Verification of Fair Transition Systems

Orna Kupferman[1] and Moshe Y. Vardi[2]

[1] Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.
Email: ok@research.att.com
[2] Rice University, Department of Computer Science, P.O. Box 1892, Houston, TX 77251-1892, U.S.A.
Email: vardi@cs.rice.edu, URL: http://www.cs.rice.edu/~vardi

Abstract. In *program verification*, we check that an *implementation* meets its *specification*. Both the specification and the implementation describe the possible behaviors of the program, though at different levels of abstraction. We distinguish between two approaches to implementation of specifications. The first approach is *trace-based implementation*, where we require every computation of the implementation to correlate to some computation of the specification. The second approach is *tree-based implementation*, where we require every computation tree embodied in the implementation to correlate to some computation tree embodied in the specification. The two approaches to implementation are strongly related to the linear-time versus branching-time dichotomy in temporal logic.
In this work we examine the trace-based and the tree-based approaches from a *complexity-theoretic* point of view. We consider and compare the complexity of verification of *fair transition systems*, modeling both the implementation and the specification, in the two approaches. We consider *unconditional, weak*, and *strong* fairness. For the trace-based approach, the corresponding problem is *language containment*. For the tree-based approach, the corresponding problem is *fair simulation*. We show that while both problems are PSPACE-complete, their complexities in terms of the size of the implementation do not coincide and the trace-based approach is more efficient. As the implementation is normally much bigger than the specification, we see this as an advantage of the trace-based approach. Our results are at variance with the known results for the case of transition systems with no fairness, where the tree-based approach is more efficient.

1 Introduction

In *program verification*, we check that an *implementation* meets its *specification*. Both the specification and the implementation describe the possible behaviors of the program, but the implementation is more concrete than the specification, or, equivalently, the specification is more abstract than the implementation (cf. [AL91]). This basic notion of verification suggests a top-down method for design development. Starting with a highly abstract specification, we can construct a sequence of "behavior descriptions". Each description refers to its predecessor as a specification, so it is less abstract than its predecessor. The last description contains no abstractions, and constitutes the implementation. Hence the name *hierarchical refinement* for this methodology (cf. [LS84, LT87, Kur94]).

We distinguish between two approaches to implementation of specifications. The first approach is *trace-based implementation*, where we require every computation of the implementation to *correlate* to some computation of the specification. The second approach is *tree-based implementation*, where we require every computation tree embodied in the implementation to correlate to some computation tree embodied in the specification. The exact notion of correct implementation then depends on how we interpret correlation. We can, for example, interpret correlation as identity. Then, correct trace-based implementation is one in which every computation is also a computation of the specification, and correct tree-based implementation is one in which every embodied computation tree is also embodied in the specification. Numerous interpretations of correlation are suggested and studied in the literature [Hen85, Mil89, AL91]. Here, we consider a very simple definition of correlation and interpret it as equivalence with respect to the variables joint to the implementation and the specification, as the implementation is typically defined over a wider set of variables, reflecting the fact that it is more concrete than the specification.

The tree-based approach is stronger in the following sense. If $\mathcal{I}$ is a correct tree-based implementation of the specification $\mathcal{S}$, then $\mathcal{I}$ is also a correct trace-based implementation of $\mathcal{S}$. As shown by Milner [Mil80], the opposite direction is not true. The two approaches to implementation are strongly related to the linear-time versus branching-time dichotomy in temporal logic [Pnu85]. The temporal-logic analogy to the strength of the tree-based approach is the expressiveness superiority of $\forall$CTL*, the universal fragment of CTL*, over LTL [CD88]. Indeed, while a correct trace-based implementation is guaranteed to satisfy all the LTL formulas satisfied in the specification, a correct tree-based implementation is guaranteed to satisfy all the $\forall$CTL* formulas satisfied in the specification [GL94].

In this work we examine the traced-based and the tree-based approaches from a *complexity-theoretic* point of view. More precisely, we consider and compare the complexity of the problem of determining whether $\mathcal{I}$ is a correct trace-based implementation of $\mathcal{S}$, and the problem of determining whether $\mathcal{I}$ is a correct tree-based implementation of $\mathcal{S}$. The different levels of abstraction in the implementation and the specification are reflected in their sizes. The implementation is typically much larger than the specification and it is its size that is the computational bottleneck. Therefore, of particular interest to us is the *implementation complexity* of these problems; i.e., their complexity in terms of $\mathcal{I}$, assuming $\mathcal{S}$ is fixed.

We model specifications and implementations by *transition systems* [Kel76]. The systems are defined over the sets $AP_{\mathcal{I}}$ and $AP_{\mathcal{S}}$ of *atomic propositions* used in the implementation and specification, respectively. Thus, the alphabets of the systems are $2^{AP_{\mathcal{I}}}$ and $2^{AP_{\mathcal{S}}}$. Recall that usually the implementation has more variables than the specification. Hence, $AP_{\mathcal{I}} \supseteq AP_{\mathcal{S}}$. We therefore interpret correlation as equivalence with respect to $AP_{\mathcal{S}}$. In other words, associating computations and computation trees of the implementation with these of the specification, we first projet them on $AP_{\mathcal{S}}$. Within this framework, correct trace-based implementation corresponds to *trace containment* and correct tree-based implementation corresponds to *simulation* [Mil71]. Since simulation can be checked in polynomial time [Mil80, BGS92], whereas the trace containment problem is PSPACE-complete [MS72] [3], it seems that the tree-based approach is more efficient than the trace-based approach. This is reminiscent of the computational advantage of branching-time model checking over linear-time model checking [CES86, LP85, QS81, VW86].

