_j = 0$). We represent each tuple subset by a bitmap of n_R entries. For the entire database, we concatenate the individual table bitmaps. To enumerate all cores, we start with the all-zero bitmap and, treating the bitmap as the binary representation of an integer counter, we increment the bitmap at each call until we reach the all-one bitmap. We use an analogous encoding scheme for database extensions.

Detecting visited pseudoconfigurations. To address the third requirement, we extend the bitmap encoding scheme to pseudoconfigurations. The visited configurations are then stored in a trie data structure which allows updates and membership tests in time linear in the size of the bitmap.

Translation between representations. As described shortly, the efficient evaluation of page schema rules requires pseudoconfigurations to be stored as tables in a database management system (DBMS). We encode database tables into bitmaps as follows. Consider relation $R(A_1, \ldots, A_k)$ and let n_i be the number of constants assigned by our dataflow analysis to attribute A_i. Given an R-tuple $t = (c_1, \ldots, c_k)$, let r_i be the rank of constant c_i in the list of constants assigned to A_i by the dataflow analysis ($0 \leq r_i \leq n_i - 1$). Ranks are found efficiently by maintaining a hash table for each list. The index j of the bitmap bit corresponding to t is computed as $j = r_k + n_k \times (r_{k-1} + n_{k-1} \times (\ldots n_2 \times r_1))$. The translation from bitmaps to database tables relies on the following decoding of a bitmap bit index j into an R-tuple. We first obtain a tuple of ranks $(r_1, \ldots, r_k)$ according to $r_k = j \mod n_k$, $r_{k-1} = (j \text{ div } n_k) \mod n_{k-1}$, etc. The tuple of ranks is then translated into a tuple of constants by looking up constants by rank in their list. This is a constant time operation as lists are implemented as vectors.

Evaluation of page schema rules. Page schema rules are expressed as FO queries, also known as relational calculus, for which there is a standard translation to SQL [2]. The presence of universal quantifiers in the FO queries can lead to deeply nested SQL queries, which are notoriously hard to optimize and inefficiently evaluated. We obtain more efficient queries translating away all input-bounded quantifiers by exploiting the fact that at any step in the run input table I can contain at most one tuple t. Assume that the $empty_I$ flag is set to 1 if I is empty and to 0 otherwise. Then we can simplify rules of form $\forall \bar{x}\ I(\bar{x}) \rightarrow \varphi(\bar{x}, \bar{y})$ to $empty_I = 1 \vee \{t/\bar{x}\}\varphi(\bar{y})$. Similarly, $\exists \bar{x}\ I(\bar{x}) \wedge \varphi(\bar{x}, \bar{y})$ simplifies to $empty_I = 0 \wedge \{t/\bar{x}\}\varphi(\bar{y})$. Here, $\{t/\bar{x}\}\varphi(\bar{y})$ is obtained by substituting the constants from t for the variables $\bar{x}$, leaving only the free variables $\bar{y}$. Applying the simplifications recursively in $\{t/\bar{x}\}\varphi(\bar{y})$, we obtain FO formulae with no input-bounded quantifiers. These are translated to SQL queries without nesting. A beneficial side-effect of this simplification is the elimination of joins with input tables.

At verification time, we need to re-evaluate the obtained SQL queries for each distinct value of t and $empty_I$ assumed during the *ndfs-pseudo* search. To consistently achieve optimal query running time, we would need to let the DBMS optimizer re-optimize the query every time, especially since the underlying databases are updated frequently (due to state and action updates) leading to frequent changes of the optimizer's statistics. However, it turns out that each individual configuration typically corresponds to tables with very few tuples. The query workload generated by the *ndfs-pseudo* algorithm consists of a very large number of queries and updates, running over toy-sized databases. In this setting, the impact of optimization becomes negligible, subordinating the goal of ideal optimization to that of avoiding to repeatedly incur the overhead of sending a query to the database server and having it parsed, optimized and compiled to a query plan. The solution is to treat t and $empty_I$ as parameters and translate the FO queries to parameterized SQL queries implemented as JDBC prepared statements. At verification time, we only need to repeatedly fill in the actual parameter values.

