On the Complexity of Verifying Concurrent Transition Systems*

David Harel[1] and Orna Kupferman[2]** and Moshe Y. Vardi[3]***

[1] Department of Applied Math. & Computer Science, The Weizmann Institute, Rehovot 76100, Israel.
Email: harel@wisdom.weizmann.ac.il
[2] EECS Department, UC Berkeley, Berkeley CA 94720-1770, U.S.A.
Email: orna@eecs.berkeley.edu
[3] Rice University, Department of Computer Science, Houston, TX 77251-1892, U.S.A.
Email: vardi@cs.rice.edu, URL: http://www.cs.rice.edu/~vardi

Abstract. In *implementation verification*, we check that an implementation is correct with respect to a specification by checking whether the behaviors of a transition system that models the program's implementation correlate with the behaviors of a transition system that models its specification. In this paper, we investigate the effect of concurrency on the complexity of implementation verification. We consider trace-based and tree-based approaches to the verification of concurrent transition systems, with and without fairness. Our results show that in almost all cases the complexity of the problem is exponentially harder than that of the sequential case. Thus, as in the model-checking verification methodology, the state-explosion problem cannot be avoided.

1 Introduction

While program verification has always been desirable but never easy, the advent of concurrent programming has made it significantly more necessary and difficult. We distinguish between two main methodologies for formal verification. The first is *temporal-logic model checking*. Here, we verify the correctness of a program with respect to a desired behavior by checking whether a state-transition graph that models the program satisfies a temporal-logic formula that specifies constraints on its behavior. The second methodology is *implementation verification*. Here, we check that an implementation is correct with respect to a specification by checking whether the behaviors of a state-transition graph that models the program's implementation correlate with the behaviors of a state-transition graph that models its specification.

The complexity of model checking is well known. For example, in the case of the temporal logics LTL and CTL, model checking can be carried out in space that is polynomial in $n \log m$, where n is the length of the formula and m is the size of the graph modeling the program [LP85, VW94, BVW94]. Keeping in mind that the formulas are usually small, it seems that model checking is easy and tractable. It suffers, however, acutely from the so-called *state-explosion problem*. In a concurrent setting, the program under consideration is typically the parallel composition of many processes, which implies that the size of the program graph is the product of the sizes of the graphs modeling the underlying processes. Accordingly, the model-checking problem for concurrent programs can be solved in space that is polynomial in nm,

* Part of this research was done in Bell Laboratories during the DIMACS special year on Logic and Algorithms.
** Supported in part by the ONR YIP award N00014-95-1-0520, by the NSF CAREER award CCR-9501708, by the NSF grant CCR-9504469, by the AFOSR contract F49620-93-1-0056, by the ARO MURI grant DAAH-04-96-1-0341, by the ARPA grant NAG2-892, and by the SRC contract 95-DC-324.036.
*** Supported in part by the NSF grant CCR-9628400.

where n is the length of the formula and m is the sum of the sizes of the graphs modeling the processes. Can we do better than this? Can we model-check a concurrent program and avoid the state-explosion problem? Unfortunately, the answer is no. Indeed, model checking of concurrent programs for LTL and CTL is PSPACE–complete even for a fixed formula [VW94, BVW94]. Hence, in the worst case we might need to traverse the exceedingly large state space introduced by the parallel composition. Coping with the state-explosion problem is one of the most important issues in computer-aided verification and is the subject of much active research (cf. [CGL93])

What about implementation verification? Is the state-explosion problem unavoidable there too? This is the subject of our work. We first describe implementation verification in more detail. Consider an implementation and a specification. Both describe 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 suggests a top-down method for design development, called *hierarchical refinement* (cf. [LS84, Kur94]): Starting with a highly abstract specification, we construct a sequence of behavior descriptions, each of which refers to its predecessor as a specification, and is thus less abstract than the predecessor. At each stage the current implementation is verified to satisfy its specification. The last description in the sequence contains no abstractions, and constitutes the final implementation.

There are several ways of defining what it means for an implementation to satisfy a specification. The two main ones are *trace-based* and *tree-based*. The former requires each computation of the implementation to correlate with some computation of the specification, and the latter requires each computation tree embodied in the implementation to correlate with some computation tree embodied in the specification. The exact notion of correct implementation then depends on how we interpret correlation. Numerous proposals for this have been made and studied in the literature [Hen85, Mil89, AL91]. In this paper we adopt a simple interpretation, taking correlation to mean equivalence with respect to the variables joint to the implementation and the specification. One justification for this is the fact that the more concrete implementation is typically defined over a wider set of variables than the more abstract specification. With this interpretation, trace-based verification corresponds to establishing *containment* [Kur94] and tree-based verification corresponds to establishing *simulation* [Mil71]. As shown by Milner [Mil80], while simulation implies containment, the opposite direction is not true.

