UNIVERSITY OF CALIFORNIA, SAN DIEGO

Specification and Verification of Interactive Data-Driven Web Applications

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Liying Sui

2006

The dissertation of Liying Sui is approved, and it is acceptable in quality and form for publication on microfilm:

[illegible]

[illegible]

[illegible]

[illegible] Chair

University of California, San Diego

2006

To my mother Shufang, who never had a chance to finish her academic career, and my father Huankai, who was the first to inspire me to pursue my Ph.D.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I would like to express my gratitude to my advisors Professor Victor Vianu and Professor Alin Deutsch for their constant support and motivation. They have offered invaluable advice and instruction to me on identifying problems, conducting research, and fine tuning solutions. Their diligence and commitment to science have been and will be a great influence on me for many years to come. I am glad I had the opportunity to learn from them and to work with them. I am grateful to Professor Yannis Papakonstantinou and Professor Bertram Ludäscher, whose patience, time and feedback in this endeavor will be always appreciated. I would also want to thank Professor Jeff Remmel and Professor Hans Wenzl for serving in my committee and helping me with my dissertation.

I have learned much from my fellow graduate students and colleagues in Database Lab. I thank Dayou Zhou who has worked together with me on the WAVE project. Throughout my PH.D research, I also benefited from the discussions and collaboration with many brilliant people. Also, I would like to express my thankfulness to all my wonderful lab mates Emiran, Nick, Yannis, Guilian, Yu, and Pratik for making the workspace quite fun through lively and interesting discussions.

I wish to extend my deepest gratitude to my parents, for their years of hard work and dedication. Last, but most importantly, no words can express my gratitude to my husband, Xin Liu, for his unrelenting support, understanding and love.

VITA

1992–1996	B.S. in Computer Science Shandong University, Jinan, P.R.China
1996–1999	M.E. in Computer Engineering Institute of Computing Technology, Beijing, China
1999–2006	Ph.D in Computer Science University of California, San Diego

PUBLICATIONS

Alin Deutsch, Liying Sui, Victor Vianu, Dayou Zhou. *Verification of Communicating Data-Driven Web Services.* Accepted by ACM Symposium on Principles of Database Systems (PODS) 2006.

Alin Deutsch, Liying Sui, Victor Vianu, Dayou Zhou. *A System for Specification and Verification of Interactive, Data-driven Web Applications(Demo).* Accepted by ACM International Conference on Management of Data (SIGMOD), 2006.

Alin Deutsch, Monica Marcus, Liying Sui, Victor Vianu and Dayou Zhou. *A Verifier for Interactive, Data-Driven Web Applications.* ACM International Conference on Management of Data (SIGMOD), 2005.

Marco Brambilla, Alin Deutsch, Liying Sui and Victor Vianu. *The Role of Visual Tools in a Web Application Design and Verification Framework: a Visual Notation for LTL Formulae.* International Conference on Web Engineering(ICWE) 2005.

Alin Deutsch, Liying Sui and Victor Vianu. *Specification and Verification of Data-driven Web Services.* ACM Symposium on Principles of Database Systems (PODS) 2004:71-82. Full paper invited to special issue of J. of Computer and System Sciences, to appear.

ABSTRACT OF THE DISSERTATION

Specification and Verification of Interactive Data-Driven Web Applications

by

Liying Sui
Doctor of Philosophy in Computer Science

University of California, San Diego, 2006

Professor Victor Vianu, Chair

We study data-driven Web applications provided by Web sites interacting with users or applications. The Web site can access an underlying database, as well as the state information updated as the interaction progresses, and receives user inputs. 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.

My research establishes that under what conditions, automatic verification of such properties is possible, and reveals the complexity of verification. This brings into play a mix of techniques from logic to automatic verification.

In addition, we also present WAVE, a verifier we implemented for interactive, database-driven Web applications specified using high-level modeling tools such as WebML, to demonstrate that our solution is indeed practically feasible. WAVE is a sound and complete verifier for a broad class of applications and temporal properties. Based on our experiments on four representative data-driven applications and a battery of common properties, WAVE yielded surprisingly good verification time, on the order of seconds. This suggests that interactive applications controlled by database queries may be unusually well suited to automatic verification.

The experimental results 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 not only to the database area, but also to the automatic verification in general.

Chapter 1

Introduction

Web applications interacting with users or programs while accessing an underlying database are increasingly common. They are widely used in e-commerce sites, scientific and other domain-specific portals, e-government, and data-driven Web services. Interactive Web applications provide access to information as well as transactions, and 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 [55], to the high-level workflow specifications (e.g., see [11, 9, 15, 54, 56, 35]).

For example, e-commerce applications use intricate workflows(“business models”) to specify a protocol describing exchanges among partners in a transaction. Typically, this occurs in a data-intensive fashion, with many agents interacting with a Web application, simultaneously. Therefore, it makes sense to approach such applications with a database lens, in order to integrate the data and workflow aspects.

As revealed in [30], the design of even a simple interactive Web application is error prone. For critical Web applications, where multi-million dollars transactions are carried out every day, any design error can potentially cause great losses, thus ad-hoc repairs after failure are not acceptable. Hence, it is desirable to statically ensure the correctness of Web applications design before deploying it on the web.

The objective of this research is to develop static analysis techniques to increase confidence in the robustness and correctness of complex Web applications as well as to implement an automatic verifier which can verify whether a Web application design satisfies a set of design properties.

This dissertation presents new verification techniques for Web applications, and investigates the trade-off between the expressiveness of the Web application specification language and the feasibility of verification tasks. We further describe an automatic verifier—WAVE —that we have implemented which can check properties of WebML-style specifications, and is sound and 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).

Static analysis techniques have a number of advantages over traditional approaches based on simulation, testing, and deductive reasoning. They are "static" because the checking is performed without running the program. In particular, given an application and its desired properties, static analysis techniques automatically and usually conduct quite fast an exhaustive exploration (either explicitly or symbolically) of all possible behaviors of the application. It gives designers absolute confidence when desired properties are verified. In addition, when a desired property is not satisfied, static analysis techniques generate a counterexample, which shows, step by step, how the property is violated by the application. The exact error trace reported by static analysis techniques is a big advantage over testing, because the error trace provides invaluable information for designers to understand and remove bugs. Static checking can improve software productivity because the cost of correcting an error is reduced if it is detected earlier.

In order to do static analysis, a high-level way to specify the interactive Web applications is needed. The spread of interactive Web applications has been accompanied by the emergence of tools for their high-level specification. A representative, commercially successful example is WebML [11, 10], which allows to specify a Web application using an interactive variant of the E-R model augmented with a work-flow formalism. The code for the Web application is automatically generated from the WebML specification. Such approach not only allows fast prototyping and improves programmer productivity; it also provides new opportunities for the automatic verification of Web applications. Our specification language is inspired by WebML.

In the scenario we consider here, a data-driven Web application 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 application 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 are determined dynamically by querying the underlying database, as well as the state. The actions taken by the Web application, and transitions from one Web page to another, are determined by input, state, and database. We model the queries used in the specification of the Web application as first-order queries (FO), also known as relational calculus. FO can be viewed as an abstraction of the data manipulation core of SQL.

The properties we wish to verify about Web applications involve the sequences of inputs, actions, and states that may result from interactions with a user. This covers a wide variety of useful properties, ranging from basic properties, such as soundness of the specification to complex semantic properties. For example, in a Web application supporting an e-commerce application, it may be desirable to verify semantic properties. For instance, no product is delivered before payment of the right amount is received. Or, we may wish to verify that basic specification soundness properties, like the specification of Web page transitions, is unambiguous, (the next Web page is uniquely defined at each point during the execution), and navigation properties, such as 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 conditions on the sequence of inputs, actions, and states. To describe such properties we use a variant of linear-time temporal logic LTL-FO. 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-FO and CTL*-FO.

Our results identify classes of Web applications, for which temporal properties in the above temporal logics can be checked, and establish their complexity.

For linear-time properties, the restriction needed for decidability essentially imposes a form of guarded quantification in formulas used in the specification of the Web application and the property. This is similar to the "input boundedness" restriction, first introduced by Spielmann in the context of ASM transducers [48, 49]. With this restriction, verification of linear-time properties is PSPACE-complete for schemas with fixed bound on the arity, and EXPSPACE otherwise. As justification for the input-boundedness restrictions, we show that even slight relaxations of our restrictions lead to undecidability of verification. Thus, our decidability results are quite tight.

We now informally describe the restrictions on the Web application specifications and properties that guarantee soundness and completeness, known as *input boundedness* [49, 21]. Since we model the queries used in the specification of the Web application as first-order queries (FO), 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.

In terms of expressiveness, it turns out that many practically relevant Web applications can be modeled with input-bounded specifications. For example, we have shown that significant portions of a Dell computer shopping Web site, Expedia, Barnes and Noble, and a Grand Prix motor racing web site can be specified within the restricted framework (see http://www.cs.ucsd.edu/∼lsui/project/index.html for a demo).

Furthermore, we have implemented a verifier, WAVE[20, 18], which can check linear-time temporal properties of WebML-style specifications. When checking linear-time temporal properties, the task of a verifier is to check that all runs of the Web application satisfy the given properties(as usual in verification, runs are considered to be infinite). The verifier search for counter-examples to the desired property, i.e. runs leading to a violation. A verifier is *sound* if the counter-example it find is guaranteed to be a real counter-example, and it is *complete* if it is guaranteed to find a counter-example whenever one does exist. WAVE is *sound* and *complete* under *input boundedness* restrictions described above.

For branching-time properties, the restrictions needed for decidability are considerably more stringent, and the complexity of verification ranges from PSPACE to 2-EXPTIME, depending on the restriction.

In the broader context of verification, a data-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 [34] 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 received. 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, since 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 sound and complete verification of a significant range of Web applications is well within reach.

Soundness and completeness of verification is only guaranteed under *input boundedness* restrictions described above. Note that, both soundness and completeness is lost for non-input-bounded applications or properties; however, WAVE can still be used as an incomplete but sound verifier, as it is typically done in software verification, if the specification stay input-bounded and abstract out arithmetic or other predicates with restricted semantics, as black-box relations. The heuristics we developed remain just as effective in this case.

Our theoretical results [21] show the decidability of model checking for input-bounded specifications and properties, by pseudorun techniques we described in Section 4.1.1(which also provides the basis for the implementation of a practical verifier for linear time properties, described in [20, 18]). The complexity of checking that a Web application 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, the first difficulty facing a verifier is that exhaustive exploration of all possible runs of a Web application $\mathcal{W}$ 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 [49] 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, which bring the verification times in our experiments down to seconds.

In summary, our results identify classes of Web applications 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. To show that the theoretical result is indeed practically feasible, we have implemented WAVE (Web Application VErifier). The main contribution of WAVE is an extension of finite-state model checking techniques to data-aware reactive systems, in general, and data-driven interactive Web applications, in particular.

1.1 Organization

The rest of the dissertation is organized as follows. Chapter 2 recalls some notions and terminologies from logic and modeling checking, as well as related works. A simple and abstract specification language for data-driven interactive Web application is introduced in Chapter 3. Chapter 4 presents the theoretical results of the verification of temporal properties and shows that under certain restriction, the verification problem is decidable and the restriction is so tight that even small relaxations will lead to undecidability. WAVE , a verifier we implemented for interactive, data-driven Web applications is presented in chapter 5. Chapter 6 concludes. A demo of WAVE, as well as some running examples, is available at `http://www.cs.ucsd.edu/~lsui/project/index.html`.

Chapter 2

Preliminaries and Related Works

In this chapter, we recall some notations and terminologies from logic, and provide a short introduction to model checking and the verification problem. Readers familiar with logic and model checking may skip the first part of this chapter and refer back to it as needed. In addition, some related works are also presented in this chapter.

2.1 Logic

Vocabularies. A *Vocabulary* Υ is a set of relation and function symbols. Each symbol in Υ is associated with a natural number, called the *arity* of the symbol. Symbols of arity 0 are frequently referred to as *nullary symbols.* A *boolean* (resp. *constant*) symbol is a nullary relation (resp. function) symbol. If not mentioned otherwise, it is tacitly assumed that every vocabulary is finite and contains (at least) the two constant symbols 0 and 1. A *relational vocabulary* is a vocabulary without function symbols of arity > 0. In particular, **relational vocabularies may contain constant symbols.**

Structures. Let Υ be a vocabulary. A *structure* $\mathcal{A}$ over Υ consists of a nonempty set A, an interpretation $R^{\mathcal{A}} \subseteq A^k$ for every k-ary relation symbol $R \in \Upsilon$, and an interpretation $f^{\mathcal{A}} : A^k \rightarrow A$ for every k-ary function symbol $f \in \Upsilon$. The set

A is also called the *universe* of $\mathcal{A}$. A *finite structure* is a structure whose universe is finite. The class of finite structures over Υ is denoted by $\text{Fin}(\Upsilon)$. We assume that every structure $\mathcal{A}$ over Υ satisfies the following conditions:

- If Υ contains the symbols 0 and 1, then $0^{\mathcal{A}} \neq 1^{\mathcal{A}}$.
- If Υ contains the binary relation symbol $<$, then $<^{\mathcal{A}}$ is a linear order on A, in which case $\mathcal{A}$ is called an *ordered structure*.

2.1.1 First-Order Logic

By FO we denote *first-order logic* with equality. For a logic L and a formula ψ we write $\psi \in L$ to indicate that ψ is a formula of logic L. The set of formulas $\psi \in L$ over a particular vocabulary Υ is denoted by $L(\Upsilon)$.

For every $\psi \in FO$, $\text{free}(\psi)$ denotes the set of *free variables* of ψ. Sometimes we write $\psi(x_1, x_2, \ldots, x_k)$ to indicate that variables $x_1, x_2, \ldots, x_k$ are pairwise distinct and may occur free in ψ, without implying that $\{x_1, \ldots, x_k\} \subseteq free(\psi)$ or $free(\psi) \subseteq \{x_1, \ldots, x_k\}$. If t and t' are two terms, then $\psi[t/t']$ stands for the formula obtained from ψ by replacing every occurrence of t in ψ with t'. The notation $\psi[./.]$ is also used to denote substitutions of formulas.

2.1.2 Satisfiability Problem of Formulas

Propositional formulas are expressions built over an infinite set of propositional variables $\alpha, \beta, \ldots$ using unary negation symbol $\neg$ and binary connectives $\wedge$ and $\vee$. When the quantification $(\forall, \exists)$ over propositional variables is allowed, the expressions are called Quantified Boolean Formulas(QBF). The satisfiability problem (SAT for short) for propositional formulas is NP-complete, and for quantified boolean formulas is PSPACE-complete[46].

Sometimes one also adds function symbols allowing for building more complex atomic formulas, and equality symbol. Although the SAT problem for first-order formulas is undecidable, in the general case, some special cases are decidable.

A FO formula Ψ is in Bernays-Schönfinke prefix class (sometimes denoted $\exists^*\forall^*$), if it is of the form

$$\Psi = \exists p_1 \ldots p_i \forall q_1 \ldots \forall q_j \Phi$$

that is, it is in prenex form where all existential quantifiers precede all universal quantifiers, and the formula Φ is built using only constants and predicate letters, with no function symbols or equality. The problem of deciding if such a formula is satisfiable is NEXPTIME-complete[46].

2.1.3 Temporal Logic

Temporal logic is an extension of conventional (propositional/first-order) logic, which incorporates special operators that cater to time. With temporal logic, one can specify (and verify) how components, protocols, objects, modules, procedures and functions behave, as time progresses. The specification is done with (temporal) logic statements that make assertions about properties and relationships in the past, the present, and the future.

Temporal logic comes in two varieties: linear-time temporal logic assumes that at any moment, there is only one possible future moment; on the other hand, branching-time temporal logic treats time in a branching, tree-like way, at each moment, time may split into alternative courses representing different possible futures. The temporal modalities of the temporal logic usually reflect the character of time assumed in the semantics. Thus, in linear-time temporal logic, temporal modalities are provided for describing events along a single time line, and are regarded as specifying the behavior of a single computation of a system. In contrast, in branch-time temporal logic, the modalities reflect the branching nature of time by allowing quantification over possible futures, each describing the behaviour of the possible computations of a nondeterministic system.

Linear-time Temporal Logic

Linear-time Temporal Logic (LTL) is an extension of propositional logic to include discrete time information. Formulas are interpreted as referring to events along an infinite path of time points.

Definition 2.1. The language LTL(propositional linear-time temporal logic) is obtained by closing propositional logic under negation, disjunction, and the following formula formation rule: If φ and ψ are formulas, then $\mathbf{X}\psi$("next time") and $\psi\mathbf{U}\varphi$ ("until") are formulas.

LTL formulas are evaluated over a computation, i.e., an infinite sequence of states. A computation is called model of LTL. Let $\rho = \{s_i\}_{i\geq 0} = s_0, s_1, s_2, \ldots$ be such an infinite sequence of state, and let $\rho_{\geq j}$ denote $\{s_i\}_{i\geq j}$. We say $\rho_{\geq j} \models \psi$ to mean ψ holds at position j of the model ρ. In other words, ψ holds in the state at the j-th step of the computation ρ. Let us define the semantics of LTL formulas inductively on the structure of the formulas:

- $\rho_{\geq j} \models \psi$ iff ψ holds in s_j. if ψ is an atomic formula
- $\rho_{\geq j} \models \neg\psi$ iff it is not the case $\rho_{\geq j} \models \psi$
- $\rho_{\geq j} \models \psi \vee \varphi$ iff $\rho_{\geq j} \models \psi$ or $\rho_{\geq j} \models \varphi$
- $\rho_{\geq j} \models \mathbf{X}\psi$ iff $\rho_{\geq j+1} \models \psi$
- $\rho_{\geq j} \models \psi\mathbf{U}\varphi$ iff for some k≥j, $\rho_{\geq k} \models \varphi$ and for every i, $j \leq i < k$, $\rho_{\geq i} \models \psi$

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

Branching-time Temporal Logic

In branching-time temporal logic, the underlying structure of time is assumed to have a branching tree-like nature, where each moment may have many successor moments. The structure of time thus corresponds to an infinite tree.

We provide the formal syntax and semantics for two representative systems of propositional branching-time temporal logics. The simpler logic, CTL(Computational Tree Logic)[23] allows basic temporal operators of the form: a path quantifier—either **A**("for all future") or **E** ("for some future")—followed by a single one of the usual linear temporal operators **G**, **F**, **X**, **U** or **B**. It corresponds to what one might naturally first think of as a branching-time logic. However, CTL's syntactic restrictions significantly limit its expressive power. Therefore we also consider the much richer language CTL^*, which is sometimes informally referred to as full branching-time temporal logic. The logic CTL^* extends CTL by allowing the path quantifier (A or E) followed by an arbitrary linear time formula, allowing boolean combinations and nesting, over **F, G, X, U** and **B**.

Definition 2.2. [23] The set of CTL* formulas is the set of state formulas defined inductively together with the set of path formulas as follows:

1. each propositional formula over the vocabulary is a state formula;
2. if φ and ψ are state formulas then so are $\varphi \wedge \psi$, $\varphi \vee \psi$, and $\neg\varphi$;
3. if φ is a path formula, then Eφ and Aφ are state formulas;
4. each state formula is also a path formula;
5. if φ and ψ are path formulas then so are $\varphi \wedge \psi$, $\varphi \vee \psi$, and $\neg\varphi$;
6. if φ and ψ are path formulas then so are $\mathbf{X}\varphi$ and $\varphi\mathbf{U}\psi$.

The set of CTL formulas over vocabulary is defined by replacing (4)-(6) above by the rule:

- if φ and ψ are state formulas then $\mathbf{X}\varphi$, and $\varphi\mathbf{U}\psi$ are path formulas.

The semantics of the temporal operators is the natural extension of LTL with path quantifiers. 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". More formally, satisfaction of a CTL$^{(*)}$ sentence is defined using the tree.

We now review the classical notion of satisfaction of a CTL$^{(*)}$ formula by a Kripke structure(which can be viewed as infinite trees labeled with propositional variables) (see[23]).

Definition 2.3. [42] Let AP= $\{p_1, p_2, \ldots, p_n\}$ be a finite set of atomic propositional symbols. A *Kripke structure* over AP is a 4-tuple K=(S,s_0,R,L) where:

- S is a finite set of states.
- $s_0 \in S$ is an initial state.
- R is a total binary relation on S ($R \subseteq S \times S$), called the *transition relation.*
- $L\colon S \rightarrow 2^{AP}$ assigns to each state the set of atomic propositions which are true in that state.

A *path* ρ in Kripke structure K is an infinite sequence of states $(s_0, s_1, \ldots)$ such that $(s_i, s_{i+1}) \in R$ for every $i \geq 0$. Let $\rho_{\geq i}$ denote the suffix path $(s_i, s_{i+1}, s_{i+2}, \ldots)$. The notation $K, s \models p$ indicates that a CTL* state formula p holds at state s of the Kripke Structure K. Similarly, $K, \rho \models \psi$ indicates that a CTL* path formula ψ holds at a path of ρ of the Kripke Structure K. We write $s \models p$ or $\rho \models \psi$ when it is obvious, from the context, which structure it is.

The notion of satisfaction of a CTL* formula by a Kripke structure is defined as follows:

1. $s \models p$ iff $p \in L(s)$.
2. $s \models \neg p$ iff $p \notin L(s)$.

3. $s \models \varphi_1 \wedge (\vee) \varphi_2$ iff $s \models \varphi_1$ and(or) $s \models \varphi_2$.
   $s \models \mathbf{E}\psi$ iff there exists an infinite path $\rho' = (s, s_1, s_2, \ldots)$ in K, starting from s, such that $\rho' \models \psi$.
   $s \models \mathbf{A}\psi$ iff for every infinite path $\rho' = (s, s_1, s_2, \ldots)$ in K starting from s, $\rho' \models \psi$.

4. $\rho_{\geq j} \models \psi$ iff $s' \models \psi$ where s' is the first state in $\rho_{\geq j}$.

5. $\rho_{\geq j} \models \psi_1 \wedge (\vee) \psi_2$ iff $\rho_{\geq j} \models \psi_1$ and(or) $\rho_{\geq j} \models \psi_2$.
   $\rho_{\geq j} \models \neg\psi$ iff $\rho_{\geq j} \not\models \psi$.

6. $\rho_{\geq j} \models \psi_1 \mathbf{U} \psi_2$ iff there exists i $\geq$ j such that $\rho_{\geq i} \models \psi_2$ and $\rho_{\geq k} \models \psi_1$ for all $j \leq k < i$.
   $\rho_{\geq j} \models \mathbf{X}\psi$ iff $\rho_{\geq j+1} \models \psi$.

A formula of CTL is also interpreted using the CTL* semantics. 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 [23] for a concise survey on temporal logics, and further references.

2.2 Formal Verification and Model Checking

Formal Verification, where a system is verified with respect to a desired behaviour, is a technique that establishes properties of hardware or software designs using logic, rather than testing or informal arguments. This involves formal specification of the requirement, formal modeling of the implementation, and precise rules of inference to prove, say, that the implementation satisfies the requirement.

Systems, particularly software, are notoriously buggy. Testing identifies some bugs, but truly exhaustive testing is impossible for almost all systems. Therefore, it is desirable to prove formally that a system satisfies its specified requirements.

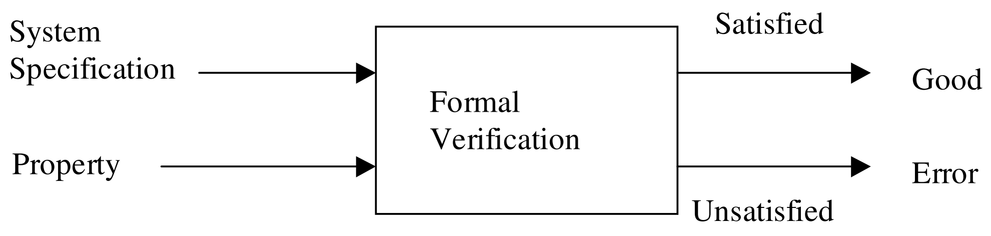

Figure 2.1 Formal verification

Formal proofs on a system require viewing that system's behaviors as sequences of events. One can then model these events as strings over an alphabet, i.e. a formal language.

Verification is frequently accomplished by translating the system into specification using the formal language, converting the requirements (properties) into a second formal language, and testing if the language produced by the system is a subset of the language of the requirements (properties). In practice, subset is computed by complementing the language of the requirements, intersecting with the language of the system, and checking for emptiness. If the resulting language is empty, the system never outputs an illegal string. Equivalently, the system's behavior always conforms to the specification. See Figure 2.1.

Specifically, model checking method, which can be fully automated, is a technique to formally verify finite-state concurrent systems, such as sequential circuit designs and communication protocols. It has a number of advantages over traditional approaches that are based on simulation, testing, and deductive reasoning. In particular, given a formal model and its desired properties, model checking will automatically and usually, conduct quite fast, an exhaustive exploration (either explicitly or symbolically) of all possible behaviors of the model. It gives designers absolute confidence when desired properties are verified. In addition, when a desired property is not satisfied, model checking will generate a counterexample which shows, step by step, how the property is violated by the model. The exact error trace reported by model checking is a big advantage over testing because the error trace provides designers with invaluable information for understanding and removing bugs.

The classical model checking applies to finite state transition systems. A finite-state transition system $\mathcal{T}$ is a tuple (S, s_0, T, P, σ) where S is a finite set of configurations (sometimes called states), $s_0 \in S$ the initial configuration, T a transition relation among the configurations such that each configuration has at least one successor, P a finite set of propositional symbols, and σ a mapping associating to each $s \in S$ a truth assignment $\sigma(s)$ for P. $\mathcal{T}$ may be specified using various formalisms such as a non-deterministic finite-state automaton, or a Kripke structure ([42], see also Definition 2.3). A run ρ of $\mathcal{T}$ is an infinite sequence of configurations $s_0, s_1, \ldots$ such that $(s_i, s_{i+1}) \in T$ for each $i \geq 0$. Intuitively, the information about configurations in S that is relevant to the property to be verified is provided by the corresponding truth assignments to P. The obvious extension of σ to a run ρ is denoted by $\sigma(\rho)$. Thus, $\sigma(\rho)$ is an infinite sequence of truth assignments to P corresponding to the sequence of configurations in ρ.

2.2.1 Propositional LTL Model Checking

Given a transition system $\mathcal{T}$ as above and an LTL formula φ using propositions in P, the associated model checking problem is to check whether every run of $\mathcal{T}$ satisfies φ, or equivalently, that no run of $\mathcal{T}$ satisfies $\neg\varphi$. Pragmatic solutions were enabled by the key result of [50], which shows that each LTL formula ψ over P can be compiled into an automaton A_ψ on infinite sequences, called a Büchi automaton, whose alphabet consists of the truth assignments to P, and which accepts precisely the runs that satisfy ψ. This reduces the model checking problem to checking the existence of a run ρ of $\mathcal{T}$ such that $\sigma(\rho)$ is accepted by $A_{\neg\varphi}$.