Evaluation of FO property components. Recall that algorithm *ndfs-pseudo* tries all assignments to the free variables of the FO components of the property φ, in search for a satisfying one. An alternative, and more focused strategy is to *compute* the satisfying assignments, by treating FO components as non-boolean queries. Now suppose we attempt at step k to extend the lollipop path along a transition labeled with δ. For all FO components mentioned by δ we run the corresponding queries, obtaining individually satisfying assignments. The results are then *joined* to obtain satisfying assignments for δ. Moreover, we enforce consistency with the assignments computed at step $k-1$ by joining with them as well. Whenever the join result becomes empty, we prune the lollipop path search, as the current prefix is inconsistent with any assignment to the free variables.

Picking the right DBMS. One advantage of our architecture is that the verifier can work on top of any DBMS with a JDBC driver. The particular system we use is the open-source HSQLDB tool (`http://hsqldb.sourceforge.net`), which is a light-weight main-memory DBMS implemented in Java. We picked this tool to save the performance overhead imposed by disk-based persistence, a functionality which is irrelevant to the verifier. Our decision was based on an experiment measuring the time for inserting and deleting database cores into the DBMS. We used a schema of 4 tables of arities 2, 3, 5 and 7. We generated tables by picking all subsets of 6 tuples for each table, yielding 2^{24} distinct cores. We measured the average time to insert and delete a core, obtaining a speedup of two orders of magnitude by using HSQLDB (500 microseconds versus 50 milliseconds for Oracle). Another advantage of HSQLDB is its so-called "embedded" mode which allows direct calls to HSQLDB's JDBC API by the application, without requiring a client-server architecture. Since HSQLDB is fully implemented in Java, we can simply package our verifier code with the HSQLDB module into a single Java archive file.

5. EXPERIMENTS

We considered four experimental setups, **E1** through **E4** covering common types of Web applications. Their specifications were provided as proof of the expressivity of input-bounded specifications and are all available from [1]. We use these to evaluate our verifier.

E1: The online computer shopping application of our running example. It is demonstrated at [1] and offers functionality similar to that of the Dell Web site. The specification uses 19 page schemas, 4 database relations of arities between 2 and 7, 10 state relations of arities between 0 and 5, 6 input relations of arities between 1 and 5, 5 action relations of arities between 0 and 5, and 29 constants.

E2: A specification of a sports Web site modeled after the Motorcycle Grand Prix Web site `www.motogp.com`. The application allows users to browse the teams, pilots, motorcycle details, racing news, etc. It contains 15 page schemas, 7 database relations, no state or action relations. This example is representative for applications whose functionality is restricted to browsing, without internal state changes.

E3: An airline reservation site similar to part of the Expedia Web site. The specification models 22 pages using 12 database tables (arities up to 10), 11 state tables (arities up to 5), one action table (arity 1) and 31 constants.

E4: A specification of a book shopping application similar to the Barnes&Noble Web site. This specification was provided by the project members of the WebML Web building suite [9]. It models 35 pages, using 22 database tables (arities up to 14) 7 state tables (arities up to 6) and 22 constants.

Classes of Properties. In all experimental setups, the properties we verified were chosen to cover property types whose frequent occurrence in verification tasks earned them standard names (e.g., see [20]). They include the following.

Property type	Abbreviation	Syntactic shape
Sequence	T1	$p\mathbf{B}q$
Session	T2	$\mathbf{G}p \rightarrow \mathbf{G}q$
Correlation	T3	$\mathbf{F}p \rightarrow \mathbf{F}q$
Response	T4	$p \rightarrow \mathbf{F}q$
Reachability	T5	$\mathbf{G}p \vee \mathbf{F}q$
Progress (recurrence)	T6	$\mathbf{G}(\mathbf{F}p)$
Strong non-progress	T7	$\mathbf{F}(\mathbf{G}p)$
Weak non-progress	T8	$\mathbf{G}(p \rightarrow \mathbf{X}p)$
Guarantee	T9	$\mathbf{F}p$
Invariance	T10	$\mathbf{G}p$

For example, the sequence type states that p must hold before (**B**) q. Recurrence means that at each step in the run (**G**), property p will finally (**F**) hold at some subsequent step. Weak non-progress requires that at every step (**G**), if p holds, it also holds at the successor step (**X**). In [20], p and q stand for propositional formulae, but in our experiments we made them more complex, allowing them to be in LTL-FO.