We model concurrent programs (and hence implementations and specifications) by what we shall call *concurrent transition systems*. The basic motivation for this comes from the *statecharts* of [Har87], which can be viewed as finite automata with both concurrency and hierarchy, though for simplicity we eliminate the hierarchy here. A concurrent transition system consists of *components*, which model the program's underlying processes. (The analogous parts of a statechart are called *orthogonal components* in [Har87].) Each component is a state-transition graph. Its states correspond to the possible positions of the process it models, and each state is labeled with the events that occur, or hold, in the corresponding position. The transitions of the graph correspond to the possible steps of the process, with branches representing nondeterminism. To model the cooperation of processes during execution, the transitions are made conditional and can depend on the states of the other components. This approach to modeling concurrency, called *bounded cooperative concurrency* in [Har89, DH94], is the dominating one in research on distributed systems (cf. [Kur94]).

A concurrent transition system with a single component models a program with no concurrency, and we call it a *sequential transition system*. By [DH94], a concurrent transition system can be translated into a sequential transition system with an exponential blow up in size. Indeed, it is the size of this sequential system that is referred to in current analyses of the complexity of verification. The question we want to address here is whether the exponential blow up that hides

in these analyses can be avoided if the program to be verified is concurrent.

Before we turn to this question, let us review some known results for the implementation verification of sequential transition systems (for full details, see Section 2.3). We examine the complexity of the containment and the simulation problems in four different ways:

1. The *joint complexity* of containment and simulation. This measure considers the complexity in terms of both the implementation and the specification. The joint complexity of simulation is PTIME-complete [Mil80, BGS92], whereas that of containment is PSPACE-complete [SVW87].
2. The *implementation complexity* of containment and simulation. This measure considers the complexity in terms of the implementation, assuming the specification is fixed. Since the implementation is typically much larger than the specification, this measure is of particular interest. According to this measure, containment is easier than simulation [KV96].
3. The *joint complexity* of *fair containment* and *fair simulation*. When we consider *fair transition systems* [MP92], which enable the description of behaviors that satisfy both liveness and safety properties, containment and simulation are revised to consider only the fair computations of the implementation and the specification. The resulting problems, of fair containment and fair simulation [BBLS92, ASB$^+$94, GL94] are both PSPACE–complete [KV96].
4. The *implementation complexity* of *fair containment* and *fair simulation*. Here, the advantage of the trace-based approach reappears [KV96].

We address the question about the power of concurrency in program verification by examining the four measures when applied to concurrent transition systems. We first define containment and simulation with respect to such systems, and then consider the complexity and the implementation complexity of detecting their presence. We then turn to defining fair-containment and fair-simulation with respect to concurrent transition systems, and study their complexities too, employing *unconditional*, *weak*, and *strong* fairness (also known as *impartiality*, *justice*, and *compassion*, respectively) [LPS81, MP92].

Before saying a little more about the results themselves, we clarify what we feel are the paper's two main contributions. First, it continues the study of implementation verification in [Mil80, BGS92, KV96]. Unlike these papers, our complexity analysis addresses the state-explosion issue explicitly, by taking the size to be that of the concurrent systems themselves and not their sequential equivalents. In addition, our work continues the study of the power of bounded cooperative concurrency undertaken in [Har89, DH94, HH94, HRV90]. the results in these papers show that cooperative concurrency exhibits inherent exponential power. The power criteria considered there are succinctness of finite automata and pushdown automata, and the effect of the succinctness gap on the difficulty of reasoning about transition systems on a propositional level. In the present paper, the power criteria is the complexity of the verification problem.

Our results strengthen the observations in [Har89, DH94, KV96]. Specifically, the question of whether the exponential nature of concurrency carries over to the verification problem is answered in the affirmative. We show that verifying concurrent transition systems is exponentially harder than verifying sequential transition systems, and thus the state-explosion problem cannot be avoided. This result is robust: It is independent of the verification approach and the fairness constraint under consideration, and remains valid when we consider implementation complexity too. In particular, we show that the fair-containment and fair-simulation problems for concurrent transition systems are EXPSPACE–complete. These results join those of [KV96] in questioning the computational superiority of tree-based verification.

One exception to the inherent exponential power of cooperative concurrency is the fair-simulation problem for strongly-fair transition systems. While the implementation complexity

of the problem is PTIME–complete for sequential transition systems [KV96], we show that it is PSPACE–complete (rather than EXPTIME–complete) for concurrent transition systems. The reason for this anomaly is the fact that translating a strongly fair concurrent system into a sequential one indeed involves an exponential blow up in the number of states, but involves no such blow up in the size of the fairness condition. Evidently, it is the size of the fairness condition that is the dominant factor when reasoning about strongly-fair transition systems. This suggests that strong fairness is the preferable fairness condition to use when specifying concurrent programs. Not only is it the most expressive condition, but it also suffers less than the others from the state-explosion problem.