To find ρ, one can employ the so-called *nested depth-first search (ndfs) algorithm* [14, 34]. Conceptually, the *ndfs* algorithm performs a systematic construction of runs of $\mathcal{T}$. It begins in the start configuration of $\mathcal{T}$ and at each subsequent step it extends the run constructed so far by following possible transitions in $\mathcal{T}$, 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 [34]. We detail it next.

Büchi Automaton. We present here the flavor of Büchi automaton used in SPIN. A Büchi automaton A is a nondeterministic finite state automaton (NFA) with a special acceptance condition for infinite input sequences: a sequence is accepted iff there exists a computation of A on the sequence that reaches some accepting state s_f of A infinitely often.

For the purpose of model checking, the alphabet of the Büchi automaton A consists of truth assignments for some given set $P{=}P_1, \ldots, P_n$ of propositional variables. The transition relation T of A 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 $(s_i, \delta_i, s_{i+1}) \in T$ 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. Wolper provides a formal description of Büchi automaton in [53].

The results of [50] show that for every LTL formula φ, there exists a Büchi automaton A_φ of size exponential in φ that accepts precisely the infinite sequences of truth assignments that satisfy φ. Furthermore, given a state p of A_φ and a truth assignment σ, the set of possible next states of A_φ under input σ can be computed directly from p and φ in polynomial space [47]. This allows to generate computations of A_φ without explicitly constructing A_φ.

Example 2.4 Figure 2.2 shows a 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.

Suppose we are given a transition system $\mathcal{T}$ whose configurations can be enu-

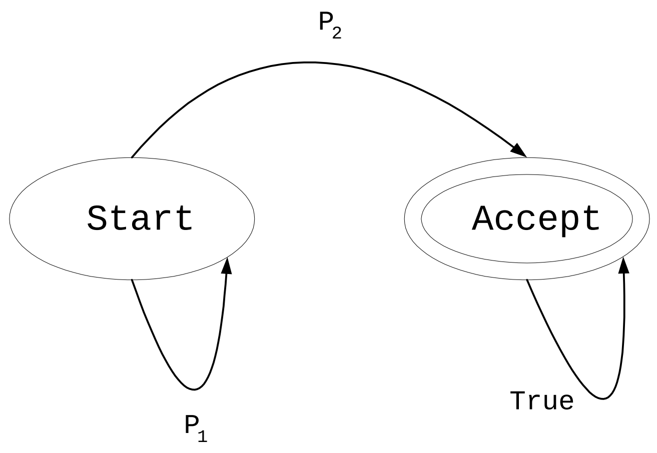

Figure 2.2 Büchi automaton for $\varphi^{aux} = p_1 \mathbf{U} \, p_2$

merated in PSPACE with respect to the specification of $\mathcal{T}$, and such that, given configurations $C_s, C_{s'}$, it can be checked in PSPACE whether $\langle C_s, C_{s'} \rangle$ is a transition in $\mathcal{T}$. Suppose φ is an LTL formula over the set P of propositions of $\mathcal{T}$. The following outlines a non-deterministic PSPACE algorithm for checking whether there exists a run of $\mathcal{T}$ satisfying $\neg\varphi$: starting from the initial configuration C_0 of $\mathcal{T}$ and s_0 of $A_{\neg\varphi}$, non-deterministically extend the current run of $\mathcal{T}$ with a new configuration C_s, and transition to a next state of $A_{\neg\varphi}$ under input $\sigma(C_s)$, until an accepting state s_f of $A_{\neg\varphi}$ is reached. At this point, make a non-deterministic choice: (i) remember s_f and the current configuration C_t of S, or (ii) continue. If a previously remembered final state s_f of $A_{\neg\varphi}$ and configuration C_t of $\mathcal{T}$ coincide with the current state in $A_{\neg\varphi}$ and configuration in $\mathcal{T}$, then stop and answer "yes". This shows that model checking is in non-deterministic PSPACE, and therefore in PSPACE.

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. [14] introduces the *ndfs (nested depth-first search)* below, which searches for runs ρ of $\mathcal{T}$ that determine a lollipop path in $A_{\neg\varphi}$. $T_{\neg\varphi}$ 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 $\mathcal{T}$

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

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

Procedure *stick* performs a depth-first search for a prefix of a run in $\mathcal{T}$ which corresponds to the "stick" prefix of a lollipop path in $A_{\neg\varphi}$. When the search reaches a configuration C_t of $\mathcal{T}$ 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 $\mathcal{T}$ 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 that serve to record that *stick*, and 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 $\mathcal{T}$. These are precisely the runs of length upper bounded by $2N$, with N the product between the number of configurations of $\mathcal{T}$ and states 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.

2.2.2 Propositional CTL(*) Model Checking

As we have seen, effective linear time model checking algorithms were developed by adopting the approach of translating the temporal formulas to nondeterministic automata on infinite words. Similarly, for branching time temporal logic, the automata-theoretic counterpart are automata over infinite trees[51]. However, while modeling checking for the full branching-time temporal logic CTL* is PSPACE-complete, automata-theoretic techniques have long been thought to introduce an exponential penalty by going from CTL* formulas to automata[22], which makes them essentially useless for model checking. Kupferman et al.[37] shows that modeling checking for branching time temporal logic is possible by translating the temporal formulas to alternating tree automata. Then the problem of branching time model checking is reduced to checking the 1-letter non-emptiness of an alternating tree automata(more precisely, the language that the automata recognizes is not empty).

Alternating tree automata generalize the standard notion of nondeterministic tree automata, by allowing several successor states to go down along the same branch of the tree. It is known that, while the translation from branching time temporal logic formulas to nondeterministic tree automata is exponential, the translation to alternating tree automata is linear[43]. The cruial observation is that for model checking, one does not need to solve the nonemptiness problem of tree automata, but rather the 1-letter non-emptiness problem of *word automata*. The problem (testing the nonemptiness of an alternating word automaton that is defined over a singleton alphabet) is substantially simpler.

To obtain an exponential decision procedure for the satisfiability of CTL, Muller et al.[43] use the fact that the nonemptiness problem for a restricted alternating automata—weak alternating automata— is in exponential time. Kupferman et al.[37] prove that the 1-letter nonemptiness of weak alternating automata is decidable in linear running time, which yield an automata-based model checking algorithm of linear time for CTL. Furthermore, they also provide the space complexity bound for branching-time model checking. It comes from the observation that the alternating automata that are obtained from CTL formulas have a special structure: they have *limited alternation*. That kind of alternating automata is called *hesitant alternating automata*. A careful analysis of the 1-letter nonemptiness problem for hesitant alternating word automata yields a top-down model-checking algorithm for CTL that uses linear space in the length of the formula, and only poly-logarithmic in the size of the Kripke structure. Moreover, CTL* formulas can also be translated into hesitant alternating automata and, hence, a space-efficient model checking algorithm is obtained. This implies that for concurrent systems, model checking for CTL and CTL* can be done in space polynomial in the size of system description, rather than requiring space of the order of the exponentially larger expansion of the system.

Please refer to [37] for details of *weak alternating automata*, *hesitant alternating automata* and model checking algorithms.

2.3 Related Works

Our notion of Web application 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.

Prior work on Web service verification has mostly focused on propositional (finite-state) abstractions of both the service workflow and the properties. These abstractions disregard the underlying database and the data values involved in the interaction. They allow one to verify, for instance, that *some* payment occurred before *some* shipment, but not that it involved the intended product and the right amount. [45] proposes an approach to the verification and automated composition of finite-state Web Services specified using the DAML-S standard [15]. The verified properties are propositional, abstracting from the data values. They pertain to safety, liveness and deadlocks, all of which are expressible in LTL. [44] is concerned with verifying a given finite-state web service flow specified in the standard WSFL [56], by using the explicit state model checker SPIN [34]. The properties are expressed in LTL (again abstracting from data content). Another data-agnostic verification effort is carried out in [31, 38], which describe verification techniques focusing on bugs in the control flow engendered by browser operations. The control flow is specified using a browser-action-aware calculus. The flow is verified using model checking, after abstraction to finite-state automata labeled with propositional predicates. The same automata are used for property specification.

Our model is related to WebML [11], a high-level conceptual model for data-driven Web applications, extended in [10] with workflow concepts to support process modeling. It is also related to [5], which proposes a model of peers with underlying databases. The model is a particular case of the one presented here, in which database and state access is restricted to key lookup only, so that at most one tuple is retrieved or updated at any given time. [5] does not address verification, focusing on automatic synthesis of a desired Web Service by "gluing" together an existing set of services.

Other related models are high-level workflow models geared towards Web applications (e.g. [9, 15, 56]), and ultimately to general workflows (see [54, 28, 33, 17, 7, 52]), whose focus is, however, quite different from ours. Non-interactive variants of Web page schemas have been proposed in prior projects such as Strudel [24], Araneus [41] and Weave [25], 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 [40, 39]. Directly relevant to us is Spielmann's work on Abstract State Machine (ASM) transducers [48, 49]. Similarly to the earlier relational transducers [4], these model database-driven reactive systems respond to input events by taking some action, and maintain state information in designated relations. Our model of Web applications is considerably more complex than ASM transducers. The techniques developed in [48, 49] remain, nonetheless, relevant, and we build upon them to obtain our decidability results on the verification of linear-time properties of Web applications. However, unlike the proof of Spielmann that consists of reducing the verification problem to finite satisfiability of E+TC formulas[1], we provide a simpler, more direct proof for our extended model. In particular, this also provides an alternative proof of Spielmann's original result on ASM. For the results on branching-time properties we use a mix of techniques from finite-model theory and temporal logic (see [23]), as well as automata-theoretic model-checking techniques developed by Kupferman, Vardi, and Wolper [37].

We first introduce some declarative specification languages for Web application in Section 2.3.1, then drill into the more related works of Relational Transducer(Section 2.3.2), ASM transducer(Section 2.3.3) and Colombo(Section 2.3.4).

2.3.1 Declarative specification for data-driven Web applications

Many tools(some are surveyed in [26]) have been developed to specify data-driven web applications. Such tools typically access a relational database, use objects (such as J2EE) and dispatchers to object methods for the application logic, provide a scripting language (such as JavaScript) and HTML links for specifying

[1]E+TC is existential first-order logic augmented with a transitive closure operator.

the web site structure, and use style sheets, such as CSS, for specifying web site appearance. The main drawbacks of these approaches are that they do not provide a unified model for all layers of applications, are not declarative, do not use structured programming for web sites, and do not provide systematic methods to deal with application-level conflicts.

Because of these limitations, a variety of research prototype systems have been proposed with the common goal of supporting web application development at a higher level of abstraction.

Araneus

Araneus [41] is a project of Università di Roma Tre which focuses on the definition and prototype implementation of an environment for managing unstructured and structured Web content in an integrated way, called Web Base Management System (WBMS). The WBMS should allow designers to effectively deploy large Web sites, integrate structured and semi-structured data, reorganize legacy Web sites, and Web-enable existing database applications.

On the modeling side, Araneus stresses the distinction between data structure, navigation, and presentation. In structure modeling, a further distinction is made between database and hypertext structure: the former is specified using the Entity-Relationship model, the latter using a notation that integrates structure and navigation specification called Navigation Conceptual Model (NCM). Conceptual modeling is followed by logical design, using the relational model for the structural part, and the Araneus Data Model (ADM) for navigation and page composition.

Of most interest to us is the fact that ADM offers the notion of page scheme, a language-independent page description notation based on such elements as attributes, lists, link anchors, and forms. The use of ADM introduces page composition as an independent modeling task: the specification of data and page structure is orthogonal and therefore different page schemes can be built for the same data. Note that the page scheme here does not involve any kind of interaction with users.

Presentation is specified orthogonal to data definition and page composition, using an HTML template approach.

Strudel

Strudel [24] defines the content of web pages in Strudel query language(StruQL), a declarative language which can access and integrate semi-structured data sources and generate web site graphs. In this way, Strudel separates the description of content from the definition of the structure and navigation of the site. The core idea of Strudel is describing both the schema and the content of a site by means of a set of queries over a data model for semi-structured information. However, Strudel only supports read-only operations. Consequently, it does not provide a uniform framework for handling applications that deal with both queries and updates.

WebML

WebML [11] is a high-level conceptual model for data-driven Web applications, which provides sophisticated tools for specifying the organization of persistent data, navigational structure, and query/update operations. At its core, WebML extends UML with the concept of links, which mirror the structure of a web site. The web site is then declaratively specified in this model using a GUI.

A drawback of the website specification tools like Araneus, Strudel and WebML is that they represent a data-driven web site as a graph, where the nodes in the graph are web pages and the edges are links between the pages. Consequently, the “control flow” of the application can jump from one web page to another so long as there is a connecting edge. This is similar to programming with “goto” statements in the domain of web pages, and has similar disadvantages as compared to structured programming.

Hilda

Hilda [57] is another powerful and declarative Web application development language that shares many common goals with WebML. The primary benefits of Hilda over existing development platforms are:

- It uses a unified data model for all layers of the application, including application logic and presentation.
- It models both application queries and updates.
- It supports structured programming for web sites, that is, there can be conditions on links which can disable some navigation, when the condition is evaluated to be false.
- It supports multi-users which naturally have a potential for conflicts due to concurrently issued application updates. To handle this, it enables conflict detection for concurrent updates.

In summary, the above declarative languages motivate our specification language which will be presented in Chapter 3. There we integrate some features we have seen.

Now we will present some previous formal models of date-driven reactive system: relational transducers and abstract state machine transducers.

2.3.2 Relational Transducers

One formal model that captures the interactive Web site scenario is the relational transducer [4]. In this model, the transducer(application) can access an underlying database, as well as state information updated as the interaction progress. The interaction from the outside world is captured by a sequence of input relations. The application responds by a sequence of output relations. Thus, this model can be viewed as an interactive computational machine similar to an active database. Since the database queried by the transducers has unbounded size, relational transducers are not finite state systems.

A run of a transducer consists of a sequence of inputs and the sequence of outputs generated in response to each of the inputs. In each computation step, such a transducer receives from its environment a set of input tuples and produces a set of output tuples according to the output rules specified in the transducer; the transducer may also update its state relations, according to the state rules. The state relations contain all temporary data necessary to keep track of the ongoing run. The purpose of the database relations is to provide all static data, i.e., data which does not change during a run. In addition, in many cases, only some of the inputs and outputs are semantically significant, while others represent syntactic sugaring that renders the interface more user friendly. For example, payment and delivery of a product might be considered significant, whereas inquiries about prices or reminders of pending bills might not. To capture this distinction, the notion of a "log" is used, which can serve as a kind of "projection" of the execution history of the relational transducer. These components are called the *log* of the transducer. In many circumstances the semantics of relational transducers are considered relative to specified logs.

Example 2.5 [4] Consider an e-commerce site where a customer interacts with the site by two input predicates, order(x, y) and pay(x, y). Catalog information about product price is provided by a relation price(x, y). The system responds to inputs with output predicates sendbill(x, y) and deliver(x). In the process it may consult relation price, and update the state information. In this example, a very simple business model is considered where a customer orders a product, is billed for it, pays, and then receives delivery. More precisely, a company may decide to provide the following business model:

```
TRANSDUCER SHORT
schema
```

```
 database: price;
 input: order, pay;
 state: past-order, past-pay;
 output: sendbill, deliver;
 log: sendbill, pay, deliver
state rules
 past-order(X)+    :-  order(X);
 past-pay(X,Y)+    :-  pay(X,Y);
output rules
 sendbill(X,Y)   :-  order(X), price(X,Y), NOT past-pay(X,Y);
 deliver(X)      :-  past-order(X), price(X,Y), pay(X,Y), NOT past-pay(X,Y).
```

Such a program specifies a relational transducer. It consists of three parts: a schema specification (database, input, output, state, and log relations), a state transition program and an output program. The database is intended to model a large external database that is available for reference by the transducer. The input holds relation schemas for which input data will be received; the state holds internal states for each run being processed; and the output holds relation schemas for which output data will be produced.

The state and output relations are defined, respectively, by a state program and an output program. Processing occurs in a sequence of phases. At each computation step, when a new set of input arrives, the transducer reacts by executing simultaneously the state program and the output program to obtain new values for all of the state and output relations. The + in the state rules indicates that the semantics is cumulative, so the state relations simply contain all previous facts. The output is not cumulative. Intuitively, the "past-" relations prevent an execution step from happening twice. For example, for each execution (as identified by a product name and price) the rule for *pay* will insert a tuple into that relation during exact one step of the transducer execution.

In the example above, the database relation is price(product_name,price). A customer interacts with the system by inserting tuples in two input relations, order and pay. The system responds by producing output relations sendbill and deliver. Imagine that the presence of tuples in these relations is followed by actual actions,

input sequence	order(*Time*) order(*Newsweek*) order(*Hustle*)	order(*Le Monde*) pay(*Time*, 55) pay(*Newsweek*, 48)	pay(*Newsweek*, 45)
output sequence	sendbill(*Time*, 55) sendbill(*Newsweek*, 45)	deliver(*Time*) sendbill(*Le Monde*, 350)	deliver(*Newsweek*)

Figure 2.3 Input and output sequences of a run of SHORT

such as sending an email with the bill and physically delivering the product. The system keeps track of the history of the business transaction using the state relations, here past-order and past-pay. The rules in the example have the obvious semantics, and are fired in parallel. A run of a transducer consists of:

- a sequence of inputs,
- the sequence of outputs generated by the transducer in response to each of the inputs, and
- the restriction of the input-output sequence to the relations in the log.

Note that a run is completely determined by the sequence of inputs. The input and output sequences of a run of SHORT are shown in Figure 2.3. (The prices of *Time, Newsweek, LeMonde* are \$55, \$45 and \$350, respectively.)

Given a relational transducer, a interesting question to ask is what analysis can be performed to validate it. The authors[4] have investigated the following verification problems concerning relational transducer:

- the problem of **verifying temporal properties** of relational transducers(“Does a business model meet its specification?”). For instance, the supplier may wish to verify that a product is never delivered before it has been paid.
- **Goal reachability** asks if a goal can be achieved by some run of the transducer, possibly with some preconditions. In the example, one might wish to

verify that it is possible to achieve the goal deliver(x) as long as $\exists y$ price(x, y) holds in the database.

- the **log validation** problem ("Can a given log sequence actually be generated by some input sequence?"), and
- The problem of **deciding equivalence** of relational transducers("Are two business models equivalent with respect to transactions?").

Although those problems are undecidable in general, positive results, i.e., decidability of variants of those problems, have been obtained for a class of restricted relational transducers called *Spocus transducers*[4]("Spocus" is a acronym for "semi-positive output and cumulative states"), in which the state relations simply accumulate the inputs; and output relations are defined by non-recursive, semi-positive datalog programs with inequality. Many useful properties can be automatically verified in Spocus transducers.

As we mentioned earlier, "log" relations can serve as the "projection" of the execution history of their restricted relational machines; results are obtained in [4] concerning whether one machine is log equivalent to another, in the sense that they produce the same log.

The technique of showing decidability is a reduction to finite satisfiability of a sentence in Bernays-Schönfinkel prefix class.

A drawback of Spocus transducers is that their state relations (i.e., the dynamic relations of their internal databases) are cumulative: the state of a Spocus transducer only accumulates all previous inputs; it cannot be updated otherwise. This limits the field of application of Spocus transducers, substantially. Imagine, for example, a supplier who wants to enable customers to order a product multiple times during a session. The business model of this supplier cannot be expressed by means of a Spocus transducer because once an order for a product has been placed, that order remains in the state of a Spocus transducer until the end of the current session (see also Example 2.5).

2.3.3 ASM transducer

Following up on [4], Spielman extended relational transducers using Gurevich's more general abstract state machines[32], and obtained new positive results on verification of temporal properties, checking equivalence, and validating transaction logs for more powerful transducers–ASM transducers[48, 49]. The proof is based on an ingenious but laborious reduction to the satisfiability problem for the logic E+TC(existential FO augmented with a transitive closure operator).

Like relational transducers [4], the ASM transducers model database-driven reactive systems that respond to input events by taking some actions, and maintain state information in designated relations(called *memory* 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 those of relational transducers. The temporal properties to be verified are also expressed by LTL-FO formulas like those of relational transducers.

ASM transducer is more powerful due to the fact that, unlike Spocus relational transducer [4], rules in the ASM transducer are defined by first-order logic rather than semi-positive datalog rules in Spocus. Furthermore, it treats the state relations as active database with immediate triggering of insertion and deletion, thus allowing state updating. On the other hand, Spocus only allows state to be cumulative. In addition, ASM transducer considers infinite runs over finite domain while Spocus can handle finite runs over infinite domain.

In order to achieve decidability of verification, Spielmann's ASM transducer only allows input from some fixed domain, while Spocus allows arbitrary input, thus, ASM transducer considers infinite runs over fixed domain, while Spocus deals with finite runs oven possible expanding domain. But only restricting the input isn't enough; Spielmann also considers several other possible restrictions. The one of interest to us is *input-bounded ASM with bounded input flow*, denoted as ASM^I, which 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:

- if φ is a FO formula, α is an input atom using a relational symbol from $\mathbf{I}$, $\bar{x} \subseteq free(\alpha)$, 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.

Intuitively, *input-boundedness* means that the first-order quantification in the guards of the rules in specification is bounded to the active domain of the current input. That domain is a finite domain because an ASM relational transducer running in realistic environment never receives "too much" input in a single computation step, due to physical and technical limitations of the environment.

The Main result of Spielmann's ASM^I is:

Theorem 2.6. Given an ASM^I transducer $\mathcal{T}$ and an input-bounded property φ, it is decidable whether $\mathcal{T}$ satisfies φ. Furthermore, It is PSPACE-complete for schemas with fixed bound on the arity, and in EXPSPACE for schemas with no fixed bound on the arity.

First, it is shown that the problem is of PSPACE-hardness by reduction from the satisfiability problem of quantified boolean formulas(QBF). Then, in order to show that is indeed contained in PSPACE, Spielmann's proof 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 containment 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 [48] that finite satisfiability of FO^W+ posTC is polynomially reducible to the finite satisfiability of E+TC(existential FO extended with a transitive closure operator), and that the latter is in PSPACE for fixed database arity, and in EXPSPACE for arbitrary arities.

Before providing more details on the reductions, we briefly review the above logics, using the terminology of [49]. We start with some notations. A finite set of constant symbols and variables is called a *witness set*. For a witness set $\mathcal{W}$ and a variable x not in $\mathcal{W}$, let $(x \in W)$ abbreviate the formula $(\bigvee_{v\in W} x = v)$. Intuitively, $(x \in W)$ holds iff the interpretation of x matches the interpretation of some symbols in W.

Definition 2.7. The *witness-bounded fragment* of FO, denoted by FO^W, is obtained from FO by replacing the formula-formation rule for first-order quantification with the following rules for witness-bounded quantification:

(WBQ) If $\mathcal{W}$ is a witness set, x is a variable not in $\mathcal{W}$, and φ is a formula, then $(\exists x \in W)\varphi$ and $(\forall x \in W)\varphi$ are formulas.

The free and bound variables of FO^W formulas are defined as usual. In particular, x occurs bound in $(\exists x \in W)\varphi$ and $(\forall x \in W)\varphi$, whereas all variables in the witness set $\mathcal{W}$ are free. Thus, FO^W can be viewed as a fragment of FO where formulas of the form $(\exists x \in W)\varphi$ and $(\forall x \in W)\varphi$ are mere abbreviations for $\exists x(x \in W \wedge \varphi)$ and $\forall x(x \in W \rightarrow \varphi)$, respectively.

Let FO+TC denote FO augmented with the transitive closure operator TC, and E+TC denote the existential fragment of FO+TC.

Definition 2.8. [49] The *witness-bounded fragment of transitive-closure logic*, denoted by $(FO^W + TC)$, is obtained from (FO+TC) by replacing the formula-formation rule for the first-order quantification with the rule **(WBQ)**. An occurrence of a TC operator in a $(FO^W + TC)$ formula is called **positive** if the occurrence is in the scope of an even number of negation. By $(FO^W + posTC)$ we denote the set of those $(FO^W + TC)$ formulas in which every occurrence of a TC operator is positive.

Let $\mathbf{T}$ be an ASM^I transducer and $\forall\bar{x}\varphi(\bar{x})$ the universal closure of an input-bounded LTL-FO formula. Clearly, every run of $\mathbf{T}$ satisfies $\forall\bar{x}\varphi(\bar{x})$ iff it is not the case that

$$(\dagger)\text{ there exists a run of }\mathbf{T}\text{ satisfying }\psi \equiv \exists\bar{x}\neg\varphi(\bar{x})$$

Thus, in order to decide whether every run of $\mathbf{T}$ satisfies $\forall\bar{x}\varphi(\bar{x})$, it is enough to solve (†). We outline the main steps of the reduction of (†) to the finite satisfiability of a FO^W+posTC sentence, provided in [48, 49].

Although (†) has the flavor of a satisfiability test, there is an immediate difficulty: transducer runs are infinite, whereas finite satisfiability involves finite structures. This is dealt with by the following:

Periodic Run Lemma: $\mathbf{T}$ *has a run satisfying* ψ *iff it has a* periodic *run satisfying* ψ.

Since a periodic run can be represented by a finite prefix, such runs are representable by finite structures.

The next step provides a definition by an FO+TC formula of the finite structures representing periodic runs of $\mathbf{T}$ that satisfy ψ. Intuitively, the formula describes the connection between consecutive configurations of the transducer by FO formulas and uses the transitive closure operator to describe the entire run and verify satisfaction of ψ. However, a difficulty arises: the formula uses universal quantification. This problem is alleviated by the following:

Local Run Lemma: *Intuitively, an approximate description of the runs of* $\mathbf{T}$ *is sufficient when checking satisfiability of* ψ. *Specifically, the description is exact on the inputs of the runs, but provides only the correct description of the restrictions of memory and action relations to a designated set* C *of constants. The set* C *consists of the database constants as well as constants standing for witnesses to the existentially quantified variables* $\bar{x}$ *in* $\psi = \exists \bar{x} \neg \varphi$. *Such a "run" is called a* local run *of* $\mathbf{T}$. *The lemma shows that there exists a run of* $\mathbf{T}$ *satisfies* ψ *iff there exists a local run of* $\mathbf{T}$ *satisfying* ψ. *Thus, it is sufficient to consider only local runs of* $\mathbf{T}$ *when checking satisfiability of* ψ.