Once, however, we want our implementations and specifications to describe behaviors that satisfy both *liveness* and *safety* properties, transition systems are too weak. Then, we need the framework of *fair transition systems*. We consider *unconditional*, *weak*, and *strong* fairness (also known as *impartiality*, *justice*, and *compassion*, respectively) [LPS81, Eme90, MP92]. Within this framework, correct trace-based implementation corresponds to *language containment* and correct tree-based implementation corresponds to *fair simulation* [BBLS92, ASB$^+$94, GL94]. Hence, it is the complexity of these problems that should be examined when we compare the trace-based and the tree-based approaches.

We present a uniform method and a simple algorithm for solving the language-containment problem for all the three types of fairness conditions. Unlike [CDK93], we consider the case where both the specification and the implementation are nondeterministic, as is appropriate in a hierarchical refinement framework. We prove that the problem is PSPACE-complete for all the three types. For the case the implementation uses the unconditional or weak fairness conditions, our nondeterministic algorithm requires space logarithmic in the size of the implementation (regardless the type of fairness condition used in the specification). For the case the implementation uses the strong fairness condition, we suggest an alternative algorithm that runs in time polynomial in the size of the implementation. We show that these algorithms are optimal; thus the implementation complexity of language containment

[3] The reduction in [MS72] considers containment of languages defined by regular expressions and can be extended to consider trace containment.

is NLOGSPACE-complete for implementations that use the unconditional or weak fairness conditions and is PTIME-complete for implementations that use the strong fairness condition. To prove the latter, we show that the nonemptiness problem for fair transition systems with a strong fairness condition is PTIME-hard, which is most likely harder than the NLOGSPACE bounds known for the unconditional and weak fairness conditions [VW94].

We also present a uniform method and a simple algorithm for solving the fair-simulation problem for all the three types of fairness conditions. Our algorithm uses the language-containment algorithm as a subroutine. We prove that the problem is PSPACE-complete for all the three types. Like Milner's algorithm for checking simulation [Mil90], our algorithm can be implemented as a calculation of a fixed-point expression, significantly improving its practicality. The running time of our algorithm is polynomial in the size of the implementation. We show that this is optimal; thus, the implementation complexity of fair simulation is PTIME-complete for all types of fairness conditions. Proving the latter we prove that the implementation complexity of simulation (without fairness conditions) is PTIME-complete too.

Our results show that when we model the specification and the implementation by fair transition systems, the advantage of the tree-based approach disappears. Furthermore, when we consider the implementation complexity, then checking implementations that use unconditional or weak fairness conditions is easier in the trace-based approach.

2 Preliminaries

2.1 Fair Transition Systems

A *fair transition system* (*transition system*, for short) $S = \langle \Sigma, W, R, W_0, L, \alpha \rangle$ consists of an alphabet Σ, a set W of states, a total transition relation $R \subseteq W \times W$ (i.e., for every $w \in W$ there exists $w' \in W$ such that $R(w, w')$), a set W_0 of initial states, a labeling function $L : W \to \Sigma$, and a fairness condition α. We will define three types of fairness conditions shortly. A *computation* of S is a sequence $\pi = w_0, w_1, w_2, \ldots$ of states such that for every $i \geq 0$ we have $R(w_i, w_{i+1})$. In order to determine whether a computation is *fair*, we refer to the set $Inf(\pi)$ of states that π visits infinitely often. Formally,

$$Inf(\pi) = \{w \in W \ : \ \text{for infinitely many } i \geq 0, \ \text{we have } w_i = w\}.$$

The way we refer to $Inf(\pi)$ depends in the fairness condition of S. Several types of fairness conditions are studied in the literature. We consider here three:

- *Unconditional* fairness (or *impartiality*), where $\alpha \subseteq W$, and π is fair iff $Inf(\pi) \cap \alpha \neq \emptyset$.
- *Weak* fairness (or *justice*), where $\alpha \subseteq 2^W \times 2^W$, and π is fair iff for every pair $\langle L, R \rangle \in \alpha$, we have that $Inf(\pi) \cap (W \setminus L) = \emptyset$ implies $Inf(\pi) \cap R \neq \emptyset$.
- *Strong* fairness (or *fairness*), where $\alpha \subseteq 2^W \times 2^W$, and π is fair iff for every pair $\langle L, R \rangle \in \alpha$, we have that $Inf(\pi) \cap L \neq \emptyset$ implies $Inf(r) \cap R \neq \emptyset$.

It is easy to see is that fair transition systems are essentially a notational variant of automata on infinite words [Tho90]. Thus, we will be able to use freely results from the theory of such automata. In particular, the unconditional and the strong fairness conditions correspond to the *Büchi* and *Streett* acceptance conditions.

For a state w, a w-computation is a computation $w_0, w_1, w_2, \ldots$ with $w_0 = w$. We use $\mathcal{L}(S^w)$ to denote the set of all words $\sigma_0 \cdot \sigma_1 \cdots \in \Sigma^\omega$ for which there exists a fair w-computation $w_0, w_1, \ldots$ in S with $L(w_i) = \sigma_i$ for all $i \geq 0$. The language $\mathcal{L}(S)$ of S is then defined as $\bigcup_{w \in W_0} \mathcal{L}(S^w)$. Thus, each transition system defines a subset of Σ^ω. We say that a transition system S is *empty* iff $\mathcal{L}(S) = \emptyset$; i.e., S has no fair computation. We sometimes say that S accepts w, meaning that $w \in \mathcal{L}(S)$.

The size of a transition system and its fairness condition, determine the *complexity* of solving questions about it. We define classes of transition systems according to these two characteristics. We write $\mathcal{U}$, $\mathcal{W}$, and $\mathcal{S}$ to denote the unconditional, weak, and strong fairness conditions, respectively. We measure the size of a transition system by its number of states (the number of edges is at most quadratic in the number of states) and, in the case of weak and strong fairness, also by the number of pairs in its fairness condition. For example, an unconditionally fair transition system with n states is denoted $\mathcal{U}(n)$. We also use a line over the transition system to denote the complementary transition system (one that accepts the complementary language). For example, the transition system complementing a strongly fair transition system with n states and m pairs is denoted $\overline{\mathcal{S}(n,m)}$.