2 Preliminaries

2.1 Fair Concurrent Transition Systems

A fair nondeterministic transition system with bounded concurrency (*concurrent transition system*, for short) is a tuple $S = \langle O, S_1, \ldots, S_n \rangle$ consisting of a finite set O of *observable events* and n *components* $S_1, \ldots, S_n$ for some $n \geq 1$. Each component S_i is a tuple $\langle O_i, W_i, W_i^0, \delta_i, L_i, \alpha_i \rangle$, where:

- $O_i \subseteq O$ is a set of local observable events. The O_j are not necessarily pairwise disjoint; hence, observable events may be shared by several components. We require that $O = \bigcup_{j=1}^{n} O_j$.
- W_i is a finite set of states, and we require that the W_j be pairwise disjoint. Also, we let $W = \bigcup_{j=1}^{n} W_j$.
- $W_i^0 \subseteq W_i$ is the set of initial states.
- $\delta_i \subseteq W_i \times \mathcal{B}(W) \times W_i$ is a transition relation, where $\mathcal{B}(W)$ denotes the set of all Boolean propositional formulas over W.
- $L_i : W_i \rightarrow 2^{O_i}$ is a labeling function that labels each state with a set of local observable events. The intuition is that $L_i(w)$ are the events that occur, or hold, in w.
- α_i is a fairness condition. We define three types of fairness conditions shortly. We require all the α_i's to be of the same type, which we refer to as the type of S.

Since states are labeled with sets of elements from O, we refer to $\Sigma = 2^O$ as the *alphabet* of S. While each component of S has its local observable events and its own states and transitions, these transitions depend not only on the component's current state but also on the current states of the other components. Also, as we shall now see, the labels of the components are required to agree on shared observable events.

A *configuration* of S is a tuple $c = \langle w_1, w_2, \ldots, w_n, \sigma \rangle \in W_1 \times W_2 \times \cdots \times W_n \times \Sigma$, satisfying $L_i(w_i) = \sigma \cap O_i$ for all $1 \leq i \leq n$. Note that, in particular, $\sigma = \bigcup_i L_i(w_i)$. Thus, a configuration describes the current state of each of the components, as well as the set of observable events labeling these states. The requirement on σ implies that these labels are *consistent*, i.e., for any S_i and S_j, and for each $o \in O_i \cap O_j$, either $o \in L_i(w_i) \cap L_j(w_j)$ (in which case, $o \in \sigma$), or $o \notin L_i(w_i) \cup L_j(w_j)$ (in which case, $o \notin \sigma$). For a configuration $c = \langle w_1, w_2, \ldots, w_n, \sigma \rangle$, we term $\langle w_1, w_2, \ldots, w_n \rangle$ the *global state* of c, and we term σ the *label* of c, and denote it by $L(c)$. A configuration is *initial* if for all $1 \leq i \leq n$, we have $w_i \in W_i^0$. We use C to denote the set of all configurations of a given system S, and C_0 to denote the set of all its initial configurations. We also use $c[i]$ to refer to S_i's state in c.

For a propositional formula θ in $\mathcal{B}(W)$ and a global state $p = \langle w_1, w_2, \ldots, w_n \rangle$, we say that p *satisfies* θ if assigning **true** to states in p and **false** to states not in p makes θ true. For example, $s_1 \wedge (t_1 \vee t_2)$, with $s_1 \in W_1$ and $\{t_1, t_2\} \subseteq W_2$, is satisfied by every global state in which S_1 is in

state s_1 and S_2 is in either t_1 or t_2. We shall sometimes write disjunctions as sets, so that the above formula can be written $\{s_1\} \wedge \{t_1, t_2\}$. Formulas in $\mathcal{B}(W)$ that appear in transitions are called *conditions*. If θ is equivalent to **true** in the transition $\langle w, \theta, w' \rangle$, we say that it is *unconditional*.

Given two configurations $c = \langle w_1, w_2, \ldots, w_n, \sigma \rangle$ and $c' = \langle w'_1, w'_2, \ldots, w'_n, \sigma' \rangle$, we say that c' is a *successor of* c *in* S, and write $succ_S(c, c')$, if for all $1 \leq i \leq n$ there is $\langle w_i, \theta_i, w'_i \rangle \in \delta_i$ such that $\langle w_1, w_2, \ldots, w_n \rangle$ satisfies θ_i. In other words, a successor configuration is obtained by simultaneously applying to all the components a transition that is enabled in the current configuration. Note that by requiring that successors are indeed configurations, we are saying that transitions can only lead to states satisfying the consistency criterion, to the effect that they agree on the labels for shared observable events.

Given a configuration c, a *c-computation* of S is an infinite sequence $\pi = c_0, c_1, \ldots$ of configurations, such that $c_0 = c$ and for all $i \geq 0$ we have $succ_S(c_i, c_{i+1})$. A *computation* of S is a c-computation for some $c \in C_0$. The c-computation $c_0, c_1, \ldots$ *generates* the infinite *trace* $\rho \in \Sigma^\omega$, defined by $\rho = L(c_0) \cdot L(c_1) \cdots$. Sometimes we want to exclude computations of S that do not meet some fairness criteria. This is particularly essential when we model concurrent programs and want to rule out computations that do not meet certain scheduling criteria. In order to determine whether a computation π is *fair*, we refer to the sets of states that each of the components visits infinitely often along π. For each $1 \leq i \leq n$, let $Inf(\pi, i)$ denote the set of states that S_i visits infinitely often. That is, $Inf(\pi, i) = \{w \in W_i : \text{for infinitely many } j \geq 0, \text{ we have } c_j[i] = w\}$. Note that the set $Inf(\pi, i)$ considers only the states of S_i and does not refer to the global states visited along π. The way we refer to $Inf(\pi, i)$ depends in the fairness condition of S. Several types of fairness conditions are studied in the literature. We consider here three: We consider three types of fairness conditions:

- *Unconditional* fairness (or *impartiality*), where for all components S_i we have $\alpha_i \subseteq 2^{W_i}$, and π is fair iff for all $1 \leq i \leq n$ and for every set $G \in \alpha_i$, we have $Inf(\pi, i) \cap G \neq \emptyset$.
- *Weak* fairness (or *justice*), where $\alpha_i \subseteq 2^{W_i} \times 2^{W_i}$, and π is fair iff for all $1 \leq i \leq n$ and for every pair $\langle G, B \rangle \in \alpha_i$, we have that $Inf(\pi, i) \cap (W_i \setminus G) = \emptyset$ implies $Inf(\pi, i) \cap B \neq \emptyset$.
- *Strong* fairness (or *fairness*), where $\alpha_i \subseteq 2^{W_i} \times 2^{W_i}$, and π is fair iff for all $1 \leq i \leq n$ and for every pair $\langle G, B \rangle \in \alpha_i$, we have that $Inf(\pi, i) \cap G \neq \emptyset$ implies $Inf(\pi, i) \cap B \neq \emptyset$.

In addition, we consider *non-fair concurrent transition systems*; i.e., concurrent transition systems in which all the computations are fair. For simplicity, we denote components of non-fair concurrent transition system by quintuplet, leaving α_i out.

We use $\mathcal{T}(S^c)$ to denote the set of all traces generated by fair c-computations, and the *trace set* $\mathcal{T}(S)$ of S is then defined as $\bigcup_{c \in C_0} \mathcal{T}(S^c)$. In this way, each concurrent transition system S defines a subset of Σ^ω. We say that S *accepts* a trace ρ if $\rho \in \mathcal{T}(S)$. Also, we say that S is *empty* if $\mathcal{T}(S) = \emptyset$; i.e., S has no fair computation. Note that for a non-fair concurrent transition system S, the trace set $\mathcal{T}(S)$ contains all traces $\rho \in \Sigma^\omega$ for which there exists a computation π with $L(\pi) = \rho$.

The *size* of a concurrent transition system S is the sum of the sizes of its components. Symbolically, $|S| = |S_1| + \cdots + |S_n|$. Here, for a component $S_i = \langle O_i, W_i, W_i^0, \delta_i, L_i, \alpha_i \rangle$, we define $|S_i| = |O_i| + |W_i| + |\delta_i| + |L_i| + |\alpha_i|$, where $|\delta_i| = \sum_{\langle w, \theta, w' \rangle \in \delta_i} |\theta|$, $|L_i| = |O_i| \cdot |W_i|$, and $|\alpha_i|$ is the sum of the cardinalities of the sets in α_i. Clearly, S can be stored in space $O(|S|)$.

When S has a single component, we say that it is a *sequential transition system*. Note that the transition relation of a sequential transition system can be really viewed as a subset of $W \times W$, and that a configuration of a sequential transition system is simply a labeled state.

Example 1. We construct a non-fair concurrent transition system S as a binary counter; it counts up to 2^n in base 2 using n components. Given n, let $S = \langle \{bit_1, \ldots, bit_n\}, S_1, \ldots, S_n \rangle$, where $S_i = \langle \{bit_i\}, \{w_i^0, w_i^1\}, \{w_i^0\}, \delta_i, L_i \rangle$, with δ_i and L_i defined as follows:

- $\delta_i = \{\langle w_i^0, \theta_i, w_i^0 \rangle, \langle w_i^0, \neg\theta_i, w_i^1 \rangle, \langle w_i^1, \theta_i, w_i^1 \rangle, \langle w_i^1, \neg\theta_i, w_i^0 \rangle\}$, where $\theta_i = \bigvee_{j<i} w_j^0$. Note that $\theta_1 \equiv$ **false**. Thus, S_1 corresponds to the least significant bit of the counter and always alternates between w_1^0 and w_1^1. The component S_i, for $i > 1$, switches between w_i^0 and w_i^1 whenever all the S_j with $j < i$ are in their w_j^1 states. Otherwise, S_i stays in its current state.
- For every S_i, we set $L_i(w_i^0) = \emptyset$ and $L_i(w_i^1) = \{bit_i\}$.

The trace $(\emptyset \cdot \{bit_1\} \cdot \{bit_2\} \cdot \{bit_2, bit_1\} \cdot \{bit_3\} \cdot \{bit_3, bit_1\} \cdots \{bit_n, bit_{n-1}, \ldots, bit_1\})^\omega$ is the single initial trace induced by the system S.

Note that although S has n components, its size is quadratic in n. Indeed, the size of each transition relation δ_i is $O(i)$. However, we can define a slightly more sophisticated version of this system that is of size $O(n)$. Each component S_i has four states, corresponding to the possible values of both the i'th bit of the counter and the i'th carry bit. The conditions in the transitions in δ_i then refer only to the states of S_{i-1}, and are of a constant size.