The Local Run Lemma allows replacing the quantifiers of the FO+TC formula by witness-bounded quantifiers, with C serving as a witness set. With some work, this yields an equivalent formula in FO^W+posTC constructed in polynomial time from $\mathbf{T}$ and ψ.

With the periodic lemma and local lemma, we get:

Theorem 2.9. Checking whether an input-bounded ASM transducer satisfies an input-bounded temporal property is PSPACE for fixed arity schemas and EXPSPACE otherwise.

2.3.4 The Colombo Framework

Colombo[5] is a rich formal framework for web service composition, which addresses message exchanges, data flow management, and effects on the real world. Using Colombo, the authors study the problem of automatic service composition (synthesis) and devise a sound, complete and decidable algorithm for building a composite service. Specifically, it develops (i) a technique for handling the data, which ranges over an infinite domain, in a finite, symbolic way, and (ii) a technique to automatically synthesize composite web services, based on Propositional Dynamic Logic. Although the problem Colombo considers is different from the problem presented in this thesis, but their work is nevertheless related to this dissertation, since it is the first unified model that take data into account.

More precisely, Colombo is a framework for web services that combines

- A world state, representing the real world, viewed as a database instance over a relational database schema
- Atomic processes in the spirit of OWL-S,
- Message passing, including a simple notion of ports and links, as found in web services standards (e.g., WSDL, BPEL)
- An automata-based model(Guarded Automata) of the internal behavior of the web services, where the individual transaction correspond to atomic processes, message writes, and message reads.
- a local store for each web service, used manage the data read/written with messages and input/output by atomic processes; and
- a simple form of integrity constraints on the world state

Data may be internal to services or shared by the service community; therefore, the authors introduce, respectively, the notions of local store and world state to represent them. In general the data contained in the local store is not a subset of the data in the world schema, since the former may refer to variables that are internal to the services, and therefore, are not exported.

In addition to the world state, a service community is also characterized by a set of atomic processes that modify the world state, a finite set of services, and a set of message types, denoting the alphabet of the community. Each atomic process has some internal structures (input, output, or interaction with the world state).

Each service is generally non-atomic, and has process flow characterized in terms of its behavior expressed as a guarded automaton, defined over the alphabet of the community as usual. Each service interacts with the other services in the community and with the client, in terms of messages and actions. Therefore, the primitive actions it defines are not only essentially used to send a message (type), receive a message (type), but also to perform an operation, by specifying input parameters, and to assign some value to a location in the local store. The transition relation defines the set of successor states given a state, a guard and a primitive action. A guard is a first order logic formula in the general case, it is a propositional formula in a restricted, decidable framework. Finally, between the various services in the community, a set of (bi-directional) channels can be defined: in general, the authors envision a queue based communication topology, where the various queues are not part of the services, but are seen as part of the community.

The problem of service composition, in such a framework, aims at building a new service that realizes a client request, by coordinating the available services. Several solutions to this problem may be found, depending also on the formalism used to express the client request.

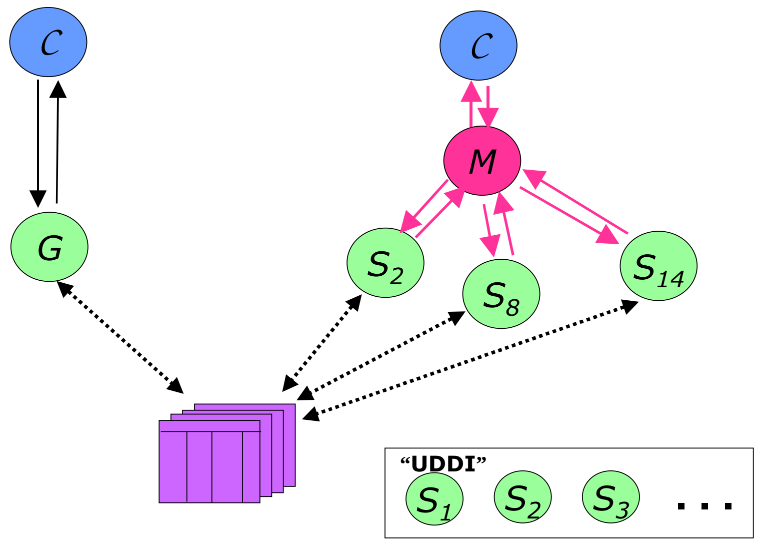

Figure 2.4 Illustration of mediator synthesis

Using the Colombo model, [5] develops a framework for posing composition problems, that closely parallels the way composition might be done using standard-based web services. One composition problem is called the *mediator synthesis* problem. This focuses on how to build a mediator service that simulates the behavior of a target web service, where the mediator can only use message passing to get the pre-existing web services to invoke atomic processes (which in turn impact the "real world"). The mediator synthesis problem is illustrated in Figure 2.4. As suggested in the left side of that figure we assume that a (virtual) "goal service" $\mathcal{G}$ is given as a Colombo guarded automaton. $\mathcal{G}$ includes atomic processes that can manipulate "real world" relations, and also messaging ports that can be used by a hypothetical client $\mathcal{C}$. Also provided as inputs is a set of Colombo web services $\mathcal{S}_1, \ldots, \mathcal{S}_n$. (These can be viewed as existing services in a generalized form of "UDDI" directory). The challenges, as suggested in the right side of Figure 2.4 are to:

- select a subset of services from the UDDI directory,
- synthesize a new service M (a "mediator"), and
- construct "linkages" between M, the chosen services, and the client $\mathcal{C}$,

so that the set of possible behaviors of the system involving M and the selected services, at least as can be observed by $\mathcal{C}$, by the invocations of atomic processes, and by the impact on the "real world", is identical to the set of possible behaviors of $\mathcal{G}$.

A second problem studied in [5], called *choreography synthesis*, is illustrated in Figure 2.5. The inputs for this problem are identical to those of the mediator synthesis problem. However, in choreography synthesis the challenge is to select elements of the "UDDI" directory and then to construct a linkage for message passing that goes between those services and the client, so that the overall system simulates the goal service(i.e., there is no "new" service involved).

Under certain restrictions, [5] demonstrates the decidability of the existence of a solution to the mediator synthesis problem, and similarly for the choreography synthesis problem. Further, a method for building a solution for a given synthesis problem is provided, if a solution exists. These results are based on

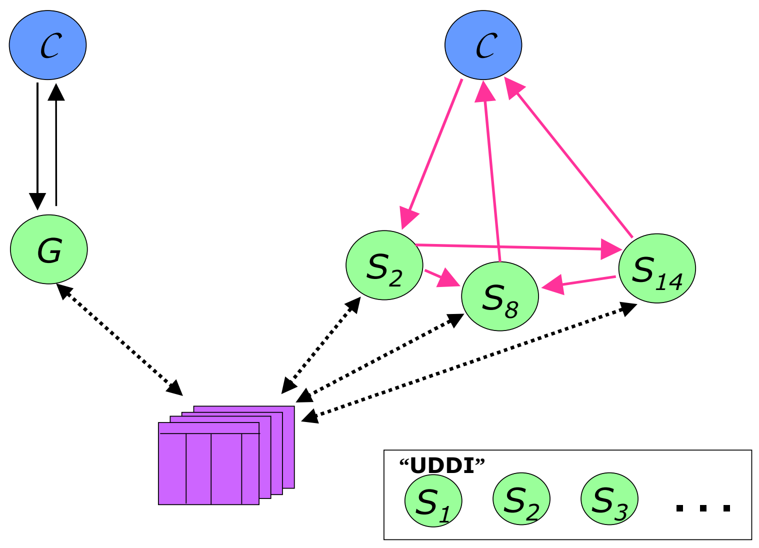

Figure 2.5 Illustration of choreography synthesis

- Database accesses are restricted to key lookup only, so that at most one tuple is retrieved or updated at any given time.
- a technique for reducing potentially infinite domains of data values (in the "real world") into a finite set of symbolic data values, and
- in a generalization of [6], a mapping of the composition problem into PDL.

The results reported in [5] rely on a number of restrictions; a broad open problem concerns how these restrictions can be relaxed while still retaining decidability. Also, the worst-case complexity as reported in [5] is doubly exponential time; it is hoped to reduce it at least to exponential time.

In addition, [5] does not address verification, focusing on automatic synthesis of a desired Web service, by "gluing together" an existing set of services.

Chapter 3

Specification of Interactive Data-Driven Web Applications

In this chapter we provide our model and specification language for data-driven Web applications. It is particularly important that the specification of the Web application is concise and easy to understand, which motivates us to focus on simple rule-based specifications. Moreover, the specification language should be expressive enough such that we can model interesting Web applications.

3.1 Syntax of Specification Language

Our model of Web application captures the interaction of an external user with the Web site, referred to as a "run". Informally, a Web application specification has the following components:

- A database that remains fixed throughout every 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, states, and all previous inputs. In addition, it specifies the state transitions 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 application. 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 application.

We adopt here an active domain semantics for FO formulas, as commonly done in database theory (e.g., see [3]).

Definition 3.1. A *Web application* $\mathcal{W}$ 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 $\{prev_I \mid I \in \mathbf{I} - \mathbf{const}(\mathbf{I})\}$, 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* $\mathit{Options}_I(\bar{x}) \leftarrow \varphi_{I,W}(\bar{x})$ where $\mathit{Options}_I$ 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 $\mathbf{D} \cup \mathbf{S} \cup \mathbf{Prev_I} \cup \mathbf{const}(\mathbf{I})$, 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 $\varphi^{\theta}_{S,W}(\bar{x})$ are FO formulas over schema $\mathbf{D} \cup \mathbf{S} \cup \mathbf{Prev_I} \cup \mathbf{const}(\mathbf{I}) \cup \mathbf{I}_W$, with free variables $\bar{x}$ and $\theta \in \{+, -\}$.

- 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 $\mathbf{D} \cup \mathbf{S} \cup \mathbf{Prev_I} \cup \mathbf{const}(\mathbf{I}) \cup \mathbf{I}_W$, 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 $\mathbf{D} \cup \mathbf{S} \cup \mathbf{Prev_I} \cup \mathbf{const}(\mathbf{I}) \cup \mathbf{I}_W$.

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, i.e. if in the same page, a state tuple will both be inserted and deleted from the state relation, then it will be treated as a no-op, neither insertion nor deletion will be performed. Refer to Definition 3.3 for more details). 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 $prev_J$ contains the input to J in the previous step of the run (if any). The choice of this semantics for $prev_J$ relations is somewhat arbitrary, and other choices are possible without affecting the results. For example, another possibility is to have $prev_J$ hold the *most recent* input to J occurring anywhere in the run, rather than in the previous step. Also note that $prev_J$ relations are really state relations with very specific functionality, and are redundant in the general model. However, they are very useful when defining tractable restrictions 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 3.2 below, error $\in \mathbf{S}$, $\underline{\mathsf{user}} \in \mathbf{D}$ and $\mathsf{name}, \mathsf{password}, \mathsf{button} \in \mathbf{I}$.

3.2 Example

Example 3.2 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[1] implementing this example, together with its full specification, is provided at http://www.cs.ucsd.edu/~lsui/project/index.html.

The demo site implements the Web application $\langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, \mathbf{HP}, W_\epsilon \rangle$. Figure 3.1 represents an overview of all Web pages of our demo, depicted in WebML style. We list here only the pages in $\mathbf{W}$ that are mentioned 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:`

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

`State Rules:`

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

`Target Web Pages` $\mathbf{T}_{HP}$: HP, RP, CP, AP, MP

`Target Rules:`

$\text{HP} \leftarrow \mathsf{button}(\text{“}clear\text{”})$

$\text{RP} \leftarrow \mathsf{button}(\text{“}register\text{”})$

$\text{CP} \leftarrow \quad \underline{\mathsf{user}}(\mathsf{name}, \mathsf{password}) \wedge \mathsf{button}(\text{“}login\text{”}) \wedge \mathsf{name} \neq \text{“}Admin\text{”}$

$\text{AP} \leftarrow \quad \underline{\mathsf{user}}(\mathsf{name}, \mathsf{password}) \wedge \mathsf{button}(\text{“}login\text{”}) \wedge \mathsf{name} = \text{“}Admin\text{”}$

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

`End Page` HP

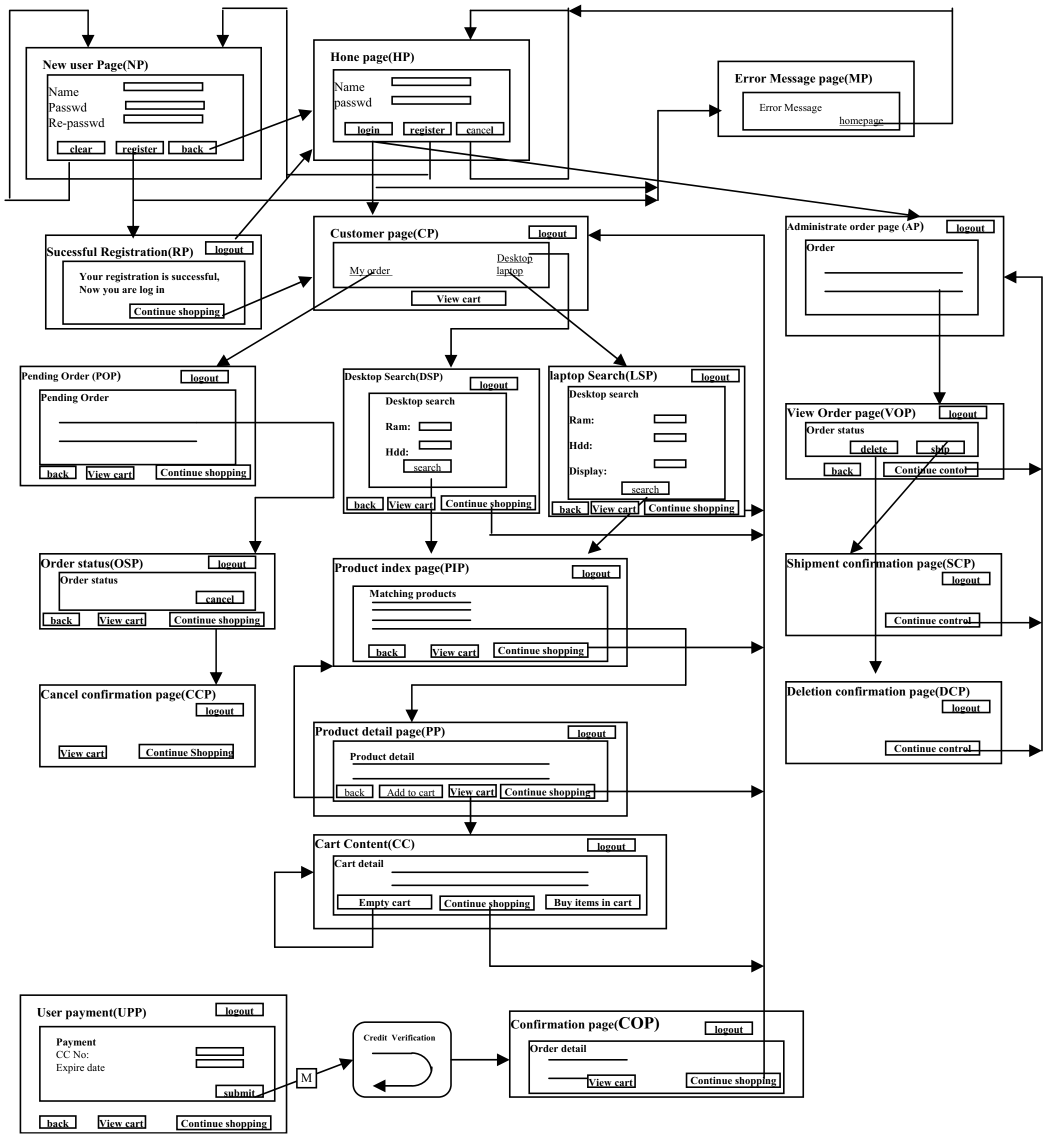

Figure 3.1 Web pages in the demo

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 3.3), 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 state rule looks up the name/password combination in the database table $\underline{\mathsf{user}}$. If the lookup fails, the 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 [1] for the full version, which also allows users to search for desktops).

`Page` LSP

`Inputs` $\mathbf{I}_{LSP}$: $\mathsf{laptopsearch}(ram, hdisk, display), \mathsf{button}(x)$

`Input Rules:`

$\text{Options}_{\mathsf{button}}(x) \leftarrow \quad x = \text{“}search\text{”} \vee x = \text{“}view\ cart\text{”} \vee x = \text{“}logout\text{”}$
$\text{Options}_{\mathsf{laptopsearch}}(r, h, d) \leftarrow \underline{\mathsf{criteria}}(\text{“}laptop\text{”}, \text{“}ram\text{”}, r)$
$\qquad \wedge \underline{\mathsf{criteria}}(\text{“}laptop\text{”}, \text{“}hdd\text{”}, h) \wedge \underline{\mathsf{criteria}}(\text{“}laptop\text{”}, \text{“}display\text{”}, d)$

`State Rules:`

$\text{userchoice}(\mathsf{r},\mathsf{h},\mathsf{d}) \leftarrow \mathsf{laptopsearch}(r, h, d) \wedge \mathsf{button}(\text{“}search\text{”})$

`Target Web Pages` $\mathbf{T}_{LSP}$: HP, PIP, CC

`Target 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` LSP

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. 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 transits to the PIP page. □

3.3 Definition of Run

We next define the notion of "run" of a Web application. 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 $V_i \in \mathbf{W}\cup\{W_\epsilon\}$, 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 $\mathbf{A}_{V_i}$. We call $\langle V_i, S_i, I_i, P_i, A_i\rangle$ a *configuration* of the run.

The input constants play a special role in runs. 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 application 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 3.3. Let $\mathcal{W} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, W_0, W_\epsilon\rangle$ be a Web application and D a database instance over schema $\mathbf{D}$. A *run* of $\mathcal{W}$ for database D is an infinite sequence of configurations $\{\langle V_i, S_i, I_i, P_i, A_i\rangle\}_{i\geq 0}$ where $V_i \in \mathbf{W} \cup \{W_\epsilon\}$, 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 involve 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 $v \in \mathit{Options}_R$, where $\mathit{Options}_R$ 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 $\mathbf{prev_I}$, $P_i(\mathit{prev}_I) = I_{i-1}(I)$ if $I \in \mathbf{I}_{V_{i-1}}$ and $P_i(\mathit{prev}_I)$ is empty otherwise.
- for each $i \geq 0$, and relation S in $\mathbf{S}$, $S_{i+1}(S)$ is the result of evaluating

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

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:

- the value of an input constant is required by some formula before it was provided by the user;

- the user is asked to provide a value for the same input constant more than once; and,

- 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 application error-free if it generates only error-free runs. Clearly, it would be desirable to verify that a given Web application 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 applications.

3.4 Temporal properties of the Web applications

With the definition of "run", we next begin to define the requirements that must be satisfied by the Web application. Such requirements are expressed using a variant of temporal logic.

Temporal logic comes in two varieties: linear-time temporal logic assumes implicit universal quantification over all paths(runs) of the Web applications; branching-time temporal logic allows explicit existential and universal quantification over all paths(runs), expresses temporal properties involving several different branches of the runs.

3.4.1 Linear-time First-order Temporal Logic

We begin with linear-time properties, that must be satisfied by *all* runs of the Web applications. Let $\mathcal{W} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, W_0, W_\epsilon \rangle$ be a Web application. LTL is not expressive enough to capture some interesting properties of runs of $\mathcal{W}$; therefore, we extend LTL with first-order components, adapted from [23, 2, 49].

Starting from the propositional LTL formula such as p **B** q, we replace each proposition by a first-order formula over the schema of the Web application, which is exactly the first-order component of the linear-time first-order formula(denoted by LTL-FO).

Definition 3.4. [23, 2, 49] The language LTL-FO (linear-time first-order 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. For a given LTL-FO sentence, we refer to its maximal subformulas containing no temporal operators, as the *FO components* of the sentence.

Let $\mathcal{W} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, W_0, W_\epsilon \rangle$ be a Web application. 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 application $\mathcal{W}$ satisfies $\forall \bar{x} \varphi(\bar{x})$ iff every run of $\mathcal{W}$ satisfies it. Let $\rho = \{\rho_i\}_{i \geq 0}$ be a run of $\mathcal{W}$ for database D, and let $\rho_{\geq j}$ denote $\{\rho_i\}_{i \geq j}$, for $j \geq 0$. Note that $\rho = \rho_{\geq 0}$. Let $\text{Dom}(\rho)$ be the active domain of ρ, i.e. the set of all elements occurring in relations or as interpretations for constants in ρ. The run ρ satisfies $\forall \bar{x} \varphi(\bar{x})$ iff for each valuation ν of $\bar{x}$ in $\text{Dom}(\rho)$, $\rho_{\geq 0}$ satisfies $\varphi(\nu(\bar{x}))$. The latter is defined by structural induction on the formula. An FO sentence ψ is satisfied by $\rho_i = \langle V_i, S_i, I_i, P_i, A_i \rangle$ if the following hold:

- the set of input constants occurring in ψ is included in κ_i;
- the structure ρ'_i satisfies ψ, where ρ'_i is the structure obtained by augmenting ρ_i with interpretation σ_i for the input constants. Furthermore, ρ_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 $j \geq 0$):

- $\rho_{\geq j} \models \mathbf{X}\varphi$ iff $\rho_{\geq j+1} \models \varphi$,
- $\rho_{\geq j} \models \varphi \mathbf{U} \psi$ iff $\exists k \geq j$ such that $\rho_{\geq k} \models \psi$ and $\rho_{\geq l} \models \varphi$ 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 $\neg(\neg\varphi \mathbf{U} \neg\psi)$, $\mathbf{G}\varphi \equiv \textit{false}\ \mathbf{B}\ \varphi$, and $\mathbf{F}\varphi \equiv \textit{true}\ \mathbf{U}\ \varphi$. We use the above operators as shorthand in LTL-FO formulas whenever convenient.

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

Example 3.5 The following property states that if page P is reached in a run, then page Q will be eventually reached as well:

$$\mathbf{G}(\neg\ \mathrm{P}) \vee \mathbf{F}(\ \mathrm{P} \wedge \mathbf{F}\ \mathrm{Q}) \tag{3.1}$$

□

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

Example 3.6 Assume that the Web application in Example 3.2 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 UPP, 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 3.2). 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:

$$\begin{aligned} &\forall pid, price\ [\xi(pid, price)\ \mathbf{B} \\ &\neg(\overline{\mathsf{conf}}(\mathsf{name}, price) \wedge \overline{\mathsf{ship}}(\mathsf{name}, pid))\ ] \end{aligned} \tag{3.2}$$

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

$$\begin{aligned} \mathrm{UPP} \wedge \mathsf{pay}(price) \wedge \mathsf{button}(\text{“}\mathit{authorize\ payment}\text{”}) \\ \wedge \mathrm{pick}(pid, price) \\ \wedge \exists pname\ \underline{\mathsf{catalog}}(pid, price, pname) \end{aligned} \tag{3.3}$$

□

3.4.2 Branching-time First-order Temporal Logic

Branching-time logics allow expressing temporal properties involving quantification over runs. For example, such quantification is needed to express the property "At any point in a run, there is a way to return to the shopping cart page".

We next provide the syntax and semantics of the branching-time logics CTL-FO and CTL*-FO, adapted from [2, 49]. These are extensions of the well-known languages CTL and CTL* (see [23]). We also review the notion of satisfaction of a CTL$^{(*)}$ formula by a Kripke structure.

Definition 3.7. Let $\mathcal{W} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, W_0, W_\epsilon \rangle$ be a Web application. The set of CTL*-FO formulas over $\mathcal{W}$ is the set of state formulas defined inductively together with the set of path formulas as follows:

1. each FO formula over the vocabulary of $\mathcal{W}$ is a state formula;
2. if φ and ψ are state formulas then so are $\varphi \wedge \psi$, $\varphi \vee \psi$, and $\neg\varphi$;
3. if φ is a path formula, then $\mathrm{E}\varphi$ and $\mathrm{A}\varphi$ are state formulas;
4. each state formula is also a path formula;
5. if φ and ψ are path formulas, then so are $\varphi \wedge \psi$, $\varphi \vee \psi$, and $\neg\varphi$;
6. if φ and ψ are path formulas, then so are $\mathbf{X}\varphi$ and $\varphi\mathbf{U}\psi$.

The set of CTL-FO formulas over $\mathcal{W}$ is defined by replacing (4)-(6) above by the rule:

- if φ and ψ are state formulas then $\mathbf{X}\varphi$, and $\varphi\mathbf{U}\psi$ are path formulas.

The set of CTL$^{(*)}$-FO sentences consists of the universal closures of CTL$^{(*)}$-FO formulas.

Note that, as in the case of LTL-FO, first-order quantifiers cannot be applied to formulas using temporal operators or path quantifiers. The formula is closed at the very end by universally quantifying all remaining free variables, yielding an CTL$^{(*)}$-FO sentence.

The semantics of the temporal operators is the natural extension of LTL-FO with path quantifiers. 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". More formally, satisfaction of a CTL$^{(*)}$-FO sentence by a Web application $\mathcal{W}$ 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 [23]), and is provided below. The main difference is that propositional variables are not explicitly provided; instead, the FO components of the formulas have to be evaluated on the current configuration (last of the prefix defining the node), as described earlier. We say that a Web application $\mathcal{W}$ satisfies φ, denoted $\mathcal{W} \models \varphi$, iff $\mathcal{T}_{\mathcal{W},D} \models \varphi$ for every database D.

Example 3.8 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 3.6 (3.3). □

Chapter 4

Theoretical Results on Verification

The verification problem concerns the correctness of Web applications and thus emerges while designing Web applications. We use temporal logic to specify requirements on Web applications.

As we saw in Chapter 3, temporal logic comes in two varieties: linear-time temporal logic assumes implicit universal quantification over all paths(runs) of the Web applications; branching-time temporal logic allows explicit existential and universal quantification over all paths(runs), expresses temporal properties involving several different branches of the runs.

Section 4.1 introduces ASM$^+$ transducers and presents the verification results in this context. The main interest of ASM$^+$ transducer is that, as we shall see, it is sufficiently powerful to simulate a wide class of Web applications. The verification problem is shown decidable in section 4.1.1. Section 4.1.2 goes further and prove that the restriction is so tight that even small relaxation will lead to undecidability. Branching-time properties are verified in section 4.1.3. Finally, Section 4.2 establishes the verification results for Web application, mostly by reduction to verification of ASM$^+$ trabsducers.