2.2 The Language-Containment and the Fair-Simulation Problems

In this section we formalize correct trace-based and tree-based implementations in terms of language containment and fair simulation between an implementation $\mathcal{I}$ and a specification $\mathcal{S}$. Recall that $\mathcal{I}$ and $\mathcal{S}$ are given as fair transition systems over the alphabets $2^{AP_{\mathcal{I}}}$ and $2^{AP_{\mathcal{S}}}$ respectively, with $AP_{\mathcal{I}} \supseteq AP_{\mathcal{S}}$. For technical convenience, we assume that $AP_{\mathcal{I}} = AP_{\mathcal{S}}$; thus, the implementation and the specification are defined over the same alphabet. By taking, for each $\sigma \in 2^{AP_{\mathcal{I}}}$, the letter $\sigma \cap AP_{\mathcal{S}}$ instead the letter σ, all our algorithms and results are valid also for the case $AP_{\mathcal{I}} \supset AP_{\mathcal{S}}$.

Given two transition systems S and S' over the same alphabet, the *language containment* problem of S and S' is to determine whether $\mathcal{L}(S) \subseteq \mathcal{L}(S')$. That is, whether every word accepted by S is also accepted by S'. While language containment refers to each word in $\mathcal{L}(S)$ independently, *fair simulation* refers also to the branching structure of the transition system.

Let S and S' be two transition systems over the same alphabet and let $H \subseteq W \times W'$ be a relation over their states. It is convenient to extend H to relate also infinite computations of S and S'. For two computations $\pi = w_0, w_1, \ldots$ in S, and $\pi' = w'_0, w'_1, \ldots$ in S', we say that $H(\pi, \pi')$ holds iff $H(w_i, w'_i)$ holds for all $i \geq 0$. For a pair $\langle w, w' \rangle \in W \times W'$, we say that $\langle w, w' \rangle$ is *good in* H iff for every fair w-computation π in S, there exists a fair w'-computation π' in S', such that $H(\pi, \pi')$.

Let w and w' be states in W and W', respectively. A relation $H \subseteq W \times W'$ is a *simulation relation* from $\langle S, w \rangle$ to $\langle S', w' \rangle$ iff the following conditions hold:

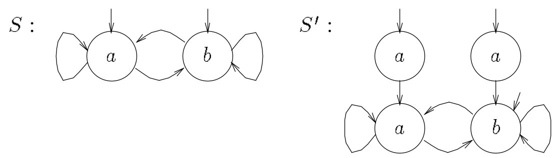
(1) $H(w, w')$.
(2) For all t and t' with $H(t, t')$, we have $L(t) = L(t')$.
(3) For all t and t' with $H(t, t')$, the pair $\langle t, t' \rangle$ is good in H.

A simulation relation H is a *simulation from* S *to* S' iff for every $w \in W_0$ there exists $w' \in W'_0$ such that $H(w, w')$. If there exists a simulation from S to S', we say that S *simulates* S' and we write $S \leq S'$. Intuitively, it means that the transition system S' has more behaviors than the transition system S. In fact, every computation tree embodied in S is embodied in S'. The *fair-simulation problem* is, given S and S', to determine whether $S \leq S'$.

It is easy to see that fair simulation implies language containment. That is, if $S \leq S'$ then $\mathcal{L}(S) \subseteq \mathcal{L}(S')$. The opposite, however, is not true. In the figure below we present two transition systems S and S' such that the language of both transition systems is $(a+b)^\omega$. As such, $\mathcal{L}(S) \subseteq \mathcal{L}(S')$, but still, S does not simulate S'. Indeed, no initial state of S' can be paired, by any H, to the initial state labeled a of S.

3 The Complexity of the Language-Containment Problem

Theorem 1. *The language-containment problem* $\mathcal{L}(S) \subseteq \mathcal{L}(S')$ *for* $S \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ *and* $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ *is PSPACE-complete.*



Proof: As there are three possible types for the transition system S and three possible types for the transition system S', we have nine containment problems to solve in order to prove a PSPACE upper bound. We solve them all using the same method:

(1) Translate the transition system S to an unconditionally fair transition system S_U.
(2) Construct an unconditionally fair transition system $\overline{S'_U}$ that complements the transition system S'.
(3) Check $\mathcal{L}(S_U) \cap \mathcal{L}(\overline{S'_U})$ for nonemptiness.

This is how we perform step (1) for the three possible types of S.

1. $\mathcal{U}(n) \rightarrow \mathcal{U}(n)$.
2. $\mathcal{W}(n, m) \rightarrow \mathcal{U}(nm)$ [easy, and will be proven in the full version].
3. $\mathcal{S}(n, m) \rightarrow \mathcal{U}(n2^{O(m)})$ [not hard, and will be proven in the full version].

This is how we perform step (2) for the three possible types of S'.

1. $\overline{\mathcal{U}(n)} \rightarrow \mathcal{U}(2^{O(n \log n)})$ [Saf88].
2. $\overline{\mathcal{W}(n, m)} \rightarrow \overline{\mathcal{U}(nm)} \rightarrow \mathcal{U}(2^{O(nm \log(nm))})$ [Saf88].
3. $\overline{\mathcal{S}(n, m)} \rightarrow \mathcal{U}(2^{O(nm \log(nm))})$ [Saf92].