The measurements. We chose lists of properties which both hold and fail on our specifications, because the verifier behaves differently in the two cases. If a property is violated, the verifier exits the search upon finding the first offending pseudorun. Satisfied properties force the verifier to explore the entire search space. For each property, we measured the running time of the verifier, and the maximal length of generated pseudoruns. To shed light on the memory consumption of the verifier, we counted the maximum number of visited pseudoconfigurations stored at any given time in the trie data structure. The experiments were performed on an Intel Pentium 4 (2.4 GHz) laptop with 256 MB RAM, running JDK 1.4.2 under Windows XP Home Edition.

Verification results for E1. The following table summarizes the results of verifying 17 properties $P_1, \ldots, P_{17}$.

Type	Property name	Time [seconds]	Max run* length	Max. trie size
T1	P5 (true)	4	11	268
	P7 (true)	2	6	168
T2	P9 (true)	1	10	70
T3	P10 (true)	.23	3	5
	P11 (false)	.29	11	12
	P12 (true)	.6	6	23
	P13 (false)	.44	13	21
T4	P14 (false)	.19	7	5
T5	P2 (true)	.9	8	45
	P3 (false)	.37	14	19
T6	P17 (false)	.15	5	5
T7	P15 (false)	.26	11	11
T8	P6 (false)	.49	15	30
T9	P1 (true)	.02	1	0
	P8 (false)	.11	6	4
T10	P16 (false)	.14	7	6
	P4 (true)	1	6	213

* Max length of run (if true) or of counterexample (if false)

Due to space constraints, we only detail a few properties.

P1: page HP is eventually reached in all runs: **F** HP. This is clearly true as HP is the start page. We picked this property as a minimum yardstick since it requires the generation of pseudoruns of length 1 only, thus measuring mainly the time needed to generate all starting pseudoconfigurations.

P4: At each step, there can be no two distinct successor pages. This property holds, and is expressed by a formula which lists for each page all pairs of its successor pages. We chose this property because of its size (12 **G** and 12 **X** operators), to study the impact of the size of the property automaton (30 states) on the running time.

P5: Any confirmed product was previously paid for. This is Property (1) from Example 3.1 and it requires more running time because the search space is larger, as the universally quantified variables lead to 7 values in $\mathbf{C}_\exists$ and therefore to more cores. Also, since the property holds, the entire search space is explored.

P7: An order must have status "ordered" before it can be cancelled: $\forall oid, uname, pid, price, status$
$(\underline{\text{orders-db}}(oid, uname, pid, price, \text{"}ordered\text{"})\mathbf{B}$
$(\text{ CCP} \wedge \text{userorderpick}(oid, pid, price, status)))$. The order

status is read from the orders-db database table. The order being cancellable corresponds to the user being on the customer cancel page CCP and the internal state recording the customer's pick of that order.

P9: If the user always clicks on a link at page EP, then whenever EP is reached, HP will eventually be reached as well: $\mathbf{G}(\text{ EP} \rightarrow \exists x \mathsf{clicklink}(x)) \rightarrow \mathbf{G}(\mathbf{G}(\neg \text{ EP}) \vee \mathbf{F}(\text{ EP} \wedge \mathbf{F}\text{ HP}))$. This turns out to be true since the only link on page EP leads to HP and since the page transition is not affected by the internal state of the application.

P12: If a product is eventually added into the cart, then the user must eventually view the details of this product: $\forall pid, price(\mathbf{F}\mathsf{cart}(pid, price) \rightarrow \mathbf{F}\mathsf{pick}(pid, price))$. The event of viewing the product details is detected by that product being picked by the user from a pull-down list modeled by the input relation pick.

P15: Every run must reach the error page EP and be trapped there forever: $\mathbf{F}(\mathbf{G}\text{ EP})$. Fortunately, this is false.

Verification results for E2. We verified 13 properties on this specification, again covering all types. We do not list the measurements in detail, as the results are similar to those for configuration **E1**. The measured times ranged from 20 milliseconds to 1 second. We measured maximum pseudorun lengths from 12 to 68. The maximum number of pseudoconfigurations coexisting in the trie ranged from 35 to 102. We illustrate only one property here.