4.1 ASM$^+$ Transducers

In this section we present an extension of Spielmann's Abstract State Machine (ASM) transducers [48, 49] and prove our verification results within this framework. We denote the extended transducer model by ASM$^+$. Informally, ASM$^+$ transducers extend Spielmann's ASM transducer model with the ability to inspect the previous user input, and with input option rules constraining the choice of input by the user. The main interest of the extension is that, as we shall see, it is sufficiently powerful to simulate a wide class of Web applications. We next define ASM$^+$ transducers formally.

Definition 4.1. An ASM$^+$ transducer $\mathcal{A}$ is a tuple $\langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathcal{R} \rangle$, where:

- **D, S, I, A** are relational schemas called database, state, input, and action schemas, respectively, where **S, I, A** contain no constant symbols. We denote by $\mathbf{Prev_I}$ the relational vocabulary $\{prev_I \mid I \in \mathbf{I}\}$, where $prev_I$ has the same arity as I (intuitively, $prev_I$ refers to the input I at the previous step in the run).

- $\mathcal{R}$ is a set of rules containing the following:

  – For each input relation $I \in \mathbf{I}$ of arity $k > 0$, an *input rule* $Options_I(\bar{x}) \leftarrow \varphi_I(\bar{x})$ where $Options_I$ is a relation of arity k, $\bar{x}$ is a k-tuple of distinct variables, and $\varphi_I(\bar{x})$ is an FO formula over schema $\mathbf{D} \cup \mathbf{S} \cup \mathbf{Prev_I}$, 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 $S(\bar{x}) \leftarrow \varphi_S^+(\bar{x})$,

    ∗ a deletion rule $\neg S(\bar{x}) \leftarrow \varphi_S^-(\bar{x})$,

    where the arity of S is k, $\bar{x}$ is a k-tuple of distinct variables, and $\varphi_S^\epsilon(\bar{x})$ are FO formulas over schema $\mathbf{D} \cup \mathbf{S} \cup \mathbf{Prev_I} \cup \mathbf{I}$, with free variables $\bar{x}$.

– For each action relation $A \in \mathbf{A}$, 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 $\mathbf{D} \cup \mathbf{S} \cup \mathbf{Prev_I} \cup \mathbf{I}$, with free variables $\bar{x}$.

Note that an ASM$^+$ transducer is isomorphic to a Web application with a single Web page schema, and no input constants. The definition of runs, as well as the syntax and semantics of LTL-FO and CTL(*)-FO formulas, are therefore inherited from Web applications. Also, any lower bounds for verification problems proven for ASM$^+$ transducers apply trivially to Web applications. To transfer the upper bounds, we will need to show appropriate reductions from Web applications to the simpler ASM$^+$ transducers.

For completeness, we briefly describe configurations and runs of ASM$^+$ transducers. Let $\mathcal{A} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathcal{R} \rangle$ be an ASM$^+$ transducer. A configuration of $\mathcal{A}$ is a tuple $\langle S, I, P, A \rangle$, where S is an instance of $\mathbf{S}$, I and P are instances of $\mathbf{I}$ and $\mathbf{Prev_I}$ consisting of at most one tuple, and A is an instance of $\mathbf{A}$. Let D be an instance of $\mathbf{D}$. A *run* of $\mathcal{A}$ on database D is an infinite sequence of configurations $\{\langle S_i, I_i, P_i, A_i \rangle\}_{i \geq 0}$ where:

- all relations in S_0, P_0, and A_0 are empty and all propositions are false;
- I_0 consists of at most one tuple for each input relation, belonging to the result of evaluating the corresponding input option rule; and,
- for $i \geq 0$, $\langle S_{i+1}, I_{i+1}, P_{i+1}, A_{i+1} \rangle$ is obtained from D and $\langle S_i, I_i, P_i, A_i \rangle$ using the rules $\mathcal{R}$, as done for Web applications with a single Web page (details omitted, see Definition 3.3).

Note that each configuration in a run over database D uses values from the domain of D. In particular, every run, although infinite, has only finitely many distinct configurations.

We next present our results on verification of ASM$^+$ transducers, first for linear-time properties, then for branching-time properties.

4.1.1 Verification of LTL-FO properties

It is easily seen that it is undecidable if an ASM$^+$ transducer satisfies an LTL-FO formula, as shown next.

Proposition 4.2. It is undecidable, given an ASM$^+$ transducer $\mathcal{A}$ and an LTL formula ψ, whether $\mathcal{A} \models \psi$.

Proof. By Trakhtenbrot's theorem, finite satisfiability of FO sentences is undecidable. Let $\mathbf{D}$ be a database schema and φ an FO sentence over $\mathbf{D}$. Consider an ASM$^+$ transducer with database schema $\mathbf{D}$ and an action rule $A \leftarrow \varphi$, where A is a proposition. Clearly, φ is finitely satisfiable iff $\mathcal{A} \not\models \mathbf{G}\neg A$. $\square$

To obtain decidability, we must restrict both the transducers and the LTL-FO sentences. We use a restriction proposed in [48, 49] for ASM transducers, limiting the use of quantification in state and action rule formulas to "input-bounded" quantification, and an additional restriction on input rules. The restrictions are formulated in our framework as follows. Let $\mathcal{A} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathcal{R} \rangle$ be an ASM$^+$ transducer. The set of *input-bounded* FO formulas over the schema $\mathbf{D} \cup \mathbf{S} \cup \mathbf{I} \cup \mathbf{A} \cup \mathbf{Prev_I}$ 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.

An ASM$^+$ transducer is input-bounded iff all formulas in state and action rules are input bounded, and all input rules use $\exists^*$FO formulas in which all state atoms are ground (note that the input option rules do not have to obey the restricted quantification formation rule above). An LTL-FO sentence over the schema of $\mathcal{A}$ is input-bounded iff all of its FO components are input-bounded.

It was shown in [48, 49] that checking satisfaction of an input-bounded LTL-FO sentence by an input-bounded ASM transducer is PSPACE-complete. The lower bound, shown by reduction of Quantified Boolean Formula [27], transfers immediately to input-bounded ASM$^+$ transducers and LTL-FO formulas, since ASM transducers are special cases of ASM$^+$ transducers. The PSPACE upper bound is shown in [48, 49] by reducing the verificatiom problem to finite satisfiability in the logic E+TC, existential FO extended with a transitive closure operator. With some care, this proof can be adapted to ASM$^+$ transducers. However, the proof we provide is considerably more direct, circumventing the laborious reduction to E+TC and the proof of decidability of finite satisfiability for this logic. It also provides an alternative proof to Spielmann's original result on input-bounded ASM transducers. Furthermore, the construction used in our proof provides the basis for a practical implementation of a verifier for Web applications, described in [20, 18].

We next present our proof that verification of input-bounded ASM$^+$ transducers can be done in PSPACE assuming a fixed bound on the arity of relations, and in EXPSPACE otherwise. Consider an ASM$^+$ transducer $\mathcal{A} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathcal{R} \rangle$. For simplicity of exposition, we assume that $\mathbf{I}$ consists of only one input relation (the development easily extends to multiple input relations). In particular, a configuration of $\mathcal{A}$ is a tuple $\langle S, I, P, A \rangle$ where S is an instance of $\mathbf{S}$, A is an instance of $\mathbf{A}$, and I and P are instances of the unique input relation in $\mathbf{I}$, each consisting of at most one tuple.

Consider an input-bounded ASM$^+$ transducer $\mathcal{A}$ and an input-bounded LTL-FO formula $\varphi_0 = \forall \bar{x} \psi_0(\bar{x})$ over the schema of $\mathcal{A}$. To check that every run of $\mathcal{A}$ satisfies φ_0 we equivalently verify that there is no run of $\mathcal{A}$ satisfying $\neg\varphi_0 = \exists \bar{x} \neg\psi_0(\bar{x})$. To do so, we would like to adapt classical model checking based on Büchi automata. We informally recall this approach (see e.g. [13] for a formal development).

Propositional model checking Classical model checking applies to finite state transition systems. A finite-state transition system $\mathcal{T}$ is a tuple (S, s_0, T, P, σ) where S is a finite set of configurations (sometimes called states), $s_0 \in S$ the initial configuration, T a transition relation among the configurations such that each configuration has at least one successor, P a finite set of propositional symbols, and σ a mapping associating to each $s \in S$ a truth assignment $\sigma(s)$ for P. $\mathcal{T}$ may be specified using various formalisms such as a non-deterministic finite-state automaton, or a Kripke structure ([13], see also Section 2.1.3). A run ρ of $\mathcal{T}$ is an infinite sequence of configurations $s_0, s_1, \dots$ such that $(s_i, s_{i+1}) \in T$ for each $i \geq 0$. Intuitively, the information about configurations in S that is relevant to the property to be verified is provided by the corresponding truth assignments to P. The obvious extension of σ to a run ρ is denoted by $\sigma(\rho)$. Thus, $\sigma(\rho)$ is an infinite sequence of truth assignments to P corresponding to the sequence of configurations in ρ.

Given a transition system $\mathcal{T}$ as above and an LTL formula φ using propositions in P, the associated model checking problem is to check whether every run of $\mathcal{T}$ satisfies φ, or equivalently, that no run of $\mathcal{T}$ satisfies $\neg\varphi$. This can be done using a key result of [50], showing that from each LTL formula ϕ over P one can construct an automaton A_ϕ on infinite sequences, called a Büchi automaton, whose alphabet consists of the truth assignments to P, and which accepts precisely the runs of $\mathcal{T}$ that satisfy ϕ. This reduces the model checking problem to checking the existence of a run ρ of $\mathcal{T}$ such that $\sigma(\rho)$ is accepted by $A_{\neg\varphi}$.

We briefly recall Büchi automata. A Büchi automaton A is a nondeterministic finite state automaton (NFA) with a special acceptance condition for infinite input sequences: a sequence is accepted iff there exists a computation of A on the sequence that reaches some accepting state f infinitely often. For the purpose of model checking, the alphabet consists of truth assignments for some given set P of propositional variables. The results of [50] show that for every LTL formula φ there exists Büchi automaton A_φ of size exponential in φ that accepts precisely the infinite sequences of truth assignments that satisfy φ. Furthermore, given a state p of A_φ and a truth assignment σ, the set of possible next states of A_φ under input σ can be computed directly from p and φ in polynomial space [47]. This allows to generate computations of A_φ without explicitly constructing A_φ.

Suppose we are given a transition system $\mathcal{T}$ whose configurations can be enumerated in PSPACE with respect to the specification of $\mathcal{T}$, and such that, given configurations s, s', it can be checked in PSPACE whether $\langle s, s' \rangle$ is a transition in $\mathcal{T}$. Suppose φ is an LTL formula over the set P of propositions of $\mathcal{T}$. The following outlines a non-deterministic PSPACE algorithm for checking whether there exists a run of $\mathcal{T}$ satisfying $\neg\varphi$: starting from the initial configuration s_0 of $\mathcal{T}$ and q_0 of $A_{\neg\varphi}$, non-deterministically extend the current run of $\mathcal{T}$ with a new configuration s, and transition to a next state of $A_{\neg\varphi}$ under input $\sigma(s)$, until an accepting state f of $A_{\neg\varphi}$ is reached. At this point, make a non-deterministic choice: (i) remember f and the current configuration s of S, or (ii) continue. If a previously remembered final state f of $A_{\neg\varphi}$ and configuration s of $\mathcal{T}$ coincide with the current state in $A_{\neg\varphi}$ and configuration in $\mathcal{T}$, then stop and answer "yes". This shows that model checking is in non-deterministic PSPACE, and therefore in PSPACE.

From classical model checking to ASM$^+$ verification There are two main obstacles to using classical model checking to verify ASM$^+$ transducers. First, LTL-FO formulas are not propositional. Second, the transition systems corresponding to ASM$^+$ transducers are not finite state, since they have infinitely many possible configurations. We next show how to overcome both obstacles.

Consider an input-bounded ASM$^+$ transducer $\mathcal{A}$ and an input-bounded LTL-FO formula $\varphi_0 = \forall\bar{x}\psi_0(\bar{x})$. Let $\psi = \neg\psi_0$ and $\varphi = \neg\varphi_0 = \exists\bar{x}\psi(\bar{x})$. Let $\bar{c}$ be a tuple of distinct constant symbols of the same arity as $\bar{x}$. Verifying that all runs of a transducer $\mathcal{A}$ satisfy φ_0 is equivalent to checking that no run satisfies $\psi(\bar{x} \leftarrow \bar{c})$ (the formula obtained by substituting $\bar{c}$ for $\bar{x}$ in $\psi(\bar{x})$) for any interpretation of the constants $\bar{c}$. Let us denote $\psi(\bar{x} \leftarrow \bar{c})$ by $\psi_{\bar{c}}$. Consider now a maximal subformula ξ of $\psi_{\bar{c}}$ that contains no temporal operator, which we call an FO component of $\psi_{\bar{c}}$. Note that ξ has no free variables (as variables previously free in ξ have been replaced by the constant symbols $\bar{c}$). Thus, ξ can be evaluated to true or false in every configuration of a run of $\mathcal{A}$. This allows treating every such ξ as a proposition. More precisely, for each FO component ξ of $\psi_{\bar{c}}$, let p_ξ be a propositional symbol. Let ${\psi_{\bar{c}}}^{aux}$ be the LTL formula obtained by replacing in $\psi_{\bar{c}}$ every FO component ξ

by p_ξ. For each configuration of $\mathcal{A}$, the truth value of p_ξ is defined as the truth value of ξ. Clearly, a run of $\mathcal{A}$ satisfies $\psi_{\bar{c}}$ iff it satisfies ${\psi_{\bar{c}}}^{aux}$. Specifically, for $i \geq 0$, let $\sigma(\rho_i)$ be the truth assignment to the propositions in ${\psi_{\bar{c}}}^{aux}$ such that p_ξ is true iff $\rho_i \models \xi$, and let $\sigma(\rho) = \{\sigma(\rho_i)\}_{i\geq 0}$. Let us denote by $A_{\psi_{\bar{c}}}$ the Büchi automaton corresponding to the propositional LTL formula ${\psi_{\bar{c}}}^{aux}$. A run of $A_{\psi_{\bar{c}}}$ on $\sigma(\rho)$ is an infinite sequence of states $q_0, s_0, s_1, \ldots, s_i, \ldots$ such that q_0 is the start state of $A_{\psi_{\bar{c}}}$, and $\langle q_0, \sigma(\rho_0), s_0\rangle$, $\langle s_i, \sigma(\rho_{i+1}), s_{i+1}\rangle$ are transitions in $A_{\psi_{\bar{c}}}$ for each $i \geq 0$. Clearly, $\rho \models \psi_{\bar{c}}$ iff there exists a run of $A_{\psi_{\bar{c}}}$ on input $\sigma(\rho)$ that goes through some accepting state, say f, infinitely often.

Example 4.3 The following LTL-FO property referring to Example 3.2 states that any shipped product must have previously been paid for.

$$\begin{aligned}
&\forall pid, pname, price \\
&(\ \text{UPP} \wedge \mathsf{pay}(price) \wedge \mathsf{button}(\text{``}\textit{authorize payment}\text{''}) \wedge \\
&\text{pick}(pid, price) \wedge \underline{\mathsf{catalog}}(pid, pname, price)) \\
&\qquad \mathbf{B} \\
&\neg(\overline{\mathsf{conf}}(name, price) \wedge \overline{\mathsf{ship}}(name, pid))
\end{aligned} \tag{4.1}$$

Property (4.1) is negated to the following formula ψ:

$$\begin{aligned}
&\exists pid, pname, price \\
&\neg(\ \text{UPP} \wedge \mathsf{pay}(price) \wedge \mathsf{button}(\text{``}\textit{authorize payment}\text{''}) \wedge \\
&\text{pick}(pid, price) \wedge \underline{\mathsf{catalog}}(pid, pname, price)) \\
&\qquad \mathbf{U} \\
&(\overline{\mathsf{conf}}(name, price) \wedge \overline{\mathsf{ship}}(name, pid))
\end{aligned} \tag{4.2}$$

Let $\bar{c}$ be a sequence of constants $pid_0, pname_0, price_0$. By replacing the existentially quantified variables with $\bar{c}$, we obtain $\psi_{\bar{c}}$:

$$\begin{aligned} \neg(\ \text{UPP} \wedge \mathsf{pay}(price_0) \wedge \mathsf{button}(\text{“}\textit{authorize payment}\text{”}) \wedge \\ \text{pick}(pid_0, price_0) \wedge \underline{\mathsf{catalog}}(pid_0, pname_0, price_0)) \\ \mathbf{U} \\ (\overline{\mathsf{conf}}(name, price_0) \wedge \overline{\mathsf{ship}}(name, pid_0)) \end{aligned} \tag{4.3}$$

which yields the propositional property $\psi_{\bar{c}}{}^{aux}$

$$p_1 \mathbf{U} p_2 \tag{4.4}$$

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 (4.4). We have already seen in Figure 2.2 the Büchi automaton corresponding to Property (4.4).

In the simple example above, the FO components of $\psi_{\bar{c}}$ happen to be quantifier free. In general however, FO components may have input-bounded quantifiers.

Next, we address the harder issue of the infinite number of configurations of $\mathcal{A}$. Here we make crucial use of the input-boundedness restriction. Let $\mathcal{A} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathcal{R} \rangle$ be an input-bounded ASM$^+$ transducer, and φ an input-bounded LTL-FO formula. Let $\psi_{\bar{c}}$ be obtained from φ as described above, for some sequence $\bar{c}$ of constant symbols. Let C be the set of constant symbols in the database schema $\mathbf{D}$. We can assume without loss of generality that $\bar{c}$ belong to C (otherwise we extend the database schema to include $\bar{c}$). Let D be a database instance of $\mathbf{D}$, and let $\rho = \{\langle S_i, I_i, P_i, A_i \rangle\}_{i \geq 0}$ be a run of $\mathcal{A}$ on D. Let C_D be the set of all domain elements that are interpretations of constant symbols in C in the instance D. We say that two instances H and H' over the same database schema are C_D-isomorphic iff there exists an isomorphism from H to H' that is the identity on C_D. The C_D-isomorphism type of H consists of all instances H' that are C_D-isomorphic to H. The critical observation is that, due to input-boundedness, the truth value of each FO component of $\psi_{\bar{c}}$ in a configuration $\langle S_i, I_i, P_i, A_i \rangle$ is completely determined by the restriction[1] of S_i and A_i to C_D, together with the C_D-isomorphism type of the subinstance of $\langle I_i, P_i, D \rangle$ restricted to C_D together with the elements in I_i, P_i. Since I_i and P_i contain at most one tuple each, the number of such C_D-isomorphism types is finite. Moreover, due to the input-boundedness of the state transition rules, the same information about $\langle S_{i+1}, I_{i+1}, P_{i+1}, A_{i+1} \rangle$ is determined by the corresponding information about $\langle S_i, I_i, P_i, A_i \rangle$. This will allow us to limit ourselves to inspecting a transition system whose configurations are the finitely many C_D-isomorphism types as above. This essentially reduces verification back to a classical model checking problem and, with some care, yields a PSPACE verification algorithm. We provide the details next.

For an instance K and a set T of elements, let $K|_T$ denote the restriction of K to T. Using the same notation as above, let $\rho_i = \langle S_i, I_i, P_i, A_i \rangle$ be a configuration in a run ρ of $\mathcal{A}$ on D. Let k be the arity of I, and C^i_{IP} consist of C_D together with all elements in $I_i \cup P_i$ (note that $I_i \cup P_i$ contains at most $2k$ elements). Let $D_{i,\bar{x}}$ be the restriction of D to C^i_{IP} together with witnesses to the existentially quantified variables $\bar{x}$ in the input-options formula $\exists \bar{x} \varphi_I(\bar{x})$ for I, satisfied by $D, P_i, I_i, S_i|C_D$. Let $\rho^{\downarrow}_i = \langle S_i|C_D, I_i, P_i, A_i|C_D, D_{i,\bar{x}} \rangle$. We refer to the sequence $\{\rho^{\downarrow}_i\}_{i \geq 0}$ as the *local run*[2] of ρ. We say that a sequence $\{\rho'_i\}_{i \geq 0}$ is a local run of $\mathcal{A}$ on D if it is the local run of some run of $\mathcal{A}$ on D.

Lemma 4.4. Let $\mathcal{A}$ be an input-bounded ASM$^+$ transducer, φ an input-bounded LTL-FO formula, and $\bar{c}, \psi_{\bar{c}}$, D, and ρ be as above. Let ξ be an FO component of $\psi_{\bar{c}}$. Then for each configuration ρ_i in the run ρ, $\rho_i \models \xi$ iff $\rho^{\downarrow}_i \models \xi$.

Proof. Let $\rho_i = \langle S_i, I_i, P_i, A_i \rangle$. We show by induction the following:

[1]The restriction of a database instance K to a set T of domain elements is the instance consisting of the tuples in K using only elements in T.

[2]Our local run is an extension of the notion of local run introduced in [48, 49].

(†) for every subformula $\xi'(\bar{x})$ of ξ with free variables $\bar{x}$, and sequence $\bar{e}$ of elements in C^i_{IP} of the same arity as $\bar{x}$, $\rho_i \models \xi'(\bar{x} \leftarrow \bar{e})$ iff $\rho_i^{\downarrow} \models \xi'(\bar{x} \leftarrow \bar{e})$.

As a consequence of (†), $\rho_i \models \xi$ iff $\rho_i^{\downarrow} \models \xi$, since ξ has no free variables.

Consider (†). We can assume wlog that ξ uses only $\wedge, \neg$ and $\exists$. For the basis, suppose $\xi'(\bar{x})$ is an atom $R(t_1, \ldots, t_m)$ where each t_i is an element in C_D or a variable in $\bar{x}$. If R is a state or action relation, all t_i's are elements in C_D by input boundedness, so (†) holds because $\rho_i^{\downarrow}$ retains $S|C_D$ and $A|C_D$. If R is an input or database relation, then again (†) holds because $\rho_i^{\downarrow}$ retains I_i, P_i, and $D|_{C^i_{IP}}$. Consider the induction step. If $\xi' = \xi_1 \wedge \xi_2$ or $\xi' = \neg\xi_1$ and ξ_1, ξ_2 satisfy (†), it immediately follows that ξ' satisfies (†). Now suppose $\xi'(\bar{x}) = \exists y(R(t_1, \ldots, t_k) \wedge \varphi(\bar{x}, y))$ where R is the input or previous input relation, each variable among the t_i's is either y or in $\bar{x}$ (at least one t_i is y by input boundedness), and (†) holds for $\varphi(\bar{x}, y)$. If R is empty in ρ_i then $\xi'(\bar{x} \leftarrow \bar{e})$ is false in both ρ_i and $\rho_i^{\downarrow}$, so (†) holds. If R is not empty, then $\rho_i \models \xi'(\bar{x} \leftarrow \bar{e})$ iff there exists c occurring in R such that $\rho_i \models R(t_1, \ldots, t_k)[\bar{x} \leftarrow \bar{e}, y \leftarrow c] \wedge \varphi(\bar{x}, y)[\bar{x} \leftarrow \bar{e}, y \leftarrow c]$. By the induction hypothesis, this happens iff $\rho_i^{\downarrow} \models R(t_1, \ldots, t_k)[\bar{x} \leftarrow \bar{e}, y \leftarrow c] \wedge \varphi(\bar{x}, y)[\bar{x} \leftarrow \bar{e}, y \leftarrow c]$, so $\rho_i^{\downarrow} \models \xi'(\bar{x} \leftarrow \bar{e})$, which shows (†). □

Lemma 4.4 shows that in a configuration ρ_i, the information relevant to satisfaction of $\psi_{\bar{c}}$ is captured by $\rho_i^{\downarrow}$. In other words, a run satisfies $\psi_{\bar{c}}$ iff its local run satisfies $\psi_{\bar{c}}$. Let $C_k = C \cup \{c_1, \ldots, c_{2k}\}$ where $c_1, \ldots, c_{2k}$ are distinct new elements (recall that k is the arity of I). Let m be the number of existentially quantified variables in the input-options rule for I, and $e_1, \ldots, e_m$ be m distinct new elements. Let $C_{km} = C_k \cup \{e_1, \ldots, e_m\}$. It will be convenient to assume, without loss of generality, than $C_{km} \subset \mathbf{dom}_\infty$. We can represent the C_D-isomorphism type of $\rho_i^{\downarrow}$ by an instance whose domain is C_{km}, which we denote $\tau(\rho_i)$. Thus, Lemma 4.4 says that $\sigma(\rho_i) = \sigma(\rho_i^{\downarrow}) = \sigma(\tau(\rho_i))$. Note that the domain of $\tau(\rho_i)$ is the *fixed* set of elements C_{km}, whereas the domain of $\rho_i^{\downarrow}$ depends on i.

We wish to lift the above from individual configurations to entire runs. More precisely, we would like to be able to generate *sequences* $\{\tau_i\}_{i \geq 0}$ of instances using elements in C_{km} that correspond precisely to the sequences of C_D-isomorphism types of $\{\rho_i^{\downarrow}\}_{i\geq 0}$ for runs $\{\rho_i\}_{i\geq 0}$ of $\mathcal{A}$ on each database D. We formalize this using the notion of *pseudorun*.

Definition 4.5. Let $\mathcal{A} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathcal{R}\rangle$ be an input-bounded ASM$^+$ transducer, and let $\psi_{\bar{c}}$, C, C_k, and C_{km} be defined as above. A *C-pseudorun* of $\mathcal{A}$ is a sequence of instances $\{\langle S_i, I_i, P_i, A_i, D_i\rangle\}_{i\geq 0}$ with elements in C_{km} such that:

1. S_i, I_i, P_i, A_i, D_i are state, input, previous input, action, and database instances;

2. all D_i provide the same interpretation for the constants in C, and each constant in C is interpreted within C;

3. S_i and A_i are instances using only elements in C;

4. I_i and P_i contain at most one tuple each, using elements in C_k;

5. for each $i \geq 0$, $P_{i+1} = I_i$ and $D_i|_{C\cup dom(I_i)} = D_{i+1}|_{C\cup dom(P_{i+1})}$;

6. for each $i \geq 0$, if $I_i = \{\bar{a}\}$ and the input-options formula for I is $\exists\bar{x}\varphi_I(\bar{x})$, then $\langle D_i, P_i, S_i\rangle \models \varphi_I(\bar{x} \leftarrow \bar{a})$;

7. $S_0 = A_0 = P_0 = \emptyset$;

8. for each $i \geq 0$, $S_{i+1} = S'_{i+1}|_C$ and $A_{i+1} = A'_{i+1}|_C$, where S'_{i+1} and A'_{i+1} are the state and action relations defined from the configuration $\langle S_i, I_i, P_i, A_i\rangle$ of $\mathcal{A}$ and database D_i, according to the rules of $\mathcal{A}$.