For all the three types of S, going to S_U involves an at most exponential blow up. Similarly, for all the three types of S', going to $\overline{S'_U}$ involves an at most exponential blow up. Thus, the size of the product of S_U and $\overline{S'_U}$ is exponential in the sizes of S and S' [Cho74] and checking it for nonemptiness can be done in space polynomial in their sizes [VW94].

Hardness in PSPACE follows from the known PSPACE lower bound for the case where both S and S' are unconditionally fair [Wol82]. Since $\mathcal{U}(n) \rightarrow \mathcal{W}(n, 1)$ and $\mathcal{U}(n) \rightarrow \mathcal{S}(n, 1)$, we can not do better with weak or strong fairness. $\square$

Recall that our main concern is the complexity in terms of the (much larger) implementation. We now turn to consider the implementation complexity of language containment.

Theorem 2. *The implementation complexity of checking $\mathcal{L}(S) \subseteq \mathcal{L}(S')$ for $S \in \{\mathcal{U}, \mathcal{W}\}$ and $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ is NLOGSPACE-complete.*

Proof: In the case where $S \in \{\mathcal{U}, \mathcal{W}\}$, the translation of S to S_U involves only a polynomial blow up. Thus, in this case, fixing the size of S', the nondeterministic algorithm described in the proof of Theorem 1 requires space logarithmic in the size of S. Since we can solve the nonemptiness problem of a transition system by checking its containment in a fixed-size empty transition system, hardness in NLOGSPACE follows from the NLOGSPACE lower bound for the nonemptiness problem of unconditionally fair transition systems [VW94]. $\square$

So, for the case where the implementation does not use the strong fairness condition, our language-containment algorithm requires space that is only logarithmic in the size of the implementation. Clearly, this is not the case when the implementation does use the strong fairness condition. Then, our algorithm requires space that is polynomial in the size of the implementation and time that is exponential in the size of the implementation. We can, however, do better.

Theorem 3. *The implementation complexity of checking $\mathcal{L}(S) \subseteq \mathcal{L}(S')$ for $S \in \{\mathcal{S}\}$ and $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ is in PTIME.*

Proof: We are going to use the following known results.

1. For $S_1 \in \mathcal{S}(n_1, m)$ and $S_2 \in \mathcal{U}(n_2)$, there exists $S \in \mathcal{S}(n_1 n_2, m+1)$ such that $\mathcal{L}(S) = \mathcal{L}(S_1) \cap \mathcal{L}(S_2)$ [easy, and will be proven in the full version].
2. The nonemptiness problem for strongly fair transition systems can be solved in polynomial time [EL85].

Given S and S', we construct, as in the proof of Theorem 1, the unconditionally fair transition system $\overline{S'_U}$. Unlike the algorithm there, we do not translate the transition system S to an unconditionally fair system. Rather, we check the nonemptiness of $\mathcal{L}(S) \cap \mathcal{L}(\overline{S'_U})$. By 1 and 2 above, this can be done in time polynomial in the size of S. □

Note that the algorithm presented in the proof of Theorem 3 uses time and space exponential in the size of the specification, in contrast to the algorithm in the proof of Theorem 1 that uses space polynomial in the size of the specification. Nevertheless, as S' is usually much smaller than S, the algorithm in the proof of Theorem 3 may work better in practice. Can we do better and get the NLOGSPACE complexity we have for implementations that use the unconditional or weak fairness conditions? As we now show, the answer to this question is negative. To see this, we first need the following theorem.

Theorem 4. *The nonemptiness problem for strongly fair transition systems is PTIME-hard.*

Proof: We do a reduction from Propositional Anti-Horn Satisfiability (PAHS). Propositional Anti-Horn clauses are obtained from Propositional Horn clauses by replacing each proposition p with $\neg p$. Thus, a propositional anti-Horn clause is either of the form $p \rightarrow q_1 \vee \cdots \vee q_n$ (an empty disjunction is equivalent to false) or of the form $q_1 \vee \cdots \vee q_n$. As Propositional-Horn Satisfiability is PTIME-complete [Pla84], then clearly, so is PAHS.

Given an instance I of PAHS we construct the transition system $S_I = \langle W, W, W \times W, \{w_0\}, L, \alpha \rangle$, where W is the set of all the propositions in I, the initial state w_0 is an arbitrary state in W, $L(w) = w$ for $w \in W$, and α is the strong fairness condition defined as follows.

- For a clause $p \rightarrow q_1 \vee \ldots \vee q_n$ in I, we have $\langle \{p\}, \{q_1, \cdots, q_n\} \rangle$ in α.
- For a clause $q_1 \vee \ldots \vee q_n$ in I, we have $\langle W, \{q_1, \cdots, q_n\} \rangle$ in α.

We can view each computation of S_I as an assignment to the propositions in I. A proposition is assigned $\mathtt{true}$ iff the computation visits it infinitely often. The definition of α thus guarantees that I is satisfiable iff S_I is nonempty. □

So, unlike unconditionally or weakly fair transition systems, for which the nonemptiness problem is NLOGSPACE-complete, testing strongly fair transition systems for nonemptiness is PTIME-complete. Theorems 3 and 4 imply the following theorem.