If the circuit detail page CDP is reached, then either the grand prix page GP or the grand prix detail page GDP must have been reached previously and the user must have clicked the "circuits" button, respectively picked some circuit from a menu: $((\text{ GP} \wedge \mathsf{clickbutton}(\text{"}circuits\text{"})) \vee (\text{ GDP} \wedge \exists cid\ \mathsf{pick\_circuit}(cid)))\mathbf{B}(\text{ CDP})$.

Verification results for E3. We verified 14 properties on this specification, again covering all types. The measured times ranged from 680 milliseconds to 4 seconds for 13 of them. We measured maximum pseudorun lengths from 12 to 51. The maximum number of pseudoconfigurations coexisting in the trie ranged from 32 to 302.

Verification results for E4: the results obtained were similar, omitted due to space limitations.

Failure of Classical Tools

SPIN. Recall that the first cut algorithm sketched in Section 3 relies on verifying runs over a fixed domain, thus reducing the problem to verification of Web applications with a finite set of configurations. This raises the natural question whether our verification problem can be solved using standard finite-state model checkers such as SPIN [20]. To answer it, we investigated whether, given the Web application specification, the property and the size of the fixed domain, SPIN's built-in optimizations can prune the doubly exponential search space obtaining reasonable verification times. To this end, we modeled a SPIN process which generates tuples from the domain, introduces them into a database, and nondeterministically decides whether to repeat this activity or to stop and initiate runs over the constructed database, at each step nondeterministically making an input choice. The specification was written in SPIN's input language, Promela. We observed no pruning of the search space, whose explosion lead to a timeout of the experiment even for the simplest properties. We conclude that the pseudorun search combined with our heuristics are crucial for a feasible implementation.

PVS. We also attacked experimental setup **E1** using the PVS theorem prover for higher-order logic (`http://pvs.csl.sri.com/`). Its expressivity allowed us to easily express such temporal operators as **U** and **F**. This approach required frequent interaction at verification time with a highly expert user, as the prover would ask for advice on how to prove complex lemmas whose connection to the original specification was non-obvious.

6. RELATED WORK

The model of interactive, data-driven Web applications was introduced in [14], motivated by the earlier work in [29, 3]. It is related to WebML [9], a high-level conceptual model for data-driven Web applications, extended in [8] with workflow concepts to support process modeling. It is also related to other high-level workflow models geared towards Web applications (e.g. [7, 12, 33]), and ultimately to general workflows (see [32, 18, 13, 6, 31]), whose focus is however quite different. Non-interactive variants of Web page schemas have been proposed in prior projects such as Strudel [15], Araneus [25] and Weave [16], which target the automatic generation of Web sites from an underlying database. The WAVE tool complements automatic Web site generators with a verification capability.

Our work is related to the general area of automatic verification and particularly reactive systems [24, 23]. This body of work typically applies to propositional specifications, employing abstraction techniques to handle FO features, often giving up completeness in the process. A prominent example is Microsoft's SLAM project [4] for verification of safety properties (a subclass of LTL properties) for C code. SLAM combines model checking and abstraction refinement techniques and relies on a theorem prover which may require interaction from an expert user at verification time. The Stanford Temporal Prover STeP [5] combines deductive methods with algorithmic techniques to verify LTL properties of reactive and real-time systems. Like SLAM, STeP is not complete and it resorts to interactive theorem proving for properties that cannot be proven autonomously.

The proof of Theorem 3.2 is an adaptation of the PSPACE decidability results from [28, 29] and of its extension in [14] to Web application verification. However, [28, 29, 14] reduce the problem to deciding satisfiability of sentences in a transitive-closure logic. The latter problem is solved for arbitrary sentences, missing the optimization opportunities provided by the structure of the specification and property. Moreover the PSPACE complexity bound does not demonstrate the feasibility of implementing a verifier. This is precisely the contribution of our work, achieved by combining database optimization techniques with automata-theoretic model-checking techniques developed in [30, 10].