Before proceeding, we make another useful observation showing that we can confine the search for runs of $\mathcal{A}$ satisfying $\psi_{\bar{c}}$ to *periodic* runs. Specifically, a run $\{\rho_i\}_{i\geq 0}$ is periodic iff there exist $n \geq 0$ and $p \geq 0$ such that $\rho_i = \rho_{i+p}$ for every $i \geq n$.

Lemma 4.6. Let $\mathcal{A}$ be an input-bounded ASM$^+$ transducer, φ an input-bounded LTL-FO formula, and $\bar{c}, \psi_{\bar{c}}$ be as above. Let D be a database instance. If there exists a run of $\mathcal{A}$ on D satisfying $\psi_{\bar{c}}$ then there exists a periodic run of $\mathcal{A}$ on D satisfying $\psi_{\bar{c}}$.

Proof. Consider a run $\rho = \{\rho_i\}_{i \geq 0}$ of $\mathcal{A}$ on D satisfying $\psi_{\bar{c}}$. Let $A_{\psi_{\bar{c}}}$ be the Büchi automaton corresponding to the propositional LTL formula ${\psi_{\bar{c}}}^{aux}$. Recall that for $i \geq 0$, $\sigma(\rho_i)$ denotes the truth assignment to the propositions in ${\psi_{\bar{c}}}^{aux}$ such that p_ξ is true iff $\rho_i \models \xi$, and $\sigma(\rho) = \{\sigma(\rho_i)\}_{i \geq 0}$. Since $\rho \models \psi_{\bar{c}}$, there exists a run $q_0, s_0, s_1, \ldots, s_i, \ldots$ of $A_{\psi_{\bar{c}}}$ on input $\sigma(\rho)$ that goes through some accepting state, say f, infinitely often. Since there are finitely many distinct instances ρ_i in ρ, there must exist $n < j$, such that $s_n = s_j = f$ and $\rho_n = \rho_j$. Let $p = j - n$. Consider the sequence $\rho' = \{\rho'_i\}_{i \geq 0}$ defined by $\rho'_m = \rho_m$ for $0 \leq m \leq j$ and $\rho'_m = \rho'_{m-p}$ for $m > j$. Clearly, ρ' is a periodic run of $\mathcal{A}$ on D and $A_{\psi_{\bar{c}}}$ accepts $\sigma(\rho')$, so $\rho' \models \psi_{\bar{c}}$. $\square$

We next show the following key connection between pseudoruns and actual runs.

Lemma 4.7. Let $\mathcal{A}$ be an input-bounded ASM$^+$ transducer and φ an input-bounded LTL-FO formula. Let C, $\bar{c}$, and $\psi_{\bar{c}}$ be as above. The following are equivalent:

(i) there exists some periodic run ρ of $\mathcal{A}$ on a database D such that $\rho \models \psi_{\bar{c}}$, and

(ii) there exists some periodic C-pseudorun τ of $\mathcal{A}$ such that $\tau \models \psi_{\bar{c}}$.

Proof. Consider (i) $\rightarrow$ (ii). Let $\rho = \{\rho_i\}_{i \geq 0}$ be a periodic run of $\mathcal{A}$ on a database D, that satisfies $\psi_{\bar{c}}$. Recall that we assume without loss of generality that $C \subseteq \mathbf{dom}_\infty$. We can further assume that $C_D = \mathit{dom}(D) \cap C$ (otherwise we take an isomorphic image of D on which this is true). We first construct a C-pseudorun τ of $\mathcal{A}$ such that $\sigma(\rho) = \sigma(\tau)$. In particular, τ satisfies $\psi_{\bar{c}}$. From τ one can then easily construct a periodic C-pseudorun of $\mathcal{A}$ satisfying $\psi_{\bar{c}}$, as in Lemma 4.6.

Consider $\{\rho_i^{\downarrow}\}_{i \geq 0}$, where $\rho_i^{\downarrow} = \langle S_i^{\rho}, I_i^{\rho}, P_i^{\rho}, A_i^{\rho}, D_i^{\rho} \rangle$. We define by induction a sequence of one-to one mappings $\{f_i\}_{i \geq 0}$, where f_i maps $dom(\rho_i^{\downarrow})$ to C_{km} and is the identity on C_D:

- f_0 is an arbitrary one-to-one mapping from $dom(\rho_0^{\downarrow})$ to C_{km} that fixes C_D and maps C_{IP}^0 to C_k;

- $f_{i+1}|dom(P_{i+1}) = f_i|dom(I_i)$ and f_{i+1} is an arbitrary extension of $f_{i+1}|dom(P_{i+1})$ to a one-to-one mapping from $dom(\rho_{i+1}^{\downarrow})$ to C_{km} that is the identity on C_D and maps C_{IP}^{i+1} to C_k.

Now let $\tau_i = f_i(\rho_i^{\downarrow})$ for each $i \geq 0$ (note that in particular the constants C are interpreted by τ_i as in D). By definition, τ_i and $\rho_i^{\downarrow}$ are C-isomorphic. It remains to show that $\{\tau_i\}_{i \geq 0}$ is a C-pseudorun of $\mathcal{A}$. Parts (1-6) of Definition 4.5 are obviously satisfied. Consider (7). Consider τ_i and τ_{i+1} for $i \geq 0$. Let R be a state relation and $R_i = S_i(R)$, $R_{i+1} = S_{i+1}(R)$. Suppose $\varphi_R^+(\bar{x})$ and $\varphi_R^-(\bar{x})$ are the input-bounded formulas of $\mathcal{A}$ defining the tuples to be inserted, respectively deleted from R. Let $\bar{e}$ be a sequence of elements in C of the same arity as $\bar{x}$. Since $\rho_i^{\downarrow}$ is C-isomorphic to τ_i and φ_R^+, φ_R^- are input bounded, one can show similarly to (†) in the proof of Lemma 4.4 that $\tau_i \models \varphi_R^+(\bar{e})$ iff $\rho_i^{\downarrow} \models \varphi_R^+(\bar{e})$, and also $\tau_i \models \varphi_R^-(\bar{e})$ iff $\rho_i^{\downarrow} \models \varphi_R^-(\bar{e})$. Also by (†), $\rho_i^{\downarrow} \models \varphi_R^+(\bar{e})$ iff $\rho_i \models \varphi_R^+(\bar{e})$ and $\rho_i^{\downarrow} \models \varphi_R^-(\bar{e})$ iff $\rho_i \models \varphi_R^-(\bar{e})$. It follows that $\bar{e}$ is inserted/deleted from R in the transition from ρ_i to ρ_{i+1} iff it is inserted/deleted in the transition from $\rho_i^{\downarrow}$ to $\rho_{i+1}^{\downarrow}$ iff it is inserted/deleted in the transition from τ_i to τ_{i+1} according to the state rule for R. Since by definition R is the same in $\rho_i, \rho_i^{\downarrow}$, and τ_i, and R is also the same in $\rho_{i+1}, \rho_{i+1}^{\downarrow}$, and τ_{i+1}, (7) holds for state relations. A similar argument shows that (7) also holds for action relations.

Now consider the harder (ii) $\rightarrow$ (i). Let $\tau = \{\tau_i\}_{i \geq 0}$ be a periodic C-pseudorun satisfying $\psi_{\bar{c}}$, where $\tau_i = \langle S_i^{\tau}, I_i^{\tau}, P_i^{\tau}, A_i^{\tau}, D_i^{\tau} \rangle$. We define a database D interpreting the constant symbols in C in the same way as τ, and a periodic run $\rho = \{\langle S_i, I_i, P_i, A_i \rangle\}_{i \geq 0}$ of $\mathcal{A}$ on D such that for each i, $\rho_i^{\downarrow}$ is C-isomorphic to τ_i. In particular, $\sigma(\tau) = \sigma(\rho)$, so $\rho \models \psi_{\bar{c}}$.

Recall that τ is a sequence of instances using elements in C_{km} and $\mathbf{dom}_\infty$ is an infinite domain. To construct a database D and run ρ of $\mathcal{A}$ on D, we will assign values in $\mathbf{dom}_\infty$ to occurrences of the elements in C_{km} in the different configurations of τ. The challenge is to do so while using only finitely many values.

Consider the elements in $C_k - C$. Some of these elements occurring in different configurations of τ must be assigned the same value, while others are independent of each other. We denote by $\langle i, a\rangle$ the occurrence of a in τ_i, where $i \geq 0$ and $a \in C_k - C$. To capture the required equalities among elements in different configurations, we define the following equivalence relation $\equiv$ on occurrences $\langle i, a\rangle$. First, let $\langle i, a\rangle \approx \langle i+1, a\rangle$ iff a occurs in I_i (and therefore in P_{i+1}). Next, let $\equiv$ be the symmetric, reflexive, transitive closure of $\approx$. Let f be a mapping from the set of all occurrences of elements in C_{km} in τ to $\mathbf{dom}_\infty$ such that $f(\langle i, a\rangle) = f(\langle j, a\rangle)$ iff $\langle i, a\rangle \equiv \langle j, a\rangle$, and f is the identity on C. Note that the range of f is infinite. Consider the sequence $f(\tau) = \{f(\tau_i)\}_{i\geq 0}$, where $f(\tau_i) = \langle f(S_i^\tau), f(I_i^\tau), f(P_i^\tau), f(A_i^\tau), f(D_i^\tau)\rangle$. Let $D_f = \cup_{i\geq 0} f(D_i^\tau)$. We first show that $f(\tau)$ satisfies the definition of a local run of $\mathcal{A}$ on D_f, except for the requirement that D_f be finite. Given the definition of pseudorun, and since τ_i and $f(\tau_i)$ are C-isomorphic, it is enough to show that

(‡) $D_f|dom(f(\tau_i)) = f(D_i^\tau)$ for all $i \geq 0$.

Consider $a \in range(f)$. The *span* of a is $\{i \mid \exists b\ f(\langle i, b\rangle) = a\}$. From the definition of f it follows that the span of each $a \notin C$ is an interval, possibly infinite to the right. Now consider (‡). Suppose towards a contradiction that $D_f|dom(f(\tau_i)) \neq f(D_i^\tau)$ for some $i \geq 0$. Since by definition $f(D_i^\tau) \subseteq D_f|dom(f(\tau_i))$, it follows that $f(D_i^\tau) \subset D_f|dom(f(\tau_i))$. Thus, for some database relation R, there exists a tuple t such that $dom(t) \subseteq dom(f(\tau_i))$, $R(t)$ holds in $D_f|dom(f(\tau_i))$ but $R(t)$ does not hold in $f(D_i^\tau)$. In particular, there must exist $j \neq i$ such that $R(t)$ holds in $f(D_j^\tau)$. Since $t \in dom(\tau_i) \cap dom(\tau_j)$, it follows that $i, j \in span(a)$ for each $a \in dom(t)$. However, from (4) in the definition of pseudoruns, it follows that $f(D_i^\tau)|_{dom(\tau_i)\cap dom(\tau_j)} = f(D_j^\tau)|_{dom(\tau_i)\cap dom(\tau_j)}$, so $R(t)$ holds in $f(D_i^\tau)$ iff $R(t)$ holds in $f(D_j^\tau)$. This is a contradiction. Thus, (‡) holds.

We next construct from $f(\tau)$ a local run, and then a run, whose universe is finite. Intuitively, this involves some "surgery" on $f(\tau)$, using a pumping argument. The main idea is the following. Note that f maps elements in each configuration of τ that are not in C to *new* elements in $\mathbf{dom}_\infty$, yielding the infinite universe of $f(\tau)$. It turns out that values can be assigned more economically: if two configurations τ_α and τ_β are isomorphic and far enough apart, the values assigned by f for τ_β can be reused for τ_α. Based on this observation, we can modify f so that its range has only finitely many values. We next formalize this argument.

We need to give special treatment to elements in $range(f)$ whose span is infinite. Since I contains only one tuple of arity k, it follows that at most k elements in $range(f) - C$ may have infinite span. Let R_k consist of all such elements (at most k). Let $N > 0$ be such that all elements in R_k occur in every $f(\tau_i)$ for $i \geq N$. From the periodicity of τ and the fact that all elements not in $R_k \cup C$ have finite span, it follows that there exist α, β, $N < \alpha < \beta$, where $\beta - \alpha$ is a sufficiently large multiple of the least period of τ, such that:

(a) $\tau_\alpha = \tau_\beta$ and $\tau_i = \tau_{i+p}$ for all $i \geq \alpha$, where $p = \beta - \alpha$,

(b) $f(\tau_\alpha)$ and $f(\tau_\beta)$ are $(R_k \cup C)$-isomorphic,

(c) there are no elements $a_\alpha, a_\beta, d \in range(f) - (R_k \cup C)$, such that $a_\alpha \in dom(f(\tau_\alpha))$, $a_\beta \in dom(f(\tau_\beta))$, and $span(d) \cap span(a_\alpha) \neq \emptyset$ and $span(d) \cap span(a_\beta) \neq \emptyset$.

Let h be an $(R_k \cup C)$-isomorphism from $f(\tau_\alpha)$ to $f(\tau_\beta)$. Consider the sequence of configurations $f(\tau_\alpha) \ldots f(\tau_{\beta-1})$. Let $\bar{\tau}_\alpha$ be the prefix of $f(\tau_\alpha) \ldots f(\tau_{\beta-1})$ consisting of all configurations in the sequence whose domain intersects $dom(f(\tau_\alpha)) - (R_k \cup C)$, and let $\bar{\tau}_{\beta-1}$ be the suffix of $f(\tau_\alpha) \ldots f(\tau_{\beta-1})$ consisting of all configurations in the sequence whose domain intersects $dom(f(\tau_\beta)) - (R_k \cup C)$. By (c), $\bar{\tau}_\alpha$ and $\bar{\tau}_{\beta-1}$ do not overlap, so $f(\tau_\alpha) \ldots f(\tau_{\beta-1}) = \bar{\tau}_\alpha \bar{\tau} \bar{\tau}_{\beta-1}$ for some sequence of configurations $\bar{\tau}$. Let $f_h(\bar{\tau}_\alpha)$ be obtained from $\bar{\tau}_\alpha$ by replacing each element $a \in dom(f(\tau_\alpha))$

by $h(a) \in dom(f(\tau_\beta))$. Intuitively, (c) guarantees that α and β are far enough apart that replacing a by $h(a)$ in $\bar{\tau}_\alpha$ as above creates no interference. Note that the sequence $f_h(\bar{\tau}_\alpha)\bar{\tau}\bar{\tau}_{\beta-1}$ starts with $f(\tau_\beta)$. Consider the periodic sequence $\rho^\downarrow = \{\rho^\downarrow_i\}_{i\geq 0}$ obtained by concatenating $f_h(\bar{\tau}_\alpha)\bar{\tau}\bar{\tau}_{\beta-1}$ infinitely many times to the right of $f(\tau_0), \ldots, f(\tau_{\beta-1})$. It is easily seen that τ_i and $\rho^\downarrow_i$ are C-isomorphic for all $i \geq 0$. In particular, $\sigma(\tau) = \sigma(\rho^\downarrow)$, so $\rho^\downarrow \models \psi_{\bar{c}}$. It is enough to show that $\rho^\downarrow$ is a local run of $\mathcal{A}$ for some finite database. Let $\rho^\downarrow = \{\langle S_i, I_i, P_i, A_i, D_i\rangle\}_{i\geq 0}$. Let $D = \cup_{i\geq 0} D_i$. From the periodicity of $\rho^\downarrow$ it follows that D is finite. We claim, similarly to (‡), that

(*) $D|dom(\rho^\downarrow_i) = D_i$ for each $i \geq 0$.

Suppose towards a contradiction that there exist i, j, $i \neq j$, a database relation R, and a tuple t such that $dom(t) \subseteq dom(\rho^\downarrow_i) \cap dom(\rho^\downarrow_j)$, $D_i \models R(t)$, and $D_j \not\models R(t)$. Because of (‡), $\rho^\downarrow_i$ and $\rho^\downarrow_j$ cannot both be configurations already appearing in $f(\tau)$. Thus, at least one of $\rho^\downarrow_i$ and $\rho^\downarrow_j$ are configurations in $f_h(\bar{\tau}_\alpha)$. There are three cases to consider:

1. $\rho^\downarrow_i$ and $\rho^\downarrow_j$ are both in $f_h(\bar{\tau}_\alpha)$. Due to (c) above, no $b \in range(h)$ occurs in $dom(f(D^\tau_i)) \cup dom(f(D^\tau_j))$. But then h can be extended to an isomorphism (by the identity) to the entire $dom(f(D^\tau_i)) \cup dom(f(D^\tau_j))$, and h^{-1} is also an isomorphism. Thus, $dom(h^{-1}(t)) \in dom(f(\tau_i)) \cap dom(f(\tau_j))$, $f(D^\tau_i) \models R(h^{-1}(t))$, and $f(D^\tau_j) \not\models R(h^{-1}(t))$. However, this is a contradiction with (‡).

2. $\rho^\downarrow_i$ occurs in $f_h(\bar{\tau}_\alpha)$ and $\rho^\downarrow_j$ does not. Thus, $dom(t) \subseteq dom(h(f(D^\tau_i))) \cap dom(f(D^\tau_j))$, $h(f(D^\tau_i)) \models R(t)$, and $f(D^\tau_j) \not\models R(t)$. Suppose $dom(t) \cap range(h) = \emptyset$. Then $dom(t) \subseteq dom(f(D^\tau_i))$. Thus, $dom(t) \subseteq dom(f(D^\tau_i)) \cap dom(f(D^\tau_j))$, $f(D^\tau_i) \models R(t)$, and $f(D^\tau_j) \not\models R(t)$. This contradicts (‡). Thus, $dom(t) \cap range(h) \neq \emptyset$. But then $dom(f(\tau_j)) \cap dom(f(\tau_\beta)) \neq \emptyset$. From (c) it then follows that $dom(t) \subseteq range(h)$, so $dom(t) \subseteq dom(f(\tau_\beta))$. By (‡), $f(D^\tau_\beta) \not\models R(t)$ since $dom(t) \subseteq dom(f(\tau_\beta)) \cap dom(f(\tau_j))$ and $f(D^\tau_j) \not\models R(t)$.

Since $dom(t) \subseteq dom(f(\tau_\beta))$, $dom(h^{-1}(t)) \subseteq dom(f(\tau_\alpha))$, so $dom(h^{-1}(t)) \subseteq dom(\tau_\alpha) \cap dom(f(\tau_i))$. Again by (‡) $f(D^\tau_\alpha) \models R(h^{-1}(t))$ because $f(D^\tau_i) \models R(h^{-1}(t))$. Thus, $f(D^\tau_\alpha) \models R(h^{-1}(t))$ and $f(D^\tau_\beta) \not\models R(t)$. However, this contradicts the fact that h is an $(R_k \cup C)$-isomorphism from $f(\tau_\alpha)$ to $f(\tau_\beta)$.

3. $\rho^\downarrow_j$ occurs in $f_h(\bar{\tau}_\alpha)$ and $\rho^\downarrow_i$ does not. The proof is similar to (2) and is omitted.

Thus, $(*)$ is proven.

Finally, let $\rho = \{\langle S'_i, I_i, P_i, A'_i\rangle\}_{i \geq 0}$ be obtained from $\rho^\downarrow$ by computing for each $i \geq 0$, S'_{i+1} and A'_{i+1} from $\langle S'_i, I_i, P_i\rangle$ and D, using the state and action rules of $\mathcal{A}$. From $(*)$, the definition of pseudorun, and the construction of $\rho^\downarrow$, it is clear that ρ is a run of $\mathcal{A}$ on database D, and $\rho^\downarrow$ is the local run of ρ. In particular, $\sigma(\tau) = \sigma(\rho)$, so $\rho \models \psi_{\bar{c}}$. Also, ρ is periodic. This completes the proof. ☐

Lemma 4.7 says that in order to determine whether $\mathcal{A}$ satisfies $\psi_{\bar{c}}$, it is enough to focus on periodic C-pseudoruns of $\mathcal{A}$. Summarizing the above development, we can now describe a non-deterministic PSPACE verification algorithm for input-bounded ASM$^+$ transducers and input-bounded LTL-FO properties.

The input to the algorithm is an input-bounded ASM$^+$ transducer $\mathcal{A}$ and an input-bounded LTL-FO formula φ. Let $\exists \bar{x} \psi(\bar{x})$ be the negation of φ and let $\bar{c}$ be a sequence of constant symbols, one for each variable in $\bar{x}$. Let C consist of $\bar{c}$ together with all constant symbols used in the specification of $\mathcal{A}$ or in φ. Guess an interpretation of the constants in C by values in C. Let $\psi_{\bar{c}} = \psi[\bar{x} \leftarrow \bar{c}]$ and let ${\psi_{\bar{c}}}^{aux}$ be the propositional LTL formula obtained by replacing each FO component ξ of $\psi_{\bar{c}}$ by a propositional symbol p_ξ. Let $A_{\psi_{\bar{c}}}$ be the Büchi automaton corresponding to ${\psi_{\bar{c}}}^{aux}$. Let $C_{km} = C \cup \{c_1, \ldots, c_{2k}\} \cup \{e_1, \ldots e_m\}$, where the c_i's and e_j's are distinct new elements. We use the following non-deterministic PSPACE algorithms:

- Büchi-Next: on input $({\psi_{\bar{c}}}^{aux}, s, \sigma)$, where s is a state of $A_{\psi_{\bar{c}}}$ and σ is a truth assignment to the propositions in ${\psi_{\bar{c}}}^{aux}$, the algorithm[3] returns a state s' of $A_{\psi_{\bar{c}}}$ such that $\langle s, \sigma, s'\rangle$ is a transition in $A_{\psi_{\bar{c}}}$.

[3]As noted earlier, the existence of such a PSPACE algorithm is a classical result shown in [47].

- Pseudorun-Next: given as input a configuration τ in a C-pseudorun of $\mathcal{A}$, output a possible next configuration τ' in the pseudorun.

The algorithm now proceeds as follows:

1. flag := 0;
2. set τ_0 to an initial configuration of a C-pseudorun of $\mathcal{A}$;
3. set s_0 to some output of Büchi-Next($\psi_{\bar{c}}, q_0, \sigma(\tau_0)$), where q_0 is the start state of $A_{\psi_{\bar{c}}}$;
4. set (s, τ) to (s_0, τ_0);
5. if flag $= 0$ and s is an accepting state of $A_{\psi_{\bar{c}}}$ then non-deterministically continue or set $(\bar{s}, \bar{\tau})$ to (s, τ) and set flag:= 1;
6. set τ to Pseudorun-Next(τ) and s to Büchi-Next($\psi_{\bar{c}}, s, \sigma(\tau)$);
7. if flag = 1 and $(s, \tau) = (\bar{s}, \bar{\tau})$ then output YES and stop; otherwise, go to 5.

Clearly, the above nondeterministic PSPACE algorithm accepts iff there exists a periodic C-pseudorun of $\mathcal{A}$ accepted by $A_{\psi_{\bar{c}}}$. Observe that if the arity of relations in the schema of $\mathcal{A}$ is not bounded, the above algorithm is in EXSPACE. This establishes the following.

Theorem 4.8. It is decidable, given an input-bounded ASM$^+$ transducer $\mathcal{A}$ and an input-bounded LTL-FO formula φ, whether every run of $\mathcal{A}$ satisfies φ. Furthermore, the complexity of the decision problem is PSPACE for fixed arity schemas, and EXPSPACE otherwise.

Theorem 4.8 in conjunction with Lemma 4.23 complete the proof of the main result of the section, Theorem 4.20. We note that the PSPACE algorithm described above provides the basis for a practical implementation of a verifier for Web applications. Such an implementation, including additional heuristics that improve the practical performance of the algorithm, is described in [20, 18]. The implementation turns out to be surprisingly effective, with verification times of under one minute in a battery of experiments.

4.1.2 Boundaries of decidability

One may wonder whether the input-boundedness restriction can be relaxed without affecting decidability of verification. Unfortunately, even small relaxations 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 preceding one.

(iv) relaxing the input-bounded restriction on properties to express functional dependencies (FDs) on the database relations,[4] and

(v) extending LTL-FO formulas with path quantification.

Non-Grounded State Atom in Input Option Rule

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 4.9. There exists a fixed input-bounded LTL-FO formula φ for which it is undecidable, given an input-bounded ASM$^+$ transducer $\mathcal{A}$ with input options

[4]Note that any FD can be expressed as a First-Order sentence f, so checking that ASM$^+$ transducer $\mathcal{A}$ satisfies property φ provided that its database satisfies FD f reduces to the standard verification problem $\mathcal{A} \models f \rightarrow \varphi$. The property $f \rightarrow \varphi$ however is not input-bounded.

defined by quantifier-free FO formulas over database and state relations, whether $\mathcal{A} \models \varphi$.

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 from it an ASM$^+$ transducer $\mathcal{A}$ as follows. 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 $\mathcal{A}$ 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, $\mathcal{A} \models \forall x \forall y \forall u \mathbf{G}(\neg T(x, y, u, h))$.

We now outline the construction of $\mathcal{A}$ in more detail. The database schema of $\mathcal{A}$ 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 (unary) and H (4-ary).

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):

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

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:

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

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

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 $\mathcal{A}$ satisfies the formula $\varphi = \forall x \forall y \forall u \mathbf{G}(\neg T(x,y,u,h))$. $\square$

State Projection

We next consider extension (ii): we relax input-boundedness of rules by allowing projections of state relations. We call an ASM$^+$ transducer *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 4.10. It is undecidable, given an input-bounded ASM$^+$ transducer $\mathcal{A}$ with state projections and input-bounded LTL-FO sentence φ, whether $\mathcal{A} \models \varphi$.

Proof. The proof is by reduction of the implication problem for functional and inclusion dependencies, known to be undecidable [12]. Recall that a functional dependency (FD) over relation schema S of arity k is an expression $X \rightarrow Y$ where $X, Y \subseteq \{1,\ldots,k\}$. An instance Z over S satisfies an FD $X \rightarrow Y$ iff whenever two tuples in Z agree on X they also agree on Y. An inclusion dependency (ID) over

S is an expression $[X] \subseteq [Y]$ where $X, B \subseteq \{1, \ldots, k\}$ and X and Y have the same size. An instance Z over S satisfies $[X] \subseteq [Y]$ iff for each tuple u in Z there exists a tuple v in Z such that $u|X = v|Y$. The implication problem for FDs and IDs is to determine, given a set Δ of FDs and IDs over S, and f an FD over S, whether Δ implies f (i.e., whether every instance over S satisfying Δ also satisfies f).