Theorem 5. *The implementation complexity of checking $\mathcal{L}(S) \subseteq \mathcal{L}(S')$ for $S \in \{\mathcal{S}\}$ and $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ is PTIME-complete.*

4 The Complexity of the Fair-Simulation Problem

4.1 Upper Bound

Theorem 6. *The fair-simulation problem $S \leq S'$ for $S \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ and $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ is in PSPACE.*

Proof (sketch): Given $S = \langle \Sigma, W, R, W_0, L, \alpha \rangle$ and $S' = \langle \Sigma, W', R', W_0', L', \alpha' \rangle$, we show how to check in polynomial space that a candidate relation H is a simulation from S to S'. The claim then follows, since we can enumerate using polynomial space all candidate relations. First, we check, easily, that for every $w \in W_0$ there exists $w' \in W_0'$ such that $H(w, w')$. We then check, also easily, that for all $\langle w, w' \rangle \in H$, we have $L(w) = L(w')$. It is left to check that for all $\langle w, w' \rangle \in H$, the pair $\langle w, w' \rangle$ is good in H. To do this, we define, for every $\langle w, w' \rangle \in H$, two transition systems. The alphabet of both systems is W. The first transition system, A_w, accepts all the fair w-computations in S. The second transition system, $U_{w'}$, accepts all the sequences π in W^ω for which there exists a fair w'-computation π' in S' such that $H(\pi, \pi')$. Clearly, the pair $\langle w, w' \rangle$ is good in H iff $\mathcal{L}(A_w) \subseteq \mathcal{L}(U_{w'})$.

We define A_w and $U_{w'}$ as follows. The system A_w does nothing but tracing the w-computations of S, accepting these that satisfy S's acceptance condition. Formally, $A_w = \langle W, W, R, \{w\}, L'', \alpha \rangle$, where for all $w \in W$, we have $L''(w) = w$.

The transition system $U_{w'}$ has members of H as its set of states. Thus, each state has two elements. The second element of each state in $U_{w'}$ is a state in W' and it induces, according to R', the transitions. The first element in each state of $U_{w'}$ is a state in W and it induces the labeling. This combination guarantees that a computation $\pi'' \in H^\omega$ whose W'-elements form the computation $\pi' \in {W'}^\omega$ and whose states are labeled with $\pi \in W^\omega$, satisfies $H(\pi, \pi')$. Formally, $U_{w'} = \langle W, H, R'', W_0'', L'', \alpha'' \rangle$, where $W_0'' = (W \times \{w'\}) \cap H$, for every $\langle t, t' \rangle \in H$, we have $L''(\langle t, t' \rangle) = t$, the fairness condition α'' is adjusted to the new state space (i.e., each set $L \subseteq W'$ in α' is replaced by the set $(W \times L) \cap H$ in α''), and the transition relation R'' is also adjusted to the new state space (i.e., $R''(\langle t, t' \rangle, \langle q, q' \rangle)$ iff $R'(t', q')$). Note that R'' is not necessarily total. For that, we restrict $U_{w'}$ to states that have at least one R''-successor. Clearly, this does not effect the language of $U_{w'}$.

According to Theorem 1, checking that $\mathcal{L}(A_w) \subseteq \mathcal{L}(U_{w'})$ can be done in space polynomial in the sizes of A_w and $U_{w'}$, thus polynomial in S and S'. $\square$

We note that our algorithm can be easily adjusted to check S and S' for fair *bisimulation*.

4.2 Lower Bound

For a transition system $S = \langle \Sigma, W, R, W_0, L, \alpha \rangle$, we say that S is *universal* iff $\mathcal{L}(S) = \Sigma^\omega$. The *universality problem* is to determine whether a given transition system is universal. Meyer and Stockmeyer proved that the problem of determining whether the language of an automaton over finite words is Σ^* is PSPACE-complete [MS72]. We give here the details of the proof, easily adjusted to infinite words.

Theorem 7. *The universality problem is PSPACE-hard.*

Proof (sketch): We do a reduction from polynomial-space Turing machines. Given a Turing machine T of space complexity $s(n)$, we construct a transition system S_T of size linear in T and $s(n)$ such that S_T is universal iff T does not accept the empty tape. We assume, without loss of generality, that once T reaches a final state it loops there forever. The system S_T accepts a word w iff w is not an encoding of a legal computation of T over the empty tape or if w is an encoding of a legal yet rejecting computation of T over the empty tape. Thus, S_T rejects a word w iff it encodes a legal and accepting computation of T over the empty tape. Hence, S_T is universal iff T does not accept the empty tape.

Below we give the details of the construction of S_T. Let $T = \langle \Gamma, Q, \rightarrow, q_0, F \rangle$, where Γ is the alphabet, Q is the set of states, $\rightarrow \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$ is the transition relation (we use $(q, a) \rightarrow (q', b, \Delta)$ to indicate that when T is in state q and it reads the input a in the current tape cell, it moves to state q', writes b in the current tape cell, and its reading head moves one cell to the left/right, according to Δ), q_0 is the initial state, and $F \subseteq Q$ is the set of accepting states. We encode a configuration of T by a word $\#\gamma_1\gamma_2 \ldots (q, \gamma_i) \ldots \gamma_{s(n)}\#$. That is, a configuration starts and ends with #, all its other letters are in Γ, except for one letter in $Q \times \Gamma$. The meaning of such a configuration is that the j's cell in T, for $1 \leq j \leq s(n)$, is labeled γ_j, the reading head points on cell i, and T is in state q. For example, the initial configuration of T is $\#(q_0, b)b \ldots b\#$ where b stands for an empty cell. We can now encode a computation of T by a sequence of configurations (with only one # between two configurations).

Let $\Sigma = \{\#\} \cup \Gamma \cup (Q \times \Gamma)$ and let $\#\sigma_1 \ldots \sigma_{s(n)}\#\sigma'_1 \ldots \sigma'_{s(n)}\#$ be two successive configurations of T. For each triple $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle$ with $1 \leq i \leq s(n)$, we know, by the transition relation of T, what σ'_i should be. Let $next(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$ denote our expectation for σ'_i. For example, $next(\langle \gamma_{i-1}, \gamma_i, \gamma_{i+1} \rangle)$ is γ_i, and $next(\langle (q, \gamma_{i-1}), \gamma_i, \gamma_{i+1} \rangle)$ is γ_i, in the case $(q, \gamma_{i-1}) \rightarrow (q', \gamma'_{i-1}, L)$, and is (q', γ_i), in the case $(q, \gamma_{i-1}) \rightarrow (q', \gamma'_{i-1}, R)$. In addition, since we want the letter # to repeat exactly every $s(n) + 1$ letters, we define $next(\langle \sigma_{s(n)}, \#, \sigma'_1 \rangle)$ as #. Consistency with $next$ now gives us a necessary condition for a word to encode a legal computation. In addition, the computation should start with the initial configuration.