2.2 Trace-Based and Tree-Based Implementations

The problems that formalize correct trace-based and tree-based implementations of a system are *containment* and *simulation*, respectively. Once we add fairness to the systems, the corresponding problems are *fair containment* and *fair simulation*. These problems are defined below with respect to two concurrent transition systems $S = \langle O, S_1, \ldots, S_n \rangle$ and $S' = \langle O', S'_1, \ldots, S'_m \rangle$ with $O \supseteq O'$, and with possibly different numbers of components. For technical convenience, we assume that $O = O'$ and that S and S' have the same type of fairness conditions. [4]

Containment and Fair Containment. The *fair-containment problem* for S and S' is to determine whether $\mathcal{T}(S) \subseteq \mathcal{T}(S')$. That is, whether every trace accepted by S is also accepted by S'. When S and S' are non-fair, we call the problem *containment*. If $\mathcal{T}(S) \subseteq \mathcal{T}(S')$, we say that S' *contains* S and we write $S \subseteq S'$.

Simulation. While containment refers only to the set of computations of S and S', simulation refers also to the branching structure of the systems. Let c and c' be configurations of S and S', respectively. A relation $H \subseteq C \times C'$ is a *simulation relation* from $\langle S, c \rangle$ to $\langle S', c' \rangle$ iff the following conditions hold [Mil71].

1. $H(c, c')$.
2. For all configurations $a \in C$ and $a' \in C'$ with $H(a, a')$, we have $L(a) = L(a')$.
3. For all $a \in C$ and $a' \in C'$ with $H(a, a')$ and for every configuration $b \in C$ such that $succ_S(a, b)$, there exists a configuration $b' \in C'$ such that $succ_{S'}(a', b')$ and $H(b, b')$.

A simulation relation H is a *simulation from S to S'* iff for every $c \in C_0$ there exists $c' \in C'_0$ such that $H(c, c')$. If there exists a simulation from S to S', we say that S *simulates* S' and we write $S \preceq S'$. Intuitively, it means that the system S' has more behaviors than the system S. In fact, every tree embodied in S is also embodied in S'. The *simulation problem* is, given S and S', to determine whether $S \preceq S'$.

[4] Our results hold also for the general cases. Taking, for each $\sigma \in 2^O$, the letter $\sigma \cap O'$ instead the letter σ, adjusts all our algorithms to the case $O \supset O'$. Also, when S and S' have different types of fairness conditions, the type of S is dominant, and the complexity of the problem is the same as in the case where both systems have fairness conditions of S's type.

Fair Simulation. Let $H \subseteq C \times C'$ be a relation over the configurations of S and S'. It is convenient to extend H to relate also computations of S and S'. For two computations $\pi = c_0, c_1, \ldots$ in S, and $\pi' = c'_0, c'_1, \ldots$ in S', we say that $H(\pi, \pi')$ holds iff $H(c_i, c'_i)$ holds for all $i \geq 0$. Let c and c' be configurations in S and S', respectively. A relation $H \subseteq C \times C'$ is a *fair-simulation relation* from $\langle S, c \rangle$ to $\langle S', c' \rangle$ iff the following conditions hold [GL94].

1. $H(c, c')$.
2. For all configurations $a \in C$ and $a' \in C'$ with $H(a, a')$, we have $L(a) = L(a')$.
3. For all configurations $a \in C$ and $a' \in C'$ with $H(a, a')$, and for every fair c-computation π in S, there exists a fair c'-computation π' in S', such that $H(\pi, \pi')$.

A fair-simulation relation H is a *fair simulation from S to S'* iff for every $c \in C_0$ there exists $c' \in C'_0$ such that $H(c, c')$. If there exists a simulation from S to S', we say that S *fairly simulates* S' and we write $S \preceq S'$. Intuitively, it means that the concurrent transition system S' has more fair behaviors than the concurrent transition system S. The *fair-simulation problem* is, given S and S', to determine whether $S \preceq S'$.

We say that two concurrent transition systems S and S' are *equivalent* if they fairly simulate each other. Thus, if $S \preceq S'$ and $S' \preceq S$. Note that equivalent systems agree on their trace sets.

Theorem 1. *Every concurrent transition system S can be translated into an equivalent sequential transition system of the same type and of size $2^{O(|S|)}$.*

Proof (sketch): Drusinsky and Harel prove the theorem with respect to automata, where the observable events are input to the machine and where equivalence is defined as agreement on the trace set [DH94]. Yet, their proof holds also for transition systems with our definition of equivalence (mutual simulation), as follows. Consider a concurrent transition system S with n components. The state space of its equivalent sequential transition system S' is the Cartesian product of the state sets of the n components (this would be $W_1 \times W_2 \times \cdots \times W_n$ in the notation used earlier). Thus, each state of S' corresponds to a configuration of S. Accordingly, the transition relation of S' coincides with the relation $succ_S$ over the configurations of S. We now need to define the fairness condition of S' so that a computation of S' is fair iff the corresponding computation of S is fair. Let α' be such that for all $1 \leq i \leq n$, every set $G \in \alpha_i$ (pair $\langle G, B \rangle \in \alpha_i$) induces the set $W_1 \times \cdots \times W_{i-1} \times G \times W_{i+1} \times \cdots \times W_n$ (the pair $\langle W_1 \times \cdots \times W_{i-1} \times G \times W_{i+1} \times \cdots \times W_n, W_1 \times \cdots \times W_{i-1} \times B \times W_{i+1} \times \cdots \times W_n \rangle$, respectively) in α'. It is easy to see that S and S' agree on their trace sets, that they simulate each other, and that the size of S' is at most exponential in that of S. □

In the rest of this paper we examine the traced-based and the tree-based approaches from a *complexity-theoretic* point of view. We consider and compare the complexity of the four problems. 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., the complexity of checking whether $S \subseteq S'$ and $S \preceq S'$, in terms of the size of S, assuming S' is fixed.