Let Δ be a set of FDs and IDs over a relation S, and f an FD over the same relation. We can assume without loss of generality that all FDs have singletons on the righthand side, and we denote for simplicity $X \to \{A\}$ by $X \to A$. We construct an input-bounded ASM$^+$ transducer $\mathcal{A}$ with state projections and an input-bounded LTL-FO sentence φ such that $\Delta \models f$ iff $\mathcal{A} \models \varphi$.

Let $\mathcal{A} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathcal{R} \rangle$ where $\mathbf{D} = \{R\}$, $\mathbf{A} = \emptyset$, $\mathbf{I} = \{I, \mathit{done}\}$ where I has the same arity as S and *done* is propositional, and $\mathbf{S}$ consists of the following relations:

- the relation S;
- two propositions stop_1,stop_2;
- for each ID σ of the form $[X] \subseteq [Y]$ in Δ, a relation S_X of arity $|X|$, a relation S_Y of arity $|Y|$, a relation S_X^σ of arity $|X|$, and a proposition viol_σ;
- for each FD σ of the form $X \to A$ in $\Delta \cup \{f\}$ a relation S_{XA} of arity $|XA|$, a relation $S^\sigma_{XA_1A_2}$ of arity $|XA| + 1$, and a proposition viol_σ.

Next, let $\mathcal{R}$ be defined as follows. The input option rule for I defines the cross-product of the active domain given by the database relation R. The state rules consist of the following:

$$\begin{array}{rcl} S(\bar{x}) & \leftarrow & I(\bar{x}) \wedge \neg \mathit{stop}_1 \\ \mathit{stop}_1 & \leftarrow & \mathit{done} \\ \mathit{stop}_2 & \leftarrow & \mathit{stop}_1 \end{array}$$

for each ID σ of the form $[X] \subseteq [Y]$ in Δ, the following rules (where $\pi_X(S)$ denotes the projection of S on X):

$$\begin{array}{lcl} S_X & \leftarrow & \pi_X(S) \\ S_Y & \leftarrow & \pi_Y(S) \\ S_X^\sigma(\bar{x}) & \leftarrow & S_X(\bar{x}) \wedge \neg S_Y(\bar{x}) \wedge \mathit{stop}_2 \\ \mathit{viol}_\sigma & \leftarrow & \exists \bar{x}\ S_X^\sigma(\bar{x}) \end{array}$$

for each FD σ of the form $X \rightarrow A$ in $\Delta \cup \{f\}$, the rules:

$$\begin{array}{lcl} S_{XA} & \leftarrow & \pi_{XA}(S) \\ S_{XA_1A_2}^\sigma(\bar{x}, a_1, a_2) & \leftarrow & S_{XA}(\bar{x}a_1) \wedge S_{XA}(\bar{x}a_2) \wedge a_1 \neq a_2 \wedge \mathit{stop}_2 \\ \mathit{viol}_\sigma & \leftarrow & \exists \bar{x} \exists a_1 \exists a_2 S_{XA_1A_2}^\sigma(\bar{x}, a_1, a_2) \end{array}$$

Intuitively, the state relation S is populated by repeated inputs, until *done* is set to true, which is remembered in the state propositions stop_1 and stop_2 (stop_2 is needed for timing reasons, to ensure that violations are not tested too early). The rules check for violations of the dependencies in Δ, so that viol_σ is set to true iff S violates σ.

Note that all rules are input bounded, except those consisting of projections of state relations. Next, let ξ be the input-bounded LTL-FO sentence

$$G(\neg \mathit{done}) \vee [F(\mathit{done}) \wedge (F(\bigvee_{\sigma \in \Delta} \mathit{viol}_\sigma) \vee G(\psi_f))]$$

where ψ_f is the formula $\neg S_{XA_1A_2}^f(\bar{x}, a_1, a_2)$ whose universal closure states that the FD $f = X \rightarrow A$ is satisfied. Finally, let φ be the universal closure of ξ. Intuitively, φ states that either *done* is never set to true, or it is set to true and at least one of the constraints of Δ is violated, or f is satisfied. Thus, $\mathcal{A} \models \varphi$ iff $\Delta \models f$. $\square$

Lossless Input

We now deal with extension (iii). We say that an ASM$^+$ transducer has *lossless input* if the prev_I relations record *all* previous inputs to I in the current run.

Theorem 4.11. It is undecidable, given an input-bounded ASM$^+$ transducer $\mathcal{A}$ with lossless input and an input-bounded LTL-FO formula φ, whether $\mathcal{A} \models \varphi$.

Proof. Using lossless input, one can easily simulate first-order quantification by considering runs that provide values for the quantified variables as inputs. This allows to use a reduction of finite validity of FO sentences to the above verification problem. We illustrate the reduction for FO sentences of the form $\forall x \forall y \alpha(x, y)$ where α is a quantifier free formula over relational vocabulary $\{R\}$. Let $\mathcal{A} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathcal{R} \rangle$ be an ASM$^+$ transducer where $\mathbf{D} = \{R\}$, $\mathbf{I} = \{X, Y\}$ (X, Y are unary relations), $\mathbf{A} = \emptyset$, $\mathbf{S} = \{S_X, S_Y, done_x, true_\alpha\}$ (S_X, S_Y are unary and the other states are propositional). The input option rules are:

$$\begin{array}{rcl} Options_X(x) & \leftarrow & (\psi_{dom}(x) \wedge \neg done_x) \\ & & \vee (done_x \wedge S_X(x)) \\ Options_Y(y) & \leftarrow & done_x \wedge \psi_{dom}(y) \end{array}$$

where $\psi_{dom}(x)$ defines the active domain provided by R. The state rules are the following:

$$\begin{array}{rcl} S_X(x) & \leftarrow & X(x) \\ done_x & \leftarrow & \exists x X(x) \wedge (\neg done_x) \\ true_\alpha & \leftarrow & \exists x \forall y (Prev_X(x) \wedge Prev_Y(y) \wedge \alpha(x, y)) \end{array}$$

Note that a path in $\mathcal{T}_\mathcal{A}$ starts at *root*, then proceeds to the start configuration of a run on some database D. The first input provided is a value of x, which is remembered in the state relation S_X. In the next configuration, $done_x$ is true, the same value of x as previously chosen is provided again via input X, and an arbitrary value is provided for y by the input relation Y. In the following configuration $true_\alpha$ is true if $\alpha(x, y)$ is satisfied for the chosen values of x, y. Let φ be the LTL-FO sentence $\forall y G(Y(y) \rightarrow X(G(true_\alpha)))$. Clearly, $\mathcal{A} \models \varphi$ iff $\exists x \forall y \alpha(x, y)$ is valid. Note that $\mathcal{A}$ and φ are input bounded.

Clearly, $\mathcal{A} \models \varphi$ iff $\forall x \forall y \alpha(x, y)$ is valid since if $\mathcal{A} \models \varphi$, all run of $\mathcal{A}$ must satisfies φ no matter what x value are picked, thus $\forall x \forall y \alpha(x, y)$ is valid. On the other hand, if $\forall x \forall y \alpha(x, y)$ is valid, for every run of $\mathcal{A}$, no matter what y's value being picked, $true_\alpha$ will always be set on the next configuration, thus $\mathcal{A} \models \varphi$.

□

Database with Functional Dependencies

We show the undecidability of extension (iv) next, proving that in the presence of FDs the verification problem becomes undecidable even for strictly input-bounded specifications and properties. Given property φ and transducer $\mathcal{A}$ with set $\mathcal{F}$ of functional dependencies on its database schema, we say that $\mathcal{A}$ satisfies φ under $\mathcal{F}$, denoted $\mathcal{A} \models_{\mathcal{F}} \varphi$, iff for every database D which satisfies $\mathcal{F}$, all runs of $\mathcal{A}$ over D satisfy φ.

Theorem 4.12. It is undecidable, given an input-bounded ASM$^+$ transducer $\mathcal{A}$ with functional dependencies $\mathcal{F}$ on its database schema, and an input-bounded LTL-FO formula φ, whether $\mathcal{A} \models_{\mathcal{F}} \varphi$.

Proof. By reduction from the *Post Correspondence Problem (PCP).* Consider a PCP instance, i.e. two sequences of length n: $\{u_i\}_{1\leq i\leq n}, \{v_i\}_{1\leq i\leq n}$, where all u_i, v_j are non-empty words over the alphabet $\{0, 1\}$. A *solution* to $\mathcal{P}$ is a finite non-empty sequence $\sigma \in [1, \ldots, n]^*$ such that the two strings obtained by concatenating $u_{\sigma(1)}u_{\sigma(2)} \ldots u_{\sigma(k)}$ and $v_{\sigma(1)}v_{\sigma(2)} \ldots v_{\sigma(k)}$ are identical ($\sigma(i)$ is the element at position i in σ). We say that these strings are *generated* by the solution σ. We construct ASM$^+$ transducer $\mathcal{A}$, set $\mathcal{F}$ of FDs, and property φ such that $\mathcal{P}$ has a solution iff $\mathcal{A} \not\models_{\mathcal{F}} \varphi$.

$\mathcal{A}$ simulates the search for a PCP solution as follows. The database encodes a finite string θ intended to correspond to the string generated by a solution of $\mathcal{P}$. $\mathcal{A}$ non-deterministically picks a sequence of indexes from $[1, \ldots, n]$ (by repeatedly asking an external user to pick an input among the options $[1, \ldots, n]$). Upon receiving the index i, $\mathcal{A}$ tries to match the corresponding words u_i and v_i in parallel against θ, by maintaining two cursors U and V on θ, as well as a cursor on u_i and a cursor on v_i. The cursors advance in lock-step, being incremented only if they point to the same character. Initially, U and V start from the first position in θ. The property φ is satisfied only if for all j, upon finishing to fully match u_j and v_j, U and V never meet on θ. It is easy to see that, if the database encodes a string

θ, a run of $\mathcal{A}$ violates φ if and only if the sequence of indexes picked by the user is a solution to $\mathcal{P}$, which generates a prefix of θ.

θ is encoded using two binary database relations, $\underline{\mathsf{chain}}(s, t)$ (intended to contain as a subgraph a chain of directed $s \to t$ edges) and $\underline{\mathsf{char}}(i, c)$ (intended to label each node i in the chain with a character $c \in \{0, 1\}$). We pick $\mathcal{F}$ to enforce that $\underline{\mathsf{chain}}(s, t)$ satisfies the functional dependencies (FDs) $s \to t$ and $t \to s$ and $\underline{\mathsf{char}}$ satisfies the FD $i \to c$. The FDs on $\underline{\mathsf{chain}}$ ensure that nodes have in-degree and out-degree one, so $\underline{\mathsf{chain}}$ is a union of disjoint cycles and chains. The FD on $\underline{\mathsf{char}}$ will ensure that indexes are labeled uniquely, and the rules ensure that the labels are in $\{0, 1\}$ (the fact that 0 and 1 are distinct constants is stated in the property). To ensure that the cursors U and V progress along the same path without revisiting any node, we enforce that they start from the same position, a special node '\$', and never return to '\$'.

In detail, the schema of $\mathcal{A}$ consists of

- $\mathbf{D} = \{\underline{\mathsf{chain}}(s, t),\ \underline{\mathsf{char}}(i, c),\ '\$', 0, 1\}$ as described above ('\$', 0, and 1 are constants);

- $\mathbf{I} = \{\mathsf{I}(i), \mathsf{U}(x), \mathsf{V}(x)\}$. Intuitively, the user provides his pick of a word index in I, and U and V are the cursors on θ. The options provided to the user contain the immediate successors in $\underline{\mathsf{chain}}$ of the cursors at the previous input $prev_{\mathsf{U}}, prev_{\mathsf{V}}$. Of course, there is at most one successor due to the FDs on $\underline{\mathsf{chain}}$.

- $\mathbf{S}$ contains the following propositional states:

  - for each $1 \leq i \leq n$, each $1 \leq j \leq |u_i|$ and each $1 \leq k \leq |v_i|$, state U_i^j and state V_i^k (these play the role of cursors in the u_i and v_i words);

  - state done_u, set to true only when a full u_i word is matched; begun_u which, when set to false, signals that the matching of u_i words has not yet begun; similarly, states done_v and begun_v.

- $\mathbf{A} = \emptyset$;

$\mathcal{A}$ contains

- the input rules

$$\begin{aligned}
\mathit{Options}_{\mathsf{I}}(i) &\leftarrow (i = 1 \vee i = 2 \vee \ldots \vee i = n) \\
&\wedge\ \ (\neg\text{begun}_u \wedge \neg\text{begun}_v \vee \text{done}_u \wedge \text{done}_v) \\
\mathit{Options}_{\mathsf{U}}(t) &\leftarrow (\neg\text{begun}_u \wedge t =' \$') \\
&\vee\ \ \text{begun}_u \wedge \neg\text{done}_u \wedge \\
&\qquad \exists s \exists c\ \mathit{prev}_{\mathsf{U}}(s) \wedge \underline{\mathsf{chain}}(s,t) \wedge t \neq' \$' \wedge \underline{\mathsf{char}}(t,c) \\
&\qquad \wedge (\bigvee_{i,j} \mathit{prev}_{\mathsf{I}}(i) \wedge c = u_i(j) \wedge \mathrm{U}_i^j) \\
\mathit{Options}_{\mathsf{V}}(t) &\leftarrow (\neg\text{begun}_v \wedge t =' \$') \\
&\vee\ \ \text{begun}_v \wedge \neg\text{done}_v \wedge \\
&\qquad \exists s \exists c\ \mathit{prev}_{\mathsf{V}}(s) \wedge \underline{\mathsf{chain}}(s,t) \wedge t \neq' \$' \wedge \underline{\mathsf{char}}(t,c) \\
&\qquad \wedge (\bigvee_{i,k} \mathit{prev}_{\mathsf{I}}(i) \wedge c = v_i(k) \wedge \mathrm{V}_i^k))
\end{aligned}$$

- the state rules

$$\begin{aligned}
\text{begun}_u &\leftarrow \neg\text{begun}_u \wedge \exists t\ \mathsf{U}(t) \\
\text{begun}_v &\leftarrow \neg\text{begun}_v \wedge \exists t\ \mathsf{V}(t) \\
\text{done}_u &\leftarrow \exists t\ \mathsf{U}(t) \wedge (\bigvee_{i=1}^{n} \mathrm{U}_i^{|u_i|-1}) \\
\neg\text{done}_u &\leftarrow \text{done}_u \wedge \exists x\ \mathsf{I}(x) \\
\text{done}_v &\leftarrow \exists t\ \mathsf{V}(t) \wedge (\bigvee_{i=1}^{n} \mathrm{V}_i^{|v_i|-1}) \\
\neg\text{done}_v &\leftarrow \text{done}_v \wedge \exists x\ \mathsf{I}(x)
\end{aligned}$$

Moreover, for $1 \leq i \leq n$,

$$\begin{array}{rcll} \mathrm{U}_i^1 & \leftarrow & \mathsf{I}(i) & \\ \mathrm{U}_i^j & \leftarrow & \mathrm{U}_i^{j-1} \wedge \exists t\ \mathsf{U}(t) & \text{for } 1 < j \leq |u_i| \\ \neg\mathrm{U}_i^j & \leftarrow & \mathrm{U}_i^j \wedge \exists t\ \mathsf{U}(t) & \text{for } 1 \leq j \leq |u_i| \\ \mathrm{V}_i^1 & \leftarrow & \mathsf{I}(i) & \\ \mathrm{V}_i^j & \leftarrow & \mathrm{V}_i^{j-1} \wedge \exists t\ \mathsf{V}(t) & \text{for } 1 < j \leq |v_i| \\ \neg\mathrm{V}_i^j & \leftarrow & \mathrm{V}_i^j \wedge \exists t\ \mathsf{V}(t) & \text{for } 1 \leq j \leq |v_i| \end{array}$$

$\mathcal{F}$ consists of FDs $t \to s, s \to t$ on $\underline{\mathsf{chain}}$ and $i \to c$ on $\underline{\mathsf{char}}$.

The (input-bounded) property φ is

$$\forall t\ 0 \neq 1 \wedge \mathbf{G}\neg(prev_{\mathsf{U}}(t) \wedge prev_{\mathsf{V}}(t) \wedge \mathrm{done}_u \wedge \mathrm{done}_v)$$

$\square$

Extending LTL-FO Formulas with Path Quantification

Finally, we address the undecidability of extension (v).

Theorem 4.13. It is undecidable, given an input-bounded ASM$^+$ transducer $\mathcal{A}$ and input-bounded CTL-FO sentence φ, whether $\mathcal{A} \models \varphi$.

Proof. Using path quantifiers, one can easily simulate first-order quantification by considering runs that provide values for the quantified variables as inputs. This allows to use a reduction of finite validity of FO sentences to the above verification problem. We illustrate the reduction for FO sentences of the form $\exists x \forall y \alpha(x, y)$ where α is a quantifier free formula over relational vocabulary $\{R\}$. Let $\mathcal{A} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathcal{R} \rangle$ be an ASM$^+$ transducer where $\mathbf{D} = \{R\}$, $\mathbf{I} = \{X, Y\}$ (X, Y are unary relations), $\mathbf{A} = \emptyset$, $\mathbf{S} = \{S_X, S_Y, done_x, true_\alpha\}$ (S_X, S_Y are unary and the other states are propositional). The input option rules are:

$$\begin{array}{rcl} Options_X(x) & \leftarrow & (\psi_{dom}(x) \wedge \neg done_x) \vee (done_x \wedge S_X(x)) \\ Options_Y(y) & \leftarrow & done_x \wedge \psi_{dom}(y) \end{array}$$

where $\psi_{dom}(x)$ defines the active domain provided by R. The state rules are the following:

$$\begin{array}{rcl} S_X(x) & \leftarrow & X(x) \\ done_x & \leftarrow & \neg done_x \\ true_\alpha & \leftarrow & \exists x \exists y (X(x) \wedge Y(y) \wedge \alpha(x,y)) \end{array}$$

Note that a path in $\mathcal{T}_\mathcal{A}$ starts at *root*, then proceeds to the start configuration of a run on some database D. The first input provided is a value of x, which is remembered in the state relation S_X. In the next configuration, $done_x$ is true, the same value of x as previously chosen is provided again via input X, and an arbitrary value is provided for y by the input relation Y. In the following configuration $true_\alpha$ is true if $\alpha(x,y)$ is satisfied for the chosen values of x, y. Let φ be the CTL-FO sentence EXAXAX($true_\alpha$). Clearly, $\mathcal{A} \models \varphi$ iff $\exists x \forall y \alpha(x,y)$ is valid. Note that $\mathcal{A}$ and φ are input bounded (in fact φ is propositional, so in CTL). This proof is easily extended along the same lines to the general case. $\square$

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 [8].

4.1.3 Verification of branching-time properties

In this section we consider the verification of branching-time temporal properties of ASM$^+$ transducers. As noted in the previous section, the decidability results for input-bounded ASM$^+$ transducers 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). We next consider several restrictions leading to decidability of the verification problem for CTL$^{(*)}$-FO sentences.

Propositional input-bounded ASM$^+$ transducers The first restriction further limits input-bounded ASM$^+$ transducers by requiring all states to be propositional. Furthermore, no rules can use $\mathbf{Prev_I}$ atoms. We call such ASM$^+$ transducers *propositional*. In a propositional ASM$^+$ transducer, inputs can still be parameterized. The CTL* formulas we consider are propositional and use only state symbols. For a given ASM$^+$ transducer $\mathcal{A} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathcal{R} \rangle$, we denote by $\Sigma_{\mathcal{A}}$ the propositional vocabulary $\mathbf{S}$. We first show the following:

Theorem 4.14. Given a propositional, input-bounded ASM$^+$ transducer $\mathcal{A}$ and a CTL* formula φ over $\Sigma_{\mathcal{A}}$, it is decidable whether $\mathcal{A} \models \varphi$. The complexity of the decision procedure is CO-NEXPTIME if φ is in CTL, and EXPSPACE if φ is in CTL*.

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 $K_{\mathcal{A},D}$ over alphabet $\Sigma_{\mathcal{A}}$, of size exponential in $\Sigma_{\mathcal{A}}$, such that $\mathcal{T}_{\mathcal{A},D} \models \neg\varphi$ iff $K_{\mathcal{A},D} \models \neg\varphi$. This allows us to use known model-checking techniques for CTL$^{(*)}$ on Kripke structures to verify whether $\mathcal{T}_{\mathcal{A},D} \models \neg\varphi$.

We start with the following:

Lemma 4.15. Let $\mathcal{A}$ be a propositional, input bounded ASM$^+$ transducer, and φ a CTL* formula over $\Sigma_{\mathcal{A}}$. Then $\mathcal{A} \not\models \varphi$ iff there exists a database instance D of size exponential in $\mathcal{A}$, such that $\mathcal{T}_{\mathcal{A},D} \models \neg\varphi$.

Proof. Let $\mathcal{A} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathcal{R} \rangle$ be a propositional, input-bounded ASM$^+$ transducer and φ a CTL* formula over $\Sigma_{\mathcal{A}}$. For each configuration ρ of $\mathcal{A}$, we denote by $\lambda(\rho)$ the set of states true in ρ. We also denote by λ the extension of this mapping to trees of configurations $\mathcal{T}_{\mathcal{A},D}$, where $\lambda(root) = \emptyset$. Obviously, if $\lambda(\mathcal{T}_{\mathcal{A},D_1}) = \lambda(\mathcal{T}_{\mathcal{A},D_2})$ then $\mathcal{T}_{\mathcal{A},D_1}$ and $\mathcal{T}_{\mathcal{A},D_2}$ satisfy the same CTL* formulas over $\Sigma_{\mathcal{A}}$. Now suppose that $\mathcal{A} \not\models \varphi$, so there is some D such that $\mathcal{T}_{\mathcal{A},D} \models \neg\varphi$. We show that there exists a database D_0 of size exponential in $\mathcal{A}$, such that $\lambda(\mathcal{T}_{\mathcal{A},D}) = \lambda(\mathcal{T}_{\mathcal{A},D_0})$, so $\mathcal{T}_{\mathcal{A},D_0} \models \neg\varphi$. This is done by showing that $\lambda(\mathcal{T}_{\mathcal{A},D}) = \lambda(\mathcal{T}_{\mathcal{A},D_0})$ iff D_0 satisfies a particular FO sentence ξ in the prefix class $\exists^*\forall^*$FO with a number of variables exponential in $\mathcal{A}$. Since ξ is satisfied by D, it is satisfiable. But this implies that ξ has a model D_0

whose domain has a number of elements equal to the number of existential variables of ξ, so exponential in $\mathcal{A}$ (see [8]). Thus, the size of D_0 is also exponential in $\mathcal{A}$ (for bounded database schema arity).

We next describe ξ. Note that, because states are propositional, the sets of propositions true in successors of a configuration ρ of a run of $\mathcal{A}$ on D depend only on D and $\lambda(\rho)$. Thus, $\lambda(\rho)$ uniquely determines the set $\{\lambda(\bar{\rho}) \mid \langle \rho, \bar{\rho} \rangle \in \mathcal{T}_{\mathcal{A},D}\}$ Consider a pair $\langle \Sigma, \bar{\Sigma} \rangle = \langle \lambda(\rho), \lambda(\bar{\rho}) \rangle \in \lambda(\mathcal{T}_{\mathcal{A},D})$. Let $I_1, \ldots, I_k$ be the input predicates in $\mathbf{I}$, and let $\varphi_1 \ldots, \varphi_k$ be the $\exists^*$FO formulas defining the input options for $I_1 \ldots, I_k$. We construct a quantifier-free FO sentence $\varphi_{\langle \Sigma, \bar{\Sigma} \rangle}(\bar{x}_1, \ldots, \bar{x}_k)$ on $\mathbf{D}$ such that $\lambda(\bar{\rho}) = \bar{\Sigma}$ whenever $\bar{\rho}$ is the next configuration from ρ resulting from the choice of inputs $\bar{x}_1, \ldots, \bar{x}_k$ from the options available for $I_1, \ldots, I_k$. For simplicity, we show the construction for the case when all user inputs are non-empty. The construction can be easily adapted to account for empty inputs.

Thus, $\langle \Sigma, \bar{\Sigma} \rangle \in \lambda(\mathcal{T}_{\mathcal{A},D_0})$ iff $D_0 \models \exists \bar{x}_1 \ldots \exists \bar{x}_k [\varphi_1(\bar{x}_1) \wedge \ldots \wedge \varphi_k(\bar{x}_k) \wedge \varphi_{\langle \Sigma, \bar{\Sigma} \rangle}(\bar{x}_1, \ldots, \bar{x}_k)]$. To ensure that only valid pairs $\langle \Sigma, \bar{\Sigma} \rangle$ occur in $\lambda(\mathcal{T}_{\mathcal{A},D_0})$, it must also be the case that $D_0 \models \forall \bar{x}_1 \ldots \forall \bar{x}_k [(\varphi_1(\bar{x}_1) \wedge \ldots \wedge \varphi_k(\bar{x}_k)) \rightarrow \bigvee_{\bar{\Sigma}} \varphi_{\langle \Sigma, \bar{\Sigma} \rangle}(\bar{x}_1, \ldots, \bar{x}_k)]$.

Then ξ is the conjunction of all such formulas for all pairs in $\lambda(\mathcal{T}_{\mathcal{A},D})$, yielding a formula in the prefix class $\exists^*\forall^*$FO. Since there can be exponentially many such pairs, ξ is exponential in $\mathcal{A}$.

In order to define the sentence $\varphi_{\langle \Sigma, \bar{\Sigma} \rangle}$ we need the following notation. For each FO sentence ψ let ψ_Σ be the sentence obtained by replacing in ψ every proposition $p \in \Sigma$ by *true* and $p \notin \Sigma$ by *false*. Further, for each input-bounded formula ψ let the quantifier-free version of ψ , denoted ψ^{qf}, be defined as follows. Intuitively, ψ^{qf} eliminates the quantifiers by taking advantage of the fact that each input I consists, after the user's choice, of at most a single tuple $\bar{x}_I$ ($\bar{x}_I$ is a sequence of m distinct variables, where m is the arity of I). The formula ψ^{qf} reformulates ψ using these tuples. Specifically, let ψ' be obtained by replacing each input-bounded quantification $\exists \bar{x}(\alpha \wedge \beta)$ and $\forall \bar{x}(\alpha \rightarrow \beta)$ by $\alpha \wedge \beta$.