In order to check consistency with $next$, S_T can use its nondeterminism and guess when there is a violation of $next$. Thus, S_T guesses $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle \in \Sigma^3$, guesses a position in the word, checks whether the three letters to be read starting this position are σ_{i-1}, σ_i, and σ_{i+1}, and checks whether $next(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$ is not the letter to come $s(n) + 1$ letters later. Once S_T sees such a violation, it goes to an accepting sink. In order to check that the first configuration is the initial configuration, S_T simply compares the first $s(n) + 2$ letters with $\#(q_0, b)b \ldots b\#$. Finally, checking whether a legal computation is accepting is also easy; the computation has to reach an accepting configuration (one with $q \in F$). $\square$

We would like to do a similar reduction in order to prove that the fair-simulation problem is PSPACE-hard. For every alphabet Σ, let S_Σ be the transition system $\langle \Sigma, \Sigma, \Sigma \times \Sigma, \Sigma, L_\Sigma, \alpha \rangle$, where $L_\Sigma(\sigma) = \sigma$ and α is such that all the computations of S_Σ are fair. That is, S_Σ is a universal transition system in which each state is associated with a letter $\sigma \in \Sigma$ and $\mathcal{L}(S_\Sigma^\sigma) = \sigma \cdot \Sigma^\omega$. For example, $S_{\{a,b\}}$ is the transition system S in Figure 2.2. It is easy to see that a transition system S over Σ is universal iff $\mathcal{L}(S_\Sigma) \subseteq \mathcal{L}(S)$. It is not true, however, that S is universal iff $S_\Sigma \leq S$. For example, the transition system S' in Figure 2.2 is universal yet $S_{\{a,b\}} \not\leq S'$. Our reduction overcomes this difficulty by defining S_T in such a way that if S_T is universal, then for each of its states w, we have $\mathcal{L}(S_T^w) = L(w) \cdot \Sigma^\omega$. For such S_T, we do have that S_T is universal iff $S_\Sigma \leq S_T$. Indeed, a relation that maps a state σ of S_Σ to all the states of S_T that are labeled with σ is a fair simulation.

Theorem 8. *The fair-simulation problem $S \leq S'$ for $S \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ and $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ is PSPACE-hard.*

Proof (sketch): As in the previous proof, we do a reduction from polynomial space Turing machines. Given the Turing machine T, let T' be as follows. Whenever T reaches an accepting configuration, T' cleans the tape and starts from the beginning (i.e., empty tape and initial state at the left end of the tape). Thus, T accepts the empty tape iff T' has an infinite computation, in which case it visits the initial configuration infinitely often. We now define a transition system S_T with the following behavior. Reading a word w, the transition system S_T checks for a violation of the transition relation of T' in w (by guessing a violation of $next$). If it sees a violation, it goes to an accepting sink. If it does not see a violation, it continues to trace the computation of T' forever. We define the fairness condition of S_T such that it accepts w iff it reaches the accepting sink or it never sees the initial configuration in w. This acceptance condition can be specified by a pair $\langle g, b \rangle$ of states where g is simply the accepting sink and b is a state that S_T passes in whenever it traces the initial configuration in w (note that since the initial configuration starts with # and has no other # in it, it is very easy to being traced). A computation of S_T is fair with respect to $\langle g, b \rangle$ iff it eventually visits g and never visits b. This fairness condition can be easily translated to unconditional, weak, and strong fairness; e.g., by making g an accepting sink and b

a rejecting sink. It follows that S_T does not accept a word w iff w has a finite prefix, not violating $next$, followed by an infinite computation of T' that passes in the initial configuration of T. Therefore, S_T is universal iff T does not accept the empty tape.

We, however, want more than universality test. We want to define S_T in such a way that if it is indeed universal, then for each of its states w, we have $\mathcal{L}(S_T^w) = L(w) \cdot \Sigma^\omega$. Let $S_T = \langle \Sigma, W, R, W_0, L, \alpha \rangle$. We define the transition system S'_T by adding to S_T transitions from all states to all the initial states, i.e., $S'_T = \langle \Sigma, W, R \cup (W \times W_0), W_0, \alpha \rangle$. We claim that the extension of S_T to S'_T preserves "non-universality". That is, if S'_T is universal, then so is S_T. Note that this is not the case for arbitrary transition systems. In S_T, however, if a word w is not accepted, then w is of the form yx where y is a prefix not violating $next$ and x is an infinite computation of T'. As such, all the suffixes of w are of that special form! Therefore, if w is not accepted by S_T, all its suffixes are also not accepted by S_T. Hence, w is not accepted by S'_T too.

We claim that S_T is universal iff for each state w of S'_T, we have $\mathcal{L}(S_T'^w) = L(w) \cdot \Sigma^\omega$. The direction from right to left follows from the fact that the extension of S_T to S'_T preserves non-universality and the fact that for every $\sigma \in \Sigma$, there exists $w_0 \in W_0$ with $L(w_0) = \sigma$. The second direction follows from the fact that each state w in S'_T has a transition to W_0.

We now have that $S_\Sigma \leq S'_T$ iff S_T is universal, thus $S_\Sigma \leq S'_T$ iff T does not accept the empty tape. Since the fairness conditions of both S_Σ and S'_T can be specified in terms of either unconditional, weak, or strong fairness, we are done.