2.3 Verification of Sequential Transition Systems

We mention here some known results on the verification of sequential transition systems.

Theorem 2. [SVW87, VW94, KV96] *The containment problem for sequential transition systems is PSPACE–complete; The implementation complexity of the problem is NLOGSPACE–complete.*

Theorem 3. [Mil80, BGS92, KV96] *The simulation problem for sequential transition systems is PTIME–complete; The implementation complexity of the problem is PTIME–complete.*

Theorem 4. [SVW87, VW94, KV96] *The fair-containment problem for sequential transition systems is PSPACE–complete; The implementation complexity of the problem is NLOGSPACE–complete for unconditionally-fair and weakly-fair systems, and is PTIME–complete for strongly-fair systems.*

Theorem 5. [KV96] *The fair-simulation problem for sequential transition systems is PSPACE–complete; The implementation complexity of the problem is PTIME–complete.*

It follows that, when comparing the trace-based and the tree-based approaches to verification from a complexity-theoretic point to view, there is no clear advantageous approach. While the joint complexity of simulation is lower than that of containment, it is containment that has lower implementation complexity. In addition, fair containment and fair simulation have the same joint complexity, with fair containment having lower implementation complexity for the case of unconditionally-fair and weakly-fair transition systems.

3 The Containment Problem

Theorem 6. *The containment problem for concurrent transition systems is EXPSPACE–complete.*

Proof (sketch): Membership in EXPSPACE follows from Theorems 1 and 2.

To prove hardness, we carry out a reduction from deterministic exponential-space-bounded Turing machines. Given a Turing machine T of exponential space complexity $s(n)$, we denote by Σ an alphabet for encoding T (the alphabet Σ and the encoding are defined later). We then construct a transition system S_T over the alphabet $\Sigma \cup \{\$\}$, for some $\$ \notin \Sigma$, such that (i) the size of S_T is linear in $|T|$ and in $\log s(n)$, and (ii) $\Sigma^\omega + (\Sigma^* \cdot \$^\omega) \subseteq \mathcal{T}(S_T)$ 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. Typically, the transition system S_T accepts all traces in Σ^ω, and accepts a trace $w \cdot \$^\omega \in \Sigma^* \cdot \$^\omega$ if either (i) w is not an encoding of a prefix of a legal computation of T over the empty tape, (ii) w is an encoding of a prefix of a legal computation of T over the empty tape, but, within this prefix, the computation still has not reached a final state, or (iii) w is an encoding of a prefix of a legal, but rejecting, computation of T over the empty tape. Thus, S_T rejects a trace $w \cdot \$^\omega$ iff w encodes a prefix of a legal accepting computation of T over the empty tape and the computation has already reached a final state. Hence, S_T accepts all traces in $\Sigma^\omega + \Sigma^* \cdot \$^\omega$ iff T does not accept the empty tape.

In order to check whether a given trace w encodes a computation of T, the system S_T checks that each two successive configurations encoded in w are compatible with T's transition relation. For that, S_T needs to relate positions in w that are $s(n) + 1$ far from each other. Accordingly, one component of S_T, the one that performs the "compatibility check", cooperates with $\log s(n)$ other components, whose only task is to perform this count (as described in Example 1).

Now, we construct S to be a concurrent transition system that generates the language $\Sigma^\omega + (\Sigma^* \cdot \$^\omega)$. In fact, S can be easily taken to be a sequential transition system with $|\Sigma| + 1$ states. It follows that T does not accept the empty tape iff $S \subseteq S_T$. □

The reduction we present in the proof of Theorem 6 considers a simple implementation and an elaborated specification. We now show that the specification is indeed the dominant factor of

the containment problem. Fixing it, the problem becomes significantly easier. Still, traversing the exponentially big state space of the implementation cannot be avoided.

Theorem 7. *The implementation complexity of containment for concurrent transition systems is PSPACE–complete.*

Proof (sketch): Membership in PSPACE follows from Theorems 1 and 2. For the lower-bound, we prove that the emptiness problem for concurrent transition systems is already PSPACE-hard. For that, we carry out a reduction from deterministic polynomial-space-bounded Turing machines. We show that given a Turing machine T of polynomial space complexity $s(n)$, it is possible to build, using a logarithmic amount of space, a concurrent transition system S_T of size $O(s(n))$ such that S_T is empty if and only if T does not accept the empty tape.