Next, let ψ^{qf} be obtained by first bringing ψ' to DNF (disjunctions of conjunctions), then applying to each disjunct δ the following procedure yielding δ'. Let *eq*

(neq) be the (in)equalities occurring in δ. For each input relation I occurring in δ and each i, $1 \leq i \leq m$, let $\theta(I,i)$ be the set of terms occurring in the i-th position of I in a positive occurrence $I(\bar{z})$ in δ. Let $\equiv$ be the reflexive, transitive closure of the following relation on the terms of δ: $\{(x,y) \mid x = y \in \ eq\ \} \cup \{(x,y) \mid x,y \in \theta(I,i)$ for some I and $i\}$. If for some x,y it is the case that $x \equiv y$ and $x \neq y$ is in neq, then $\delta' = false$. Otherwise, define the following equivalence relation on the pairs (I,i) of input atoms I and positions i of I: $(I,i) \equiv (J,j)$ iff there exist terms x,y so that $x \equiv y$, $x \in \theta(I,i)$, and $y \in \theta(J,j)$. For each variable y in δ, let $\nu(y)$ be one arbitrarily chosen $(x_I)_i$ for which $y \in \theta(I,i)$. Let δ' be obtained as follows:

1. add to δ the conjunction of all equalities $(x_I)_i = (y_J)_j$ where $(I,i) \equiv (J,j)$, $c = c'$ where c, c' are constants and $c \equiv c'$, and $(x_I)_i = c$ for some arbitrarily chosen $c \in \theta(I,i)$, if such exists.

2. for each negative occurrence $\neg I(z_1, \ldots, z_m)$ of an input atom, add the conjunct consisting of the disjunction $\bigvee_{i=1}^{m}(\nu(z_i) \neq (x_I)_i)$;

3. delete all input atoms;

4. replace each variable y by $\nu(y)$ in the remaining atoms.

Finally, ψ^{qf} is the disjunction of all resulting δ'.

We can now define $\varphi_{\langle \Sigma, \bar{\Sigma} \rangle}$. This is constructed using the rules of $\mathcal{A}$. Consider a proposition p in $\mathbf{S}$. We associate to p and $\neg p$ formulas β_p and $\beta_{\neg p}$ defined using the rules for $(\neg)p$. If $p \leftarrow \gamma$ and $\neg p \leftarrow \delta$ are in $\mathcal{R}$ then β_p is τ_{Σ}^{qf}, where $\tau = (\gamma \wedge \neg\delta) \vee (p \wedge \neg\delta)$, and $\beta_{\neg p}$ is π_{Σ}^{qf} where $\pi = (\neg p \wedge \neg\gamma) \vee (\neg p \wedge \gamma \wedge \delta) \vee (\delta \wedge \neg\gamma)$. Finally, $\varphi_{\langle \Sigma, \bar{\Sigma} \rangle}$ is the quantifier-free formula $\bigwedge_{p \in \bar{\Sigma}} \beta_p \ \wedge \ \bigwedge_{p \in (\Sigma_V - \bar{\Sigma})} \beta_{\neg p}$. □

The next stage towards the proof of Theorem 4.14 is to reduce the verification problem for a fixed database to a model checking problem of a CTL$^{(*)}$ formula on a Kripke structure. We therefore show the following.

Lemma 4.16. For each ASM$^+$ transducer $\mathcal{A}$ over database schema $\mathbf{D}$, each database instance D over $\mathbf{D}$, and each CTL$^{(*)}$ formula φ over $\Sigma_{\mathcal{A}}$, one can construct, in time polynomial in D and exponential in $\mathcal{A}$, a Kripke structure $K_{\mathcal{A},D}$ over $\Sigma_{\mathcal{A}}$, of size exponential in $\Sigma_{\mathcal{A}}$, such that $\mathcal{T}_{\mathcal{A},D} \models \varphi$ iff $K_{\mathcal{A},D} \models \varphi$.

Proof. The Kripke structure $K_{\mathcal{A},D}$ has one node labeled for each set of propositions $\Sigma \subseteq \Sigma_{\mathcal{A}}$ labeling a node in $\lambda(\mathcal{T}_{\mathcal{A},D})$. There is an edge $\langle \Sigma, \bar{\Sigma} \rangle$ iff there is a node labeled Σ with a child labeled $\bar{\Sigma}$ in $\lambda(\mathcal{T}_{\mathcal{A},D})$. Clearly, $K_{\mathcal{A},D}$ can be obtained by expanding $\lambda(\mathcal{T}_{\mathcal{A},D})$ until no new labels are found. Each edge involves evaluating the formulas of $\mathcal{A}$ on Σ and D, which is polynomial in Σ and D and exponential in $\mathcal{A}$. The maximum number of edges is exponential in $\Sigma_{\mathcal{A}}$. $\square$

Lemmas 4.15 and 4.16 provide the proof of Theorem 4.14: to check that $\mathcal{A} \not\models \varphi$, first guess a database D of size exponential in $\mathcal{A}$, then construct from D and $\mathcal{A}$, in time exponential in $\mathcal{A}$, the Kripke structure $K_{\mathcal{A},D}$. Finally, checking that $K_{\mathcal{A},D} \models \neg\varphi$ is in polynomial time with respect to $K_{\mathcal{A},D}$ and $\neg\varphi$ if φ is in CTL, and in polynomial space if φ is in CTL*. Overall, checking $\mathcal{A} \models \varphi$ is in CO-NEXPTIME if φ is in CTL, and in EXPSPACE if φ is in CTL*. $\square$

A special case of interest involves ASM$^+$ transducers 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 transducer *fully propositional.* We can show the following:

Theorem 4.17. Given a fully propositional ASM$^+$ transducer $\mathcal{A}$ and a CTL* formula φ over $\Sigma_{\mathcal{A}}$, it is decidable in PSPACE whether $\mathcal{A} \models \varphi$.

Proof. In the case of a fully propositional ASM$^+$ transducer $\mathcal{A}$, the Kripke structure $K_{\mathcal{A},D}$ is independent of D (let us denote it by $K_{\mathcal{A}}$). However, $K_{\mathcal{A}}$ is exponential with respect to $\mathcal{A}$ so cannot be constructed in PSPACE. We therefore need a more subtle approach, that circumvents the explicit construction of $K_{\mathcal{A}}$. 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 [37] can be adapted to fully propositional transducers. The algorithm uses a special kind of tree automaton, called *hesitant alternating tree automaton* (HAA) (see [37] for the definition). As shown in [37], 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 K, one can construct a product HAA $K \times A_\varphi$ that is nonempty iff $K \models \varphi$. The nonemptiness test can be rendered efficient using the crucial observation that nonemptiness of $K \times A_\varphi$ 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 $K \times A_\varphi$ need not be constructed explicitly. Instead, its transitions can be generated on-the-fly from K and φ, as needed in the nonemptiness test for the 1-letter word HAA corresponding to $K \times A_\varphi$. This yields a model checking algorithm of space complexity polynomial in φ and polylogarithmic in K. We refer to [37] for details.

In our case, K is $K_\mathcal{A}$, and the input consists of φ and $\mathcal{A}$ instead of φ and $K_\mathcal{A}$. The previous approach can be adapted by pushing further the on-the-fly generation of $K_\mathcal{A} \times A_\varphi$ by also generating on-the-fly the relevant edges of $K_\mathcal{A}$ from $\mathcal{A}$ when needed. This yields a polynomial space algorithm for checking whether $\mathcal{A} \models \varphi$, similar to the algorithm with the same complexity obtained in [37] for model checking of concurrent programs. $\square$

4.2 Verification of Web Applications

We finally present our verification results for Web applications. Most of the results are shown by reducing the verification problem for Web applications to corresponding verification problems for ASM$^+$ transducers. We begin with linear-time properties.

4.2.1 Linear-time properties of Web applications

As for ASM$^+$ transducers, the decidability results for verification of linear-time properties of Web applications require the input-boundedness restriction. This extends naturally from ASM$^+$ transducers to Web applications.

Definition 4.18. A Web application is *input-bounded* if all formulas used in state, action, and target rules are input bounded, and formulas used in input option rules are $\exists^*$FO formulas in which all state atoms are ground.

Example 4.19 All rules on pages HP,LSP in Example 3.2 are input-bounded. Property (3.1) in Example 3.5 is trivially input-bounded, as it contains no quantifiers. Property (3.2) in Example 3.6, 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 (3.2) to the input-bounded sentence

$$\forall pid, price\ [\xi'(pid, price)\ \mathbf{B} \neg(\overline{\mathsf{conf}}(\mathsf{name}, price) \wedge \overline{\mathsf{ship}}(\mathsf{name}, pid)\ ] \quad (4.5)$$

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

$$\begin{aligned} \text{PP} \wedge \mathsf{pay}(price) \wedge \mathsf{button}(\text{``}\textit{authorize payment}\text{''}) \\ \wedge \text{pick}(pid, price) \wedge \underline{\mathsf{prod\_prices}}(pid, price) \end{aligned} \quad (4.6)$$

□

We show the following result on the verification of linear-time properties. Recall from Section 3.3 that a Web application is error-free if no run ever leads to the special error page.

Theorem 4.20. The following are decidable:

(i) given an input-bounded Web application $\mathcal{W}$, whether it is error free;

(ii) given an error-free Web application $\mathcal{W}$ 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.

The lower bound follows immediately from the PSPACE lower bound for ASM$^+$ transducers. The upper bound is more involved, and requires a reduction to the verification problem for ASM$^+$ transducers. To begin, we note that part (i) of Theorem 4.20 can be reduced to part (ii).

Lemma 4.21. For each Web application $\mathcal{W}$ with input-bounded rules there exists an error-free Web application $\mathcal{W}'$ 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.

Proof. Let $\mathcal{W} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, W_0, W_\epsilon \rangle$ be a Web application with input-bounded rules. Intuitively, we wish to construct a Web application $\mathcal{W}'$ with a new Web page schema W'_ϵ that is reached according to the rules of the application (and without generating an error), exactly when the error page W_ϵ would be reached in the original Web application. Then it is enough to verify that W'_ϵ is never reached in any run of $\mathcal{W}'$. To this end, we define $\mathcal{W}' = \langle \mathbf{D}, \mathbf{S}', \mathbf{I}, \mathbf{A}, \mathbf{W}', W'_0, W_\epsilon \rangle$ as follows. For each input constant c of $\mathcal{W}$, let p_c be a new propositional symbol, and $\mathbf{S}' = \mathbf{S} \cup \{p_c \mid c \text{ is an input constant of } \mathcal{W}\}$. $\mathbf{W}'$ contains a new Web page schema W'_ϵ defined identically to W_ϵ, and for each Web page schema $W = \langle \mathbf{I}_W, \mathbf{A}_W, \mathbf{T}_W, \mathcal{R}_W \rangle$ of $\mathbf{W}$ different from W_0 and W_ϵ, a Web page schema $W' = \langle \mathbf{I}_W, \mathbf{A}_W, \mathbf{T}'_W, \mathcal{R}'_W \rangle$, where $\mathbf{T}'_W = \mathbf{T}_W \cup \{W'_\epsilon\}$. $\mathcal{R}'_W$ consists of the following rules. The state, input, and action rules of $\mathcal{R}_W$ remain unchanged, except for the addition of one state rule $p_c \leftarrow \textit{true}$ for each input constant $c \in \mathbf{I}_W$. Before defining the target rules, let ψ_W be $\psi_1 \vee \psi_2 \vee \psi_3$, where:

- ψ_1 is the disjunction of all formulas $\varphi_{V,W} \wedge \varphi_{V',W}$ where $V \neq V'$ and $V \leftarrow \varphi_{V,W}$, $V' \leftarrow \varphi_{V',W}$ are target rules in $\mathcal{R}_W$,
- ψ_2 is the disjunction of all formulas $\varphi_{V,W} \wedge \neg p_c$ where $V \leftarrow \varphi_{V,W}$ is a target rule in $\mathcal{R}_W$ and c is an input constant occurring in some input rule in V but not in $\mathbf{I}_W$, or occurring in some other rule of V, but not in $\mathbf{I}_W \cup \mathbf{I}_V$, and
- ψ_3 is the disjunction of the formulas $\varphi_{V,W} \wedge p_c$ where $V \leftarrow \varphi_{V,W}$ is a target rule in $\mathcal{R}_W$ and c occurs in $\mathbf{I}_V - \mathbf{I}_W$, and $\varphi_{V,W}$ if $c \in \mathbf{I}_W \cap \mathbf{I}_V$.

Intuitively, ψ_W states that the original target rules of W are ambiguous (stated by ψ_1) or the next Web page uses some input constant not yet provided (formula ψ_2), or the next Web page requires as input some constant already provided (stated by ψ_3). The target rules of W' make use of ψ_W:

- each target rule $V \leftarrow \varphi_{V,W}$, where $V \in \mathbf{T}_W$, is replaced by $V \leftarrow \varphi_{V,W} \wedge \neg\psi_W$,
- $W'_\epsilon \leftarrow \psi_W$ is a new target rule.

Finally, W'_0 is a special case. It is defined as above if input rules of W_0 contain no input constants, and the other formulas contain only input constants in $\mathbf{I}_{W_0}$; otherwise, it is defined as $\langle \emptyset, \emptyset, \{W'_\epsilon\}, \{W'_\epsilon \leftarrow \mathit{true}\}\rangle$.

It is easily verified that $\mathcal{W}'$ is error free and $\mathcal{W}'$ is input bounded if $\mathcal{W}$ is input bounded. Also, $\mathcal{W}$ is error free iff the page W'_ϵ is never reached in any run of $\mathcal{W}'$, i.e. $\mathcal{W}'$ satisfies the input-bounded LTL-FO sentence $\mathbf{G}\ \neg W'_\epsilon$. $\square$

The following shows that checking that a Web application is error-free is already PSPACE-hard.

Lemma 4.22. Checking whether an input-bounded Web application is error free is PSPACE-hard.

Proof. The proof is by reduction from Quantified Boolean Formula (QBF), known to be PSPACE-complete [27]. Let φ be a quantified Boolean formula (we can assume φ uses just $\vee, \neg, \exists$). Consider the Web application $\mathcal{W}_\varphi = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, W_0, W_\epsilon \rangle$ where:

- $\mathbf{D} = \{R : 1, 0, 1\}$, $\mathbf{S} = \emptyset$, $\mathbf{I} = \{I_0 : 1, I_1 : 1\}$, $\mathbf{A} = \emptyset$, $\mathbf{W} = \{W_0, W_1, W_2\}$;
- $W_0 = \langle \{I_0, I_1\}, \emptyset, \{W_1, W_2\}, \mathcal{R}_{W_0} \rangle$ where $\mathcal{R}_{W_0}$ consists of the input rules

$$Options_{I_i}(x) \;\leftarrow\; R(x),$$

  for $i \in \{0, 1\}$ and the target rules

$$W_i \leftarrow I_0(0) \wedge I_1(1) \wedge 0 \neq 1 \wedge \varphi', i \in \{1, 2\},$$

  where φ' is defined from φ as follows:

  - each propositional variable x is replaced by $(x = 1)$;
  - disjunction and negation remain unchanged;
  - $\exists x \psi$ becomes $\exists x((I_0(x) \vee I_1(x)) \wedge \psi)$.

- $\{W_1, W_2\}$ are arbitrary.

Clearly, $\mathcal{W}_\varphi$ is input-bounded and of size polynomial in φ, and it is error free iff there is no run for which $I_0 = \{0\}$, $I_1 = \{1\}$, and φ' is true. Obviously, there exists a run for which $I_0 = \{0\}$ and $I_1 = \{1\}$. But then φ' has the same value as φ. Therefore, $\mathcal{W}$ is error-free iff φ is false. □

We next reduce the verification of error-free Web applications to verification of ASM$^+$ transducers.

Lemma 4.23. Let $\mathcal{W} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, W_0, W_\epsilon \rangle$ be an error-free, input-bounded Web application and φ an LTL-FO or CTL$^{(*)}$-FO sentence over the schema of $\mathcal{W}$. There exists an ASM$^+$ transducer $\mathcal{A}$, of size linear in $\mathcal{W}$, such that $\mathcal{W} \models \varphi$ iff $\mathcal{A} \models \varphi$.

Proof. In brief, the reduction has to overcome two obstacles: (i) simulating the multiple Web schemas of $\mathcal{W}$, and (ii) eliminating the constants from the input schema of $\mathcal{W}$. It is easy to deal with (i): we just simulate the behavior of different Web pages and transitions using new propositional state variables corresponding to the Web pages. Overcoming (ii) makes essential use of the assumption that $\mathcal{W}$ is error free. Indeed, this guarantees that the value of each input constant is only provided once, and that no formula makes use of such constants before they are provided. This allows to assume that the input constants are provided prior to the run, as part of the database.

More precisely, let $\mathcal{A} = \langle \mathbf{D}', \mathbf{S}', \mathbf{I}', \mathbf{A}', \mathcal{R} \rangle$ where:

- $\mathbf{D}' = \mathbf{D} \cup \mathit{const}(\mathbf{I})$, where $\mathit{const}(\mathbf{I})$ denotes the set of input constant symbols in $\mathbf{I}$
- $\mathbf{S}' = \mathbf{S} \cup \mathbf{W}$, where each $W \in \mathbf{W}$ is taken to be a propositional symbol;
- $\mathbf{I}' = \mathbf{I} - \mathit{const}(\mathbf{I})$;
- $\mathbf{A}' = \mathbf{A}$;

The set of rules $\mathcal{R}$ of $\mathcal{A}$ is defined as follows. For each relational input I of $\mathbf{I}$ we add to $\mathcal{R}$ the input rule $\mathit{Options}_I(\bar{x}) \leftarrow \xi$, where ξ is the disjunction of all formulas $\varphi_{I,W}(\bar{x}) \wedge W$ for which $\mathit{Options}_I(\bar{x}) \leftarrow \varphi_{I,W}(\bar{x})$ is an input rule of the page W in $\mathcal{W}$. We define the state rules next. For each state rule $(\neg)S(\bar{x}) \leftarrow \varphi^{\epsilon}_{S,W}(\bar{x})$ of $\mathbf{W}$, we add a state rule $(\neg)S(\bar{x}) \leftarrow \varphi^{\epsilon}_{S,W}(\bar{x}) \wedge W$ to $\mathcal{R}$. In addition, for each target rule $V \leftarrow \varphi_{V,W}$ of $\mathbf{W}$ we add to $\mathcal{R}$ the state rules $V \leftarrow \varphi_{V,W} \wedge W$ and, if $V \neq W$, $\neg W \leftarrow \varphi_{V,W} \wedge W$. The action rules of $\mathcal{R}$ consist of all rules $A(\bar{x}) \leftarrow \varphi(\bar{x}) \wedge W$ for which $A(\bar{x}) \leftarrow \varphi(\bar{x})$ is an action rule of Web page schema W in $\mathbf{W}$. $\Box$

Theorem 4.20 (ii) and the PSPACE upper bound (EXPSPACE with no fixed bound on arities) now follow from Lemma 4.23 and Theorem 4.8.

The undecidability results developed in Section 4.1.2 carry over to verification of Web applications due to the reduction provided by Lemma 4.23. The following is a corollary of Lemma 4.23 and Theorems 4.9, 4.10, 4.11, and 4.12.

Corollary 4.24.

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

2. It is undecidable, given an error-free, input-bounded Web application $\mathcal{W}$ with state projections and input-bounded LTL-FO sentence φ, whether $\mathcal{W} \models \varphi$.

3. It is undecidable, given an error-free, input-bounded Web application $\mathcal{W}$ with lossless input and an input-bounded LTL-FO sentence φ, whether $\mathcal{W} \models \varphi$.

4. It is undecidable, given an error-free, input-bounded Web application $\mathcal{W}$ with functional dependencies $\mathcal{F}$ on its database schema, and an input-bounded LTL-FO sentence φ, whether $\mathcal{W} \models_{\mathcal{F}} \varphi$.

4.2.2 Branching-time properties of Web applications

Lemma 4.23 and Theorem 4.13 imply the following undecidability result:

Corollary 4.25. It is undecidable, given an error-free, input-bounded Web application $\mathcal{W}$ and input-bounded CTL-FO sentence φ, whether $\mathcal{W} \models \varphi$.

We therefore consider next several restrictions leading to decidability of the verification problem for CTL$^{(*)}$-FO sentences. Some of the results mirror directly those obtained for ASM$^+$ transducers in Section 4.1.3, while others require some development specific to the Web application formalism.

Propositional input-bounded Web applications The first restriction we consider for Web applications is an extension of propositional input-bounded ASM$^+$ transducers. The restriction limits input-bounded Web applications by requiring all states and actions to be propositional. Furthermore, no rules can use $\mathbf{Prev_I}$ atoms. We also call such Web applications *propositional*. As for ASM$^+$ transducers, in a propositional Web application, inputs can still be parameterized in the

Web application specification. The CTL* formulas we consider are propositional and use input, action, state, and Web page symbols, viewed as propositions (recall that the CTL* formulas used for propositional ASM$^+$ transducers used only the states). Satisfaction of such a CTL* formula by a Web application 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 J is true iff $J \in \mathbf{I}_V$, and an action symbol a is true iff $a \in A$.

Example 4.26 CTL$^{(*)}$-FO is particularly useful for specifying navigational properties of Web applications. Note that these applications do not necessarily have to be propositional; we could abstract their predicates to propositional symbols, thus concentrating only on reachability properties. This is in the spirit of program verification, where program variables are first abstracted to booleans [16, 29], in order to check CTL* properties such as liveness. 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:[5]

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

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

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

$\square$

[5]The most important property in electronic commerce ⌣̈

For a given Web application $\mathcal{W} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, W_0, W_\epsilon \rangle$, 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 $\Sigma_\mathcal{W}$.

Theorem 4.27. Given a propositional, input-bounded, error-free Web application $\mathcal{W}$ 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*.

Theorem 4.27 is a consequence of Theorem 4.14 on ASM$^+$ transducers together with the following.

Lemma 4.28. For each propositional, input-bounded, error-free Web application $\mathcal{W}$ and CTL* formula φ over $\Sigma_\mathcal{W}$, one can construcrt in linear time a propositional, input-bounded ASM$^+$ transducer $\mathcal{A}$ such that $\Sigma_\mathcal{A} \supseteq \Sigma_\mathcal{W}$ and $\mathcal{W} \models \varphi$ iff $\mathcal{A} \models \varphi$.

Proof. The proof is similar to that of Lemma 4.23. In order for the states of $\mathcal{A}$ to contain all propositions in $\Sigma_\mathcal{W}$, one has to introduce, in addition to the states for Web pages introduced in the proof of Lemma 4.23, new states for all actions and inputs, that are true precisely when the corresponding propositional symbol in $\Sigma_\mathcal{W}$ evaluates to true in the semantics of CTL* formulas over $\Sigma_\mathcal{W}$ described above. This is straightforward and details are omitted. $\square$

The complexity of the decision problem of Theorem 4.14 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 4.29. 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 application $\mathcal{W}$ 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 4.14. Since $\mathbf{S}$ is fixed and φ refers only to $\mathbf{W}$, it is enough to retain, in labels of $\lambda(\mathcal{T}_{\mathcal{W},D})$ only the states and Web page names. Since $\mathcal{W}$ is error free, there is exactly one Web page name per label. It follows that the number of pairs $\langle \Sigma, \bar{\Sigma} \rangle$ occurring in $\lambda(\mathcal{T}_{\mathcal{W},D})$ is quadratic in $\mathbf{W}$, so the formula ξ has polynomially many variables, and the size of the database D_0 is polynomial in $\mathcal{W}$. The Kripke structure $K_{\mathcal{W},D_0}$ can now be constructed in PSPACE with respect to $\mathcal{W}$, and checking φ can be done in PSPACE with respect to $K_{\mathcal{W},D_0}$ and φ. Altogether, checking that $\mathcal{W} \models \varphi$ is done in PSPACE with respect to $\mathcal{W}$ and φ. $\square$

Another special case of interest, as for ASM$^+$ transducers, involves Web applications 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. Such Web application are called *fully propositional.* We can show the following, which is a direct consequence of Lemma 4.23 and Theorem 4.17.

Theorem 4.30. Given a fully propositional, error-free Web application $\mathcal{W}$ and a CTL* formula φ over $\Sigma_{\mathcal{W}}$, it is decidable in PSPACE whether $\mathcal{W} \models \varphi$.

One may wonder if the restrictions of Theorem 4.27 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 applications with input-driven search The restrictions considered so far require states of a Web application 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 applications 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 4.31. A *Web application with input-driven search* is an input-bounded Web application $\mathcal{W} = \langle \mathbf{D}, \mathbf{S}, \mathbf{I}, \mathbf{A}, \mathbf{W}, W_0, W_\epsilon \rangle$ 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 symbol i_0 and a designated binary relation R_I
- the state rule for *not-start* is *not-start* $\leftarrow \neg$*not-start*
- the input option rule for I is in all Web pages of the form

$$\begin{aligned} &\mathit{Options}_I(y) \leftarrow (\neg \mathit{not\text{-}start} \ \wedge \ y = i_0) \\ &\vee(\mathit{not\text{-}start} \ \wedge \ \exists x(\mathit{prev}_I(x) \ \wedge \ R_I(x,y)) \ \wedge \ \varphi(y)) \end{aligned}$$

  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 4.32 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 4.1. The user can search the hierarchy

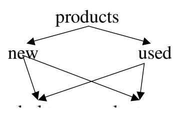

Figure 4.1 Fragment of R_I for Example 4.32

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 application with input-driven search, in which the binary database relation R_I is a graph which contains as a subgraph the one in Figure 4.1, and in which the unary database relations such as $\underline{\mathsf{newDesktop}}$,$\underline{\mathsf{usedDesktop}}$,$\underline{\mathsf{usedLaptop}}$ contain the in-stock products. Here is the input rule corresponding to the desktop search page:

$$\begin{aligned} &Options_I(y) \leftarrow (\neg not\text{-}start \wedge y = i_0) \vee \\ &\quad not\text{-}start \wedge \exists x(prev_I(x) \wedge R_I(x,y)) \wedge \\ &\quad (\text{new} \wedge \underline{\mathsf{newDesktop}}(y) \vee \neg\text{new} \wedge \underline{\mathsf{usedDesktop}}(y)) \end{aligned} \tag{4.7}$$

□

We can show the following.

Theorem 4.33. Given a Web application 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. As mentioned in Section 2.1.3, this is known to be EXPTIME-complete for CTL, and 2-EXPTIME complete for CTL*. We consider Kripke structures over the alphabet $\Sigma_{\mathcal{W}} \cup \mathbf{D}$. Intuitively, each node of the

Kripke structure represents a configuration, and its label represents the relevant information about the configuration: the set of propositions in $\Sigma_{\mathcal{W}}$ that hold, and the type of the current input with respect to the database, i.e. the set of relations Q in $\mathbf{D} - \{R_I\}$ for which $y^k \in Q$, where k is the arity of Q and y the current input. Note that the types of different inputs are independent of each other because inputs are unary, so every Kripke structure can be viewed as representing an input choice relation R_I together with type assignments for the elements of R_I. In addition, in order for a Kripke structure to represent an actual run of of $\mathcal{W}$, the assignments of literals of $\Sigma_{\mathcal{W}}$ to nodes has to be consistent with the rules of $\mathcal{W}$. However, this can be easily expressed by a CTL formula ρ computable in polynomial time from $\mathcal{W}$. It follows that $\mathcal{W} \models \varphi$ iff $\rho \wedge \neg\varphi$ is unsatisfiable. The latter is a CTL formula, if φ is in CTL, and a CTL* formula if φ is in CTL*. $\square$

Chapter 5

WAVE: A Verifier for Interactive, Data-driven Web Applications

In the last chapter, we identified a practically appealing and fairly tight class of Web applications 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. However, the PSPACE upper bound provides no indication of whether verification is practically feasible.