$\square$

Theorems 6 and 8 together imply the following.

Theorem 9. *The fair-simulation problem $S \leq S'$ for $S \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ and $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ is PSPACE-complete.*

4.3 The Implementation Complexity of the Fair-Simulation Problem

So, fair simulation has the same complexity as language containment. In Theorem 10 below we show that when we consider the implementation complexity of fair simulation, the picture is different. Here, checking implementations that use the unconditional or weak fairness conditions is not easier than checking implementations that use the strong fairness condition. Hence, fair simulation is harder than language containment and the trace-based approach is more efficient.

Theorem 10. *The implementation complexity of checking $S \leq S'$ for $S \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ and $S' \in \{\mathcal{U}, \mathcal{W}, \mathcal{S}\}$ is PTIME-complete.*

Proof: We start with the upper bound. Consider the algorithm presented in the proof of Theorem 6. It checks whether a candidate relation H is a simulation. Once we fix S', then, by Theorems 2 and 5, the complexity of checking each pair in the candidate relation is NLOGSPACE for $S \in \{\mathcal{U}, \mathcal{W}\}$ and is PTIME for $S \in \{\mathcal{S}\}$. Once we fix S', the number of pairs in each candidate relation is polynomial in the size of S. Thus, fixing S, the problem of checking a candidate relation H is in PTIME. Instead of guessing a relation H and checking it, we do a fixed-point computation as follows (cf. [Mil90]). Let $H_0 = \{\langle w, w' \rangle : w \in W, w' \in W', \text{ and } L(w) = L(w')\}$. Thus, H_0 is the maximal relation that satisfies condition (1) of fair simulation. Consider the monotonic function $f : 2^{W \times W'} \to 2^{W \times W'}$, where

$$f(H) = H \cap \{\langle w, w' \rangle : \langle w, w' \rangle \text{ is good in } H\}.$$

Thus, $f(H)$ contains all the pairs in H that are good with respect to the relation H. Let H be the greatest fixed-point of f when restricted to pairs in H_0. That is, $H = \nu z. H_0 \cap f(z)$. It can be shown that $S \leq S'$

iff for every $w \in W_0$, we have $(\{w\} \times W'_0) \cap H \neq \emptyset$. Since $W \times W'$ is finite, we can calculate H iteratively, starting with H_0 until we reach a fixed-point. Now, as f is monotonic, we have to iterate it at most polynomially many times. Hence, out of the $2^{|W \times W'|}$ candidate relations for simulation, we actually check at most $|W \times W'|$ relations. Recall that if S' is fixed, the problem of checking a candidate relation is in PTIME. Also, if S' is fixed, we have only linearly many candidate relations to check. Hence, the problem is in PTIME.

We prove hardness in PTIME by reducing the NAND Circuit Value Problem (NANDCV), proved to be PTIME-complete in [Gol77, GHR95], to the problem of determining whether a transition system S simulates a fixed transition system S'. In the NANDCV problem, we are given a Boolean circuit α constructed solely of NAND gates of fanout 2, and a vector $\langle x_1, \ldots, x_n \rangle$ of Boolean input values. The problem is to determine whether the output of α on $\langle x_1, \ldots, x_n \rangle$ is 1. The idea of the reduction is as follows. We define a fixed transition system S' that embodies all the NAND circuits α and input vectors **x** for which the value of α on **x** is 1. Then, given a circuit α and an input vector **x**, we translate them to a transition system S such that $S \leq S'$ iff the value of α on **x** is 1.

The transition system S' has 12 states. Eight states correspond to internal gates. Each of these states is an entry in the Truth Table of the operator NAND, attributed with a direction, either L or R. Thus, the "internal states" of S' are $\langle 001L \rangle$, $\langle 011L \rangle$, $\langle 101L \rangle$, $\langle 110L \rangle$, $\langle 001R \rangle$, $\langle 011R \rangle$, $\langle 101R \rangle$, and $\langle 110R \rangle$. Four more states correspond to the input gates of the circuit. Each of these states is a Boolean value, attributed with a direction. Thus, the "input states" are $\langle 0L \rangle$, $\langle 1L \rangle$, $\langle 0R \rangle$, and $\langle 1R \rangle$. The intuition is that an internal state $\langle l, r, val, d \rangle$ corresponds to a NAND gate that has the value l in its left input, has the value r in its right input, and whose output val can be only a d-input of other gates. Similarly, an input state $\langle val, d \rangle$ corresponds to an input gate with output val that can only be a d input of other gates.

Accordingly, the transitions from an internal state $\langle l, r, val, d \rangle$ correspond to the possible ways of having l and r as right and left inputs, respectively. Thus, we have transitions from this state to all (internal or input) states with either $val = l$ and $d = L$ or $val = r$ and $d = R$. For example, the internal state $\langle 100L \rangle$ has transitions to the states $\langle 001L \rangle$, $\langle 011L \rangle$, $\langle 101L \rangle$, $\langle 110R \rangle$, $\langle 1L \rangle$, and $\langle 0R \rangle$. It has transitions from all states $\langle l, r, val, d \rangle$ with $l = 0$. In addition, the input states have self loops.

We label an internal state by either L or R according to its direction element. We label an input state by both its value and direction. We define the initial states of S' to be these with $val = 1$, and we impose no fairness condition. Clearly, the size of S' is fixed.

Now, S is simply α with attributions of directions. That is, we duplicate all gates and inputs of α so that the output of each gate is either always a left input of other gates, in which case we label it with L, or always a right input of other gates, in which case we label it with R. In addition, we add self loops to the input gates and label them with their values.