Other work on formal verification of Web services was done in [27]. The results of [27] correspond in our framework to the verification of a fragment of LTL properties of runs on a *fixed* database only. The verification of *e-compositions* of collections of individual Web service peers is discussed in [21, 22]. E-compositions are specified using the standard BPEL4WS [11]. An e-composition run is a sequence of message types passed between peers during a valid execution. An open problems listed in [21] is the verification of properties that pertain not just to the type of the messages but also to data. [17] deals with data present as message parameters by translating the specification of the e-composition to Promela, the input language of SPIN.

7. DISCUSSION

Our experience with the WAVE tool in the context of data-driven Web sites demonstrates that complete verification is practically feasible for a reasonably broad class of applications. The verification times we achieved, on the order of seconds, are surprisingly good. To put this in perspective, incomplete verification of software and digital circuits often takes days, even after abstraction. This is considered acceptable for off-line verification of systems or circuits that will be deployed for months, perhaps years. Our surprisingly good performance results suggest that interactive applications controlled by database queries may be unusually well suited to automatic verification. They also suggest that the marriage of classical model checking and database optimization techniques can be extremely effective. This is significant both to the database area and to automatic verification in general.

While completeness is lost for non-input-bounded applications or properties, WAVE can be easily adapted for use as an incomplete verifier for such applications. To this end, WAVE needs to be modified as follows. Whenever a candidate pseudorun counterexample to the property is produced in the course of the ndfs search, WAVE needs to check that this in fact corresponds to a genuine run violating the property. If it does not, the ndfs search is reactivated. As expected, this yields a sound but incomplete verification tool. This approach can be coupled with various abstraction techniques, as commonly done in software verification.

One practical obstacle to the use of tools such as WAVE is that LTL-FO properties can be difficult to specify by non-expert users. Therefore, user-friendly interfaces for the specification of properties to be verified are desirable. For specific classes of temporal properties, such as safety or liveness properties, such interfaces can be easily devised. A more intricate but very common type of temporal property is implicit in workflow specifications, for which user-friendly, visual specification tools already exist. For instance, the BPMN constraints imposed on WebML workflow specifications can be automatically translated to temporal logic (see `http://www.webml.org/LTLverification/`). Furthermore, workflows are quite expressive: they can simulate all temporal operators except $\mathbf{X}$ (intuitively, $\mathbf{X}$ is a problem in general because the concurrency allowed by workflows leads to interleaving of activities, making "next state" constraints unenforceable).

8. REFERENCES

[1] 2004. http://www.cs.ucsd.edu/~lsui/project/index.html.

[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] T. Ball and S. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL*, 2002.

[5] N. Bjorner, A. Browne, E. Chang, M. Coln, A. Kapur, Z. Manna, H. Sipma, and T. E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In *Computer Aided Verification (CAV)*, 1996.

[6] A. Bonner and M. Kifer. An overview of transaction logic. *Theor. Comput. Sci.*, 133(2), 1994.

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

[8] 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.

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

[10] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. In *Computer Aided Verification (CAV)*, 1990.

[11] 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.

[12] 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.

[13] H. Davulcu, M. Kifer, C. Ramakrishnan, , and I. V. Ramakrishnan. Logic based modeling and analysis of workflows. In *PODS*, 1998.

[14] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web services. In *PODS*, 2004.

[15] 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.

[16] D. Florescu, K. Yagoub, P. Valduriez, and V. Issarny. WEAVE: A data-intensive web site management system(software demonstration). In *(EDBT)*, 2000.

[17] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *WWW'04*, 2004.

[18] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. In *Distributed and Parallel Databases*, 1995.

[19] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Int. Conf. on Protocol Specification, Testing and Verification (PSTV)*, 1995.

[20] G. Holzmann. *The Spin Model Checker – Primer and Reference Manual*. Addison-Wesley, 2003.

[21] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: a look behind the curtain. In *PODS*, 2003.

[22] R. Hull and J. Su. Tools for design of composite web services (tutorial). In *PODS*, 2004.

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

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

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

[26] S. Merz. Model checking: a tutorial overview. 2001.

[27] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of Web services. In *WWW*, 2002.

[28] M. Spielmann. Abstract State Machines: verification problems and complexity. Ph.D. thesis RWTH Aachen 2000.

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

[30] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symp. on Logic in Computer Science*, 1986.

[31] D. Wodtke and G. Weikum. A formal foundation for distributed workflow execution based on state charts. In *ICDT*, 1997.

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

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