In this chapter, we present WAVE, a verifier for input-bounded LTL-FO properties and interactive, data-driven Web applications specified using high-level modeling tools, such as the specification language we provide in Section 3.1 and WebML. WAVE is complete for a broad class of applications and temporal properties. By coupling the pseudorun technique, described in Section 4.1.1, with various database heuristics, our experiments on four representative data-driven applications and a battery of common properties yielded surprisingly good verification times, on the order of seconds. It suggests that interactive applications controlled by database queries may be unusually well suited to automatic verification. The experimental results also show that the coupling of model checking with database optimization techniques used in the implementation of WAVE can be extremely effective, which is significant both to the database area and to automatic verification in general.

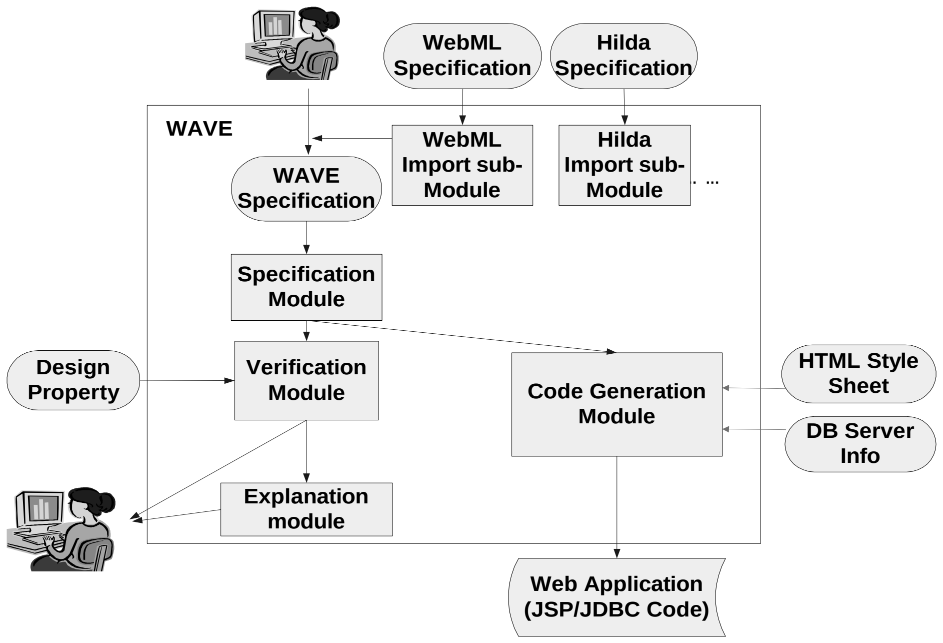

Figure 5.1 System architecture

Section 5.1 presents the architecture of our system with detailed descriptions for individual modules. Our verification algorithm is presented in Section 5.2. In particular, Section 5.2.2 addresses optimizations exploiting the structure of the database and specification rules. Section 5.3 details how our implementation exploits the capabilities of a main-memory database management system while Section 5.4 reports on the experimental evaluations of WAVE.

5.1 System Architecture

The architecture of WAVE is illustrated in Figure 5.1, which covers many aspects ranging from easy-to-understand specification of Web applications to informative explanation of verification results, which requires no background on formal verification. We next describe each module in some detail.

Specification module The specification module reads a text file conforming to the grammar of the specification language and processes it. The text file is the specification of a Web application consisting of two sections as specified in Definition 3.1[1]. The first section is known as the global declaration, which contains signatures of all relations, categorized into database, state, input and action relations, as well as identifiers for all Web pages. The second section contains the schema for each Web page, an example of which is shown in Example 3.2. Once the file conforms to the grammar, the parser generates an internal representation of the specification, which is consumed by the verification module and the code generation module.

It is also possible to create the specification file using a graphical interface. Indeed, we are currently collaborating with the WebML group to implement a *WebML import sub-module* which reads in WebML specifications (in XML) and translates them into our specifications. Consequently, the completion of this sub-module will enable WAVE to be applicable to any Web application specifiable by WebML, which is highly desirable. The dots to the right of the WebML import sub-module indicate that similar sub-modules can be added for other specification languages as well.

Verification module The verification module is essentially the core of WAVE. It takes the specification and a property as input, and computes the information needed for simulating runs of the Web application. A true or false result is produced at the end of the simulation along with useful verification information.

For details of the algorithm and implementation of the verification modual, please refer to Section 5.2 and Section 5.3.

Explanation module The explanation module helps the user understand the output from the verification module better by reproducing counter-examples and/or Web application configurations in a more intuitive and informative way.

The counter example will show the precise sequence of user inputs, the con-

[1]The complete grammar is available at http://sashimi.ucsd.edu:8080/wave/README.txt

tents of the database, and the sequence of configurations through which the Web application evolves, as well as the exact point where the property is violated. The developer can fast-forward and fast-backward through the run, inspect the contents of the state and the database at will.

Code generation module The code generation module is orthogonal to the verification module and can be regarded as an interpreter for the specification language. It reads in the specification file and automatically generates JSP pages that correspond to the specified Web application. Assuming the availability of a configured Tomcat server, the set of generated JSP pages are ready to be viewed in a browser. The code implementing the connection to the underlying database is generated in JDBC. We are currently enhancing this module to allow the Web developer to supply HTML style sheet information to specify the layouts on the generated pages.

Now that we know how WAVE works, let's drill down into the essential module of WAVE verifier—the verification module.

5.2 Web application Verification Algorithm and Optimization

Given the pseudorun technique presented in Section 4.1.1, we get a PSPACE algorithm for the verification problem. 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 that can affect the page rules and property. As we have already defined in Definition 4.5, the resulting sequence of partially specified configurations (pseudoconfigurations) is called *pseudorun*. The key advantage of pseudoruns is that their partially specified configurations have polynomial size in the application specification and property, thus yielding a PSPACE verification algorithm.

However, the algorithm we demonstrated in Section 4.1.1 is for ASM$^+$ transducer which is isomorphic to a Web application with a single Web page schema and no input constants. Section 5.2.1 takes a crucial step in extending and refining the algorithm towards a practical algorithm for the general Web application, which might contains multiple pages and some input constants.

Although the algorithm is quite reasonable as static analysis goes, it is still insufficient in practice. The pseudorun-based search achieves practical relevance only with the aid of two heuristics (presented in Section 5.2.2) 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.

The roadmap of our verification approach is the following. Given Web application $\mathcal{W}$ and property $\varphi_0 \in$ LTL-FO, we guarantee 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 well-known domain tools such as `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 it returns `yes`, otherwise it returns `no` and reports the counterexample pseudorun. Pseudoruns are pruned according to the heuristics exploiting the dataflow analysis of the specification and property.

The first two steps of the approach are demonstrated in Section 4.1.1 while step 3 is covered in Sections 5.2.1 and 5.2.2.

5.2.1 Searching for Pseudoruns

Section 4.1.1 presents a PSPACE verification algorithm which circumvents the explicit enumeration of representative databases. However, recall that the original algorithm is for the ASM$^+$ transducer which is isomorphic to a Web application with a single Web page schema and no input constants. In this section we extend and refine that algorithm to handle the general case Web applications, by actually generating pseudo-runs that involve multiple web pages and some input constants, rather than reducing it to an ASM$^+$ transducer, since this is more natural and less complex.

Remember that the original 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. In Definition 4.5, we call the partially configurations *pseudoconfiguration* and the resulting sequences of pseudoconfigurations *pseudoruns*. 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 to 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, and 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, as shown in Section 4.1.1. 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 in 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 in the ASM$^+$ transducer, we can pick the input from a fixed domain $\mathbf{C}_k$(please refer to Definition 4.5 for details) with witnesses in $\mathbf{C}_{km}$, since the ASM$^+$ transducer is isomorphic to a Web application with only one page and no input constants. It turns out (see Lemma 4.6) that we do not lose completeness by restricting our picks this way.

We now extend it into the general case(the Web applications with multiple pages and input constants): 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 $V' \neq V$. In other word, we replace $\mathbf{C}_{km}$-$\mathbf{C}$ in the SWA one page case with C_V, Similiar to the $\mathbf{C}_{km}$-$\mathbf{C}$, 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.

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 $\varphi_1(\mathbf{C}_\exists)$. Similarly, when the Web application is a SWA which contains only one web page V, we consider only database tuples over $\mathbf{C} \cup \mathbf{C}_V$, which is consistent with Definition 4.5.

Furthermore, 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 tuples 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 sub instance 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 in 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 is 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 $C_s = \langle D_s, V_s, I_s, P_s, S_s, A_s \rangle$
output: set of successor pseudoconfigurations of $C_s$
$result := \emptyset$
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 $\mathbf{C}$
$P_t := I_s$
// pick successor's partial database $D_t$ :
let $DBcore$ be the core of $D_s$
for each $DBext \in \mathbf{ext}(V_t)$
    let $D_t := DBcore \cup DBext$
    // this is the generalization of what we defined in Definition 4.5:
    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 $\mathbf{C}$
        $result := result \cup \{\langle D_t, V_t, P_t, I_t, S_t, A_t \rangle\}$
return $result$
```

Lemma 4.6 shows that it suffices to restrict the search for a run satisfying an input-bounded property to pseudoruns only.

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.

Together with the *ndfs* algorithm in Section 2.2.1, we now generalize the algorithm presented in Section 4.1.1 and derive the Algorithm *ndfs-pseudo* below, which 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 $\varphi$'s FO components:
for each choice of $\mathbf{C}_\exists$
    instantiate the free variables of $\varphi$'s FO components with $\mathbf{C}_\exists$
    $\mathbf{C} := \mathbf{C}_\mathcal{W} \cup \mathbf{C}_\exists$
    // construct the start pseudoconfigurations:
    let $V_0$ be the home page of $\mathcal{W}$
    $P_0 := \emptyset$; $S_0 := \emptyset$
    for each $DBcore \in \mathbf{cores}(\mathbf{C})$
        for each $DBext \in \mathbf{ext}(V_0)$
            $D_0 := DBcore \cup DBext$
            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 $\mathbf{C}$
        $C_0 := \langle D_0, V_0, I_0, P_0, S_0, A_0 \rangle$
        let $s_0$ be the start state of $A_{\varphi^{aux}}$
        // search for pseudorun determining lollipop path:
        stick($s_0, C_0$)

procedure stick($s, C_s$)
record $\langle (s, C_s), 0 \rangle$ as visited
evaluate $\varphi$'s instantiated FO components on $C_s$
            to get truth values of auxiliary propositions $P^{aux}$
for each $(s, \delta, t) \in \mathcal{T}_{\varphi^{aux}}$ such that $P^{aux}$ satisfies $\delta$
    for each $C_t \in succ_P(C_s)$
        if $\langle (t, C_t), 0 \rangle$ 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 $\langle (s, C_s), 1 \rangle$ as visited
evaluate $\varphi$'s instantiated FO components on $C_s$
            to get truth values of auxiliary propositions $P^{aux}$
for each $(s, \delta, t) \in \mathcal{T}_{\varphi^{aux}}$ such that $P^{aux}$ satisfies $\delta$
    for each $C_t \in succ_P(C_s)$
        if $\langle (t, C_t), 1 \rangle$ not yet visited then candy($t, C_t$)
        else if $(t, C_t) =$base then report pseudorun
```

Theorem 5.1. 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 5.2 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 3.2 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 5.2.2.

5.2.2 Optimizations

As illustrated by Example 5.2, 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 are checked by page rules and by the property. Ground tuples consist of exclusively constants and are detected by comparing all their attributes with constants. For instance, the home page schema HP of the online computer shopping example[1] authenticates users by testing for the presence of ground tuple $\underline{\text{user}}$(name,password) in the database, where name and password are input constants provided by the user at login. However, the $\underline{\text{user}}$ attribute is 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, thus 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 5.3 Assume we want to verify Property (4.1) on the computer shopping application of Example 3.2. 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 $\underline{\textsf{criteria}}$, 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. Table $\underline{\textsf{products}}$ 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 $\underline{\textsf{products}}$, yielding two cores: the empty core and the single-tuple core. Further analysis yields only four possible $\underline{\textsf{user}}$ cores, which together with the two $\underline{\textsf{products}}$ 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 rules, 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 to the size of the property and specification.

Example 5.4 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 5.2 also shows that, even if we reduce the set of cores to a manageable size, we still 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 5.5 We consider the extensions at page LSP from Example 3.2. 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 5.2 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 5.6 below, this comes at no sacrifice of completeness.

Theorem 5.6. 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.

5.3 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 below, 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[36] which allows updates and membership tests in time linear to the size of the bitmap.

The bitmap scheme is extended to configurations as follows. For the current page schema, we use a bitmap with as many entries as page schema names in the specification. For states, previous inputs and actions, we use our dataflow analysis to associate to each of their attributes the set of constants to which they may be compared. We then employ the same bitmap encoding scheme as used for cores and extensions. All bitmaps are concatenated to yield a bitmap representation of pseudorun configurations.

Recall that the trie data structure represents keys as sequences of symbols, not as a whole, like conventional structures do. Assume we have a finite alphabet Σ. We put |Σ| = m. Trie is an m-ary tree, its nodes are m-ary vectors indexed by the Σ alphabet. A node in depth l represents a set of keys starting with a prefix of length l. The node represents an m-way branch driven by (l + 1)-st character of the searched word.

When searching for a key in the trie, it starts at the root node branching by the first character. In a general case we progress as follows. We take the next symbol of the word, let it be k. Then the field of the current node indexed by the character k keeps the pointer to the subtrie, that corresponds to the search in the unread part of the key. Note that if the key is not in the trie, we find at least its longest prefix. The time complexity is linear wrt. the length of the key.

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 $R(A_1, \dots, A_k)$ and let n_i be the number of constants assigned by our dataflow analysis to attribute A_i. Given an R-tuple $t = (c_1, \dots, c_k)$, 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 $j = r_k + n_k \times (r_{k-1} + n_{k-1} \times (\dots n_2 \times r_1))$. 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 $(r_1, \dots, r_k)$ 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 [3]. The presence of universal quantifiers in the FO queries can lead to deeply nested SQL queries, which are notoriously hard to be optimized and inefficiently to be 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 when 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 to 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 the 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.

5.4 Experiments

We considered four experimental setups, **E1** through **E4** covering common types of Web applications. Their specifications were provided by the authors of [21] as proof of the expressivity of input-bounded specifications and are all available from [1]. We use them to evaluate our verifier.

E1: The online computer shopping application(our running example) offers functionality similar to that of the Dell Web site. It is demonstrated in [1]. 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 sport 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 [11]. 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 [34]). They are included in Table 5.1.

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

Table 5.1 Property types and its syntactic shapes

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	$\mathbf{G}p$

holds, it also holds at the successor step ($\mathbf{X}$). In [34], 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. Table 5.2 summarizes the results of verifying 17 properties $P_1, \ldots, P_{17}$. 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

Table 5.2 Verification result for 17 properties.

Type	Property name	Time [seconds]	Max run[2] length	Max. trie size
T1	P5 (true)	4	11	268
	P7 (false)	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

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 (4.1) from Example 4.3, 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:
$\forall oid, uname, pid, price, status(\underline{\textsf{orders-db}}(oid, uname, pid, price, \text{"}ordered\text{"})\mathbf{B}($ CCP$\wedge$

[2]Max length of run (if true) or of counterexample (if false).

userorderpick$(oid, pid, price, status)))$. The order status is read from the orders-db database table. The order being cancelable 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: $\forall pid, price(\mathbf{F}\text{cart}(pid, price) \rightarrow \mathbf{F}\mathsf{pick}(pid, price))$. 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) must have been reached and the user must have clicked the "circuits" button, or the grand prix detail page (GDP) must have been reached and the user must have 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, and omitted due to space limitations.

Failure of Classical Tools

SPIN. Recall that the first cut algorithm sketched in Section 5.2 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 [34]. To answer this question, 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 leads 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.

5.5 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 both soundness and completeness is lost for non-input-bounded applications or properties, we can still obtain soundness but not completeness for the input-bounded specifications with arithmetic other predicates with restricted semantics abstracted out as black-box relations. WAVE can be easily adapted for use as an incomplete verifier for such applications. What we know in this case is that if there exists a real counterexample then there exists a pseudorun counterxample. So WAVE can search for pseudorun counterxamples, and say "YES" if none is found. However, if a pseudorun counterexample is found, WAVE can only say "NO" (it can not check "if this is a real counterexample", because even if the pseudorun does not satisfy the arithmetic semantics, it could "come" from a genuine run that does, and there is no way to check this). So in summary, if WAVE says "YES", then the property is satisfied. If it says "NO", the property may or may not be satisfied. So WAVE can give false negatives but never false positives in this situation; 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 **X** (intuitively, **X** is a problem in general because the concurrency allowed by workflows leads to interleaving of activities, making "next state" constraints unenforceable).

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This dissertation studies the formal modeling of data-driven Web applications provided by Web sites interacting with users or applications, and develops a range of analyses and verification approaches which help designers to enhance flexibility and reliability, and ensure the implementation of web applications that will satisfy some mission-critical properties.

In the scenario we consider, the Web applications can access an underlying database, as well as state information updated as the interaction progresses, and receive user input. The structure and contents of Web pages, and the actions to be taken, are determined dynamically by querying the underlying database as well as the state and the input.

To describe and reason about Web applications, we start from the formal specification language of Web applications, where each Web application can be specified using databases, states, inputs, actions, web page set and page schemas. The structure of the Web page the user sees, at any given point, is described by a Web page schema. The content 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 semantics of the specification language is declarative, and thus, easier to explain to the participants. We are able to reason about the Web application and prove whether it satisfies some business requirements, which are expressed in properties.

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. For example, in an e-commerce application, it may be desirable to verify that no product is delivered before payment of the right amount is received. We have identified a practically appealing and fairly tight class of Web applications 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[1]. Moreover, the class is shown to be tight, since even slight relaxation leads to undecidability. For branching-time properties, we identify decidable restrictions for which the complexity of verification ranges from PSPACE to 2-EXPTIME. To obtain these results, we use a mix of techniques from logic and automatic verification.

While the PSPACE algorithm guarantees decidability, it does not demonstrate the practical feasibility of verification, since a naive implementation would immediately be intractable. To explore the practical feasibility of verification, we implemented WAVE–a verifier for input-bounded LTL-FO properties and Web applications, via a series of successive refinements to an initial implementation, whose performance improved from "decidable but impractical" to "feasible, with running times within seconds".

The contributions of this dissertation are:

- it identifies classes of Web applications for which temporal properties can be checked, and establishes their complexity.

- it shows that even slight relaxation of our restrictions leads to undecidability

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

of verification. Thus, our decidability results are quite tight.

- it extends the finite-state model checking techniques to data-aware reactive systems in general, and data-driven interactive Web applications in particular.
- it presents a technique to reduce infinite data value to finite symbolic data.
- it exploits and extends the techniques for automatical verification of Web applications, under certain assumptions.

6.2 Future Work

Other interesting aspects of Web application verification could not be addressed in this dissertation and are left for future work. We are especially interested in pursuing the following research topics.

6.2.1 Specifying and verifying sessions and multiple users

In practice, it is not always realistic to assume that verification applies to *all* possible runs of the Web application. This may be due to various reasons: there may be a need to verify properties of complex applications 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).

6.2.2 Verification of Web Service Compositions

Web applications can be regarded as Web services communicating only with external users via a web browser interface. Many settings, however, require services to interact with each other, typically by exchanging messages. For instance, even seemingly self-contained e-commerce Web sites place calls to an external Web service to charge a credit card. The interaction might be a source of error; we are interested in extending our verification work on Web applications to compositions of Web services interacting through asynchronous messages.

We already obtained preliminary theoretical results[19], in which we considered two formalisms for property specification, namely Linear Temporal First-Order Logic and Conversation Protocols. Classical conversation protocols are concerned only with the sequence of message names observed during the interaction. We extended them with awareness of the message contents. For both property formalisms, we mapped the boundaries of verification decidability. In particular, we explored various semantics for message-based communication (singleton versus set messages, lossy versus perfect communication channels, bounded versus unbounded received message queues). We also identify syntactic restrictions on the peer and property specifications which, under appropriate communication semantics, guarantee decidability of verification in PSPACE. This complexity is the best one can hope for. We show that our restrictions are quite tight: even slight relaxations thereof lead to undecidability. The favorable experimental results obtained in[20, 18] for verification of individual input-bounded services suggest that similarly good performance can be expected for compositions. We plan to pursue this in future research.

6.2.3 Web Services Discovery

A fundamental problem related to web services concerns service discovery. For example, service composition may start with locating appropriate existing services that can be used and integrated. Requirements used to select services may focus on different aspects of web services, varying from service categories to service semantics, from service interfaces(that is, input, output, messages), to service behaviors(properties that need to be satisfied). Current work on service discovery focuses on functional description, which is expressed in terms of the transformation produced by the service. Specifically, it specifies the inputs required by the service and the outputs generated. In my view, functional description should be augmented with operational descriptions such as specifications of conversations (sequences of messages exchanged by the service), of the events and activities performed by the service, and of semantics of services (e.g., input, output, preconditions and postconditions).

Operational descriptions of Web services are gaining more and more interests. For example, WSCL and WSCDL attempt to capture conversations between interacting services, WSMO works on semantic web services, and OWL-S introduces preconditions and results to each activity in the flow of a service. With the development of these standards, we envision that more services could become available with their operational descriptions exported.

For the operational description, we can use temporal properties to describe the service. Assume that both the existing web services and the target service are described by a set of properties against some global schema; web service discovery tries to find an existing service that is equivalent to the target service. How to define the equivalence of two web services is an interesting problem in itself, and might be based on property inference. But the most challenging and interesting problem is how to index the existing web services, in order to make discovery more natural and efficient in a huge number of services, since checking whether a web service satisfies a property may be too expensive. Furthermore, we need to have a query language for the users to specify the target web service, which is easy to understand and captures the properties we want to express.

6.2.4 Synthesis of Web Service Compositions

Composition addresses the situation when a target service query cannot be realized by any single available service, but instead requires suitably combining "parts of" available services. Composition involves two different aspects: composition synthesis and orchestration. We are interested in exploring the problem of automatic composition synthesis of web services, which consists of synthesizing a specification of how to coordinate the component services, in order to realize the target service. Specifically, I wish to investigate how to specify existing services as well as target services; since the synthesis problem is usually undecidable, we wish to understand under what restrictions the synthesis problem becomes decidable, establish its complexity, and explore its practical feasibility.

Bibliography

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

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

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

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

[5] D. Berardi, D. Calvanese, G.De Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *Proc. of the 31st Int. Conf. on Very Large Data Bases (VLDB 2005)*, pages 613–624, 2005.

[6] D. Berardi, D. Calvanese, G.De Giacomo, M. Lenzerini, and M. Mecella. Automatic services composition based on behavioral descriptions. In *International Journal of Cooperative Information Systems (IJCIS), 2004*, 2004.

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

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

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

[10] 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.

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

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

[13] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 2000.

[14] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithm for the verification of temporal properties. In *Computer-Aided Verification, CAV '90*, 1990.

[15] 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.

[16] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Computer Aided Verification (CAV)*, pages 160–171, 1999.

[17] 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.

[18] A. Deutsch, L.Sui, V.Vianu, and D.Zhou. A system for specification and verification of interactive, data-driven web applications. In *ACM SIGMOD Int'l. Conf. on Management of Data*, 2006. Demo.

[19] A. Deutsch, L.Sui, V.Vianu, and D.Zhou. Verification of communicating data-driven web services. In *ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems (PODS)*, 2006.

[20] A. Deutsch, M.Marcus, L.Sui, V.Vianu, and D.Zhou. A verifier for interactive, data-driven web applications. In *ACM SIGMOD Int'l. Conf. on Management of Data*, 2005.

[21] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-intensive web services. In *Proc. ACM PODS*, pages 71–82, 2004.

[22] E. A. Emerson and A. P. Sistla. Deciding branching time logic. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 14–24, New York, NY, USA, 1984. ACM Press.

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

[24] 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.

[25] 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.

[26] P. Fraternali. Tools and approaches for developing data-intensive Web applications: a survey. *ACM Computing Surveys*, 31(3):227–263, 1999.

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

[28] 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.

[29] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Computer Aided Verification (CAV)*, pages 72–83, 1997.

[30] P. Graunke, R. Findler, S. Krishnamurthi, and M. Felleisen. Modeling web interactions, 2003.

[31] Paul T. Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Modeling web interactions. In *European Symposium on Programming*, 2003.

[32] Y. Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, 2000.

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

[34] G. Holzmann. *The SPIN Model Checker - Primer and Reference (M)anual.* Addison-Wesley, 2003.

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

[36] D. E. Knuth. The art of computer programming. Vol.3:(2nd ed.) Sorting and searching. 1998.

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

[38] Daniel R. Licata and Shriram Krishnamurthi. Verifying interactive web programs. In *IEEE International Symposium on Automated Software Engineering*, 2004.

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

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

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

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

[43] D. E. Muller, A. Saoudi, and P. E. Schupp. Alternating automata, the weak monadic theory of the tree, and its complexity. In *International Colloquium on Automata, Languages and Programming on Automata, languages and programming*, pages 275–283, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[44] S. Nakajima. Verification of web service flows with model-checking techniques. In *International Symposium on Cyber Worlds (CW)*, pages 378–385, 2002.

[45] Srini Narayanan and Sheila A. McIlraith. Simulation, verification and automated composition of web services. In *International World Wide Web Conference (WWW)*, pages 77–88, 2002.

[46] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[47] A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for buechi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

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

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

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

[51] M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *J. Comput. Syst. Sci.*, 32(2):183–221, 1986.

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

[53] P. Wolper. Constructing automata from temporal logic formulas: a tutorial. pages 261–277, 2002.

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

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

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

[57] F. Yang, J. Shanmugasundaram, M. Riedewald, J. Gehrke, and A. Demers. Hilda: A high-level language for data-driven web applications. In *Proceedings of the 22nd IEEE International Conference on Data Engineering (ICDE 2006)*, April 2006.