Let $T = \langle \Gamma, Q, \mapsto, q_0, F_{rej} \rangle$, where Γ is the alphabet, Q is the set of states, and $\mapsto : (Q \times \Gamma) \to (Q \times \Gamma \times \{L, R\})$ is the transition function. We write $(q, a) \mapsto (q', b, \Delta)$ for $\mapsto (q, a) = (q', b, \Delta)$, with the meaning that when in state q and reading a in the current tape cell, T moves to state q', writes b in the current tape cell and moves its head one cell to the left or right, depending on Δ. Finally, q_0 is T's initial state, and $F_{rej} \subseteq Q$ is the set of final rejecting states. The system S_T has $s(n)$ components, one for each tape cell that is used. For all $1 \leq i \leq s(n)$, the component $S_i = \langle \emptyset, W_i, W_i^0, \delta_i, L_i \rangle$ has all states labeled with $\emptyset$ and is defined as follows:

- $W_i = (((Q \setminus F_{rej}) \times \Gamma) \cup \Gamma) \times \{i\}$. A state of the form (q, a, i) indicates that T is in state q and its head is at the i'th cell, whose contents is a. A state of the form (a, i) indicates that the contents of the i'th cell is a but the head is not at that cell.
- A transition $(q, a) \mapsto (q', b, \Delta)$ of T, with $q' \notin F_{rej}$, induces the following transitions in δ_i.
  - An unconditional transition from (q, a, i) to (b, i); i.e., $\langle (q, a, i), \mathbf{true}, (b, i) \rangle \in \delta_i$. This transition corresponds to the head moving from cell i to cell $i + 1$ or $i - 1$.
  - A transition from every $(z, i) \in \Gamma \times \{i\}$ to (q', z, i), with a condition defined as follows: If $\Delta = R$, then the current state of S_{i-1} is $(q, a, i - 1)$; i.e., if $\Delta = R$, then $\langle (z, i), \{(q, a, i - 1)\}, (q', z, i) \rangle \in \delta_i$. Dually, if $\Delta = L$, then the current state of S_{i+1} is $(q, a, i + 1)$; i.e., if $\Delta = L$, then $\langle (z, i), \{(q, a, i + 1)\}, (q', z, i) \rangle \in \delta_i$. This transition corresponds to the head moving from cell number $i + 1$ or $i - 1$ to cell number i.

  In addition, we have a transition in δ_i from every $(z, i) \in \Gamma \times \{i\}$ to (z, i), with a condition stating that the head is not moving now to the i'th cell. Let W_R^i be the set of states $(q, a, i-1)$ in W_{i-1} such that $(q, a) \mapsto (q', b, R)$ is a transition of T for some q' and b. In a dual way, let W_L^i be the set of states $(q, a, i + 1)$ in W_{i+1} such that $(q, a) \mapsto (q', b, L)$ is a transition of T for some q' and b. Then, $\langle (z, i), \neg(W_R^i \cup W_L^i), (z, i) \rangle \in \delta_i$.
- W_i^0, the set of initial states of S_i, is a singleton that corresponds to the initial contents of the i'th cell. Thus, $W_i^0 = \{(q_0, \beta, 1)\}$ for $i = 1$, and $W_0 = \{(\beta, i)\}$ for $1 < i \leq s(n)$.

Since T is deterministic, the system S_T proceeds in a deterministic fashion, which corresponds to the single computation of T on the empty tape. To see this, observe that each reachable configuration of S_T has exactly one component S_i which is in a state in $Q \times \Gamma \times \{i\}$. Thus, each reachable configuration of S_T corresponds to a configuration of T. Also, a transition in S_T from configuration c to c' corresponds to the single possible transition of T from its configuration corresponding to c to the one corresponding to c'. Since states in $F_{rej} \times \Gamma \times \{i\}$ are not reachable in W_i, the system S_T has no corresponding move whenever T moves to a final rejecting state. So, if T rejects the empty tape, then S_T is empty. In addition, if T accepts the empty tape, then, as T loops in its final state, S_T accepts the trace $\emptyset^\omega$. Hence, T rejects the empty tape iff S_T is empty. □

In view of the known PSPACE lower bound for emptiness in communicating finite state machines [Koz77], our PSPACE lower bound here is not surprising. Note, however, that the bound in [Koz77] does not directly imply our bound here, since concurrent transition systems generate *infinite* traces.

4 The Simulation Problem

Establishing simulation involves only local checks. One could hope that locality circumvents the state-explosion problem. We show here that while locality neutralizes the dominance of the specification, an exponential blow-up in the implementation cannot be avoided. Moreover, the branching nature of simulation can be used to encode alternation, making the implementation complexity of simulation higher than that of containment.

Theorem 8. *The simulation problem for concurrent transition systems is EXPTIME–complete.*

Proof (sketch): Membership in EXPTIME follows from Theorems 1 and 3. To prove hardness in EXPTIME, we carry out a reduction from alternating linear-space-bounded Turing machines, proved to be EXPTIME-hard in [CKS81]. Similarly to the construction in the proof of Theorem 7, we show that there exists a fixed concurrent transition system S', such that, given an alternating Turing machine T of space complexity $s(n)$, it is possible to build, using a logarithmic amount of space, a concurrent transition system S_T of size $O(s(n))$ such that $S \preceq S'$ if and only if T accepts the empty tape.