It is not hard to prove that for a simulation relation H from S to S' and for every pair $\langle s, \langle l, r, val, d \rangle \rangle$ or $\langle s, \langle val, d \rangle \rangle$ in H, the output of the gate s on the vector **x** is val. Hence, the output of α on **x** is 1 iff S simulates S'. $\square$

We note that our lower bound is different from the PTIME-hardness established for the bisimulation problem in [BGS92]. We consider simulation between two systems, one of them is fixed. Balcazar et al. consider bisimulation between the states of a single system, whose size is not fixed.

5 Discussion

We have examined the trace-based and the tree-based approaches to implementation from a complexity-theoretic point of view. Our results show that when we model the specification and the implementation by fair transition systems, the complexity of checking the correctness of a trace-based implementation coincides with that of checking the correctness a tree-based implementation. Furthermore, when we consider the implementation complexity, then checking implementations that use the unconditional or weak fairness condition is easier in the trace-based approach.

It is interesting to compare our results with the known complexities of LTL and $\forall$CTL* model checking. Trace-based implementations are part of the linear-time paradigm and correspond to LTL model checking. Tree-based implementations are part of the branching-time paradigm and correspond to $\forall$CTL* model checking. All the four problems are PSPACE-complete [SC85, EL85]. The model-checking algorithm of $\forall$CTL* uses as a subroutine the model-checking algorithm of LTL [EL85]. In a similar manner, our fair-simulation algorithm uses as a subroutine the language-containment algorithm. So, the implementation dichotomy and the temporal-logic dichotomy have a lot in common. When we turn to consider the program complexity of model checking, which is the analogue to our implementation complexity, this is no longer true. The program complexity of model checking for both LTL and $\forall$CTL* is NLOGSPACE-complete [VW86, BVW94]. In contrast, we saw here that implementation is easier in the trace-based approach.

References

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[ASB$^+$94] A. Aziz, V. Singhal, F. Balarin, R. Brayton, and A.L. Sangiovanni-Vincentelli. Equivalences for fair kripke structures. In *Proc. 21st Int. Colloquium on Automata, Languages and Programming*, Jerusalem, Israel, July 1994.

[BBLS92] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Proc. 4th Workshop on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, Montreal, June 1992. Springer-Verlag.

[BGS92] J. Balcazar, J. Gabarro, and M. Santha. Deciding bisimilarity is P-complete. *Formal Aspects of Computing*, 4(6):638–648, 1992.

[BVW94] O. Bernholtz, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In D. L. Dill, editor, *Computer Aided Verification, Proc. 6th Int. Conference*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155, Stanford, June 1994. Springer-Verlag, Berlin.

[CD88] E.M. Clarke and I.A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Proc. Workshop on Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, pages 428–437. Lecture Notes in Computer Science, Springer-Verlag, 1988.

[CDK93] E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. *Information Processing Letters* **46**, pages 301–308, (1993).

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[Cho74] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

[EL85] E.A. Emerson and C.-L. Lei. Temporal model checking under generalized fairness constraints. In *Proc. 18th Hawaii International Conference on System Sciences*, Hawaii, 1985.

[Eme90] E.A. Emerson. Temporal and modal logic. *Handbook of theoretical computer science*, pages 997–1072, 1990.

[GHR95] R. Greenlaw, H.J. Hoover, and W.L. Ruzzo. *Limits of parallel computation*. Oxford University Press, 1995.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.

[Gol77] L.M. Goldschlager. The monotone and planar circuit value problems are log space complete for p. *SIGACT News*, 9(2):25–29, 1977.

[Hen85] M. Hennessy. *Algebraic theory of Processes.* MIT Press, Cambridge, 1985.

[Kel76] R.M. Keller. Formal verification of parallel programs. *Comm ACM*, 19:371–384, 1976.

[Kur94] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processesl*. Princeton Univ. Press, 1994.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[LPS81] D. Lehman, A. Pnueli, and J. Stavi. Impartiality, justice, and fairness – the ethic of concurrent termination. In *Proc. 8th Colloq. on Automata, Programming, and Languages (ICALP)*, volume 115 of *Lecture Notes in Computer Science*, pages 264–277. Springer-Verlag, July 1981.

[LS84] S.S. Lam and A.U. Shankar. Protocol verification via projection. *IEEE Trans. on Software Engineering*, 10:325–342, 1984.

[LT87] N. A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 137–151, 1987.

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 481–489, September 1971.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1980.

[Mil89] R. Milner. *Communication and Concurrecny*. Prentice-Hall, Englewood Clifs, 1989.

[Mil90] R. Milner. Operational and algebraic semantics of concurrent processes. *Handbook of theoretical computer science*, pages 1201–1242, 1990.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, January 1992.

[MS72] A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential time. In *Proc. 13th IEEE Symp. on Switching and Automata Theory*, pages 125–129, 1972.

[Pla84] D.A. Plaisted. Complete problems in the first-order predicate claculus. *J. on Computer and System Sciences*, 29(1):8–35, 1984.

[Pnu85] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proc. 12th Int. Colloquium on Automata, Languages and Programming*, pages 15–32. Lecture Notes in Computer Science, Springer-Verlag, 1985.

[QS81] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th Int'l Symp. on Programming*, volume 137, pages 337–351. Springer-Verlag, Lecture Notes in Computer Science, 1981.

[Saf88] S. Safra. On the complexity of omega-automata. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, White Plains, October 1988.

[Saf92] S. Safra. Exponential determinization for ω-automata with strong-fairness acceptance condition. In *Proceedings of the 24th ACM Symposium on Theory of Computing*, Victoria, May 1992.

[SC85] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *J. ACM*, 32:733–749, 1985.

[Tho90] W. Thomas. Automata on infinite objects. *Handbook of theoretical computer science*, pages 165–191, 1990.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[Wol82] P. Wolper. *Synthesis of Communicating Processes from Temporal Logic Specifications*. PhD thesis, Stanford University, 1982.