Consider an alternating Turing machine $T = \langle \Gamma, Q_u, Q_e, \mapsto, q_0, F_{acc}, F_{rej} \rangle$, where the four sets of states, Q_u, Q_e, F_{acc}, and F_{rej} are disjoint, and contain the universal, the existential, the accepting, and the rejecting states, respectively. We denote their union (the set of all states) by Q. Our model of alternation prescribes that $\mapsto \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R, H\}$ has a binary branching degree, is universal in its even-numbered steps, and is existential in its odd-numbered ones (H means that the head of T stays on the same cell). In particular, $q_0 \in Q_e$. When a universal or existential state of T branches into two states, we distinguish between the left and the right branches. Accordingly, we use $(q, a) \mapsto \langle (q_l, b_l, \Delta_l), (q_r, b_r, \Delta_r) \rangle$ to indicate that when T is in state $q \in Q_u \cup Q_e$ reading input symbol a, it branches to the left with (q_l, b_l, Δ_l) and to the right with (q_r, b_r, Δ_r). (Note that the directions left and right here have nothing to do with the movement direction of the head; these are determined by Δ_l and Δ_r.) We term q_l the $\swarrow$*-child* of q, and q_r its $\searrow$*-child*. Finally, we assume that once T reaches a final state, it loops there forever in a deterministic fashion. Accordingly, we use $(q, a) \mapsto (q, a, H)$ to indicate that when T is in state $q \in F_{acc} \cup F_{rej}$ reading symbol a, it stays in the same configuration.

The possible computations of T on w induce an AND-OR graph, whose nodes are T's configurations. We say that a node *corresponds to* state q if T's state in the node's configuration is q. With each node in the graph we associate an *acceptance value* in $\{0, 1\}$ as follows. Nodes that correspond to states in F_{acc} (respectively, F_{rej}) have acceptance value 1 (respectively, 0). The acceptance value of an AND-node (which corresponds to a universal state) is the minimum of the acceptance values of its two children, and that of an OR-node (which corresponds to an existential state) is the maximum of the acceptance values of its children.

We now construct the fixed transition system S'. The intention is for S' to embody all possible AND-OR graphs that may be induced by accepting computations of all alternating Turing machines (using the model of alternation just described). The system S' has a single component (thus, it is really a sequential transition system), whose 20 states "model" states of such machines as follows: Eight states model the Turing machine's universal states. Each of these

states matches an entry in the truth table of the operator AND, adorned with a direction, either $\swarrow$ or $\searrow$, and a flag $\wedge$ that indicates that this is a universal state. Thus, the *universal internal states* of S' are $\langle \wedge 000 \swarrow \rangle$, $\langle \wedge 010 \swarrow \rangle$, $\langle \wedge 100 \swarrow \rangle$, $\langle \wedge 111 \swarrow \rangle$, $\langle \wedge 000 \searrow \rangle$, $\langle \wedge 010 \searrow \rangle$, $\langle \wedge 100 \searrow \rangle$, and $\langle \wedge 111 \searrow \rangle$. Eight states model the Turing machine's existential states. These match the entries of the truth table of OR, adorned with a direction and a flag $\vee$. Thus, the *existential internal states* of S' are $\langle \vee 000 \swarrow \rangle$, $\langle \vee 011 \swarrow \rangle$, $\langle \vee 101 \swarrow \rangle$, $\langle \vee 111 \swarrow \rangle$, $\langle \vee 000 \searrow \rangle$, $\langle \vee 011 \searrow \rangle$, $\langle \vee 101 \searrow \rangle$, and $\langle \vee 111 \searrow \rangle$. Finally, four states model the Turing machine's final states. Each of these is a Boolean value, adorned with a direction. Thus, the *final states* of S' are $\langle 0 \swarrow \rangle$, $\langle 1 \swarrow \rangle$, $\langle 0 \searrow \rangle$, and $\langle 1 \searrow \rangle$.

The intuition is that an internal state $\langle *, l, r, val, d \rangle$ corresponds to a state of the Turing machine with the following properties: Its left child has acceptance value l, it right child has acceptance value r, its own acceptance value is, therefore, val, and it can be only a d-child of other states. Similarly, a final state $\langle val, d \rangle$ corresponds to a final state of the Turing machine with acceptance value val that can be only a d-child of other states.

Accordingly, the transitions in S' from an internal state $\langle *, l, r, val, d \rangle$ cover all the possible ways that l and r can be acceptance values of the left and right children, respectively. Thus, we have transitions from $\langle *, l, r, val, d \rangle$ to another state $w = \langle *', l', r', val', d' \rangle$ iff w is an internal state of the opposite type (i.e., either $* = \wedge$ and $*' = \vee$, or vice versa) or $w = \langle val', d' \rangle$ is a final state, and, in both cases, either $val' = l$ and $d' = \swarrow$, or $val' = r$ and $d' = \searrow$. For example, the internal state $\langle \wedge 100 \swarrow \rangle$ has transitions to states $\langle \vee 011 \swarrow \rangle$, $\langle \vee 101 \swarrow \rangle$, $\langle \vee 111 \swarrow \rangle$, $\langle \vee 000 \searrow \rangle$, $\langle 1 \swarrow \rangle$, and $\langle 0 \searrow \rangle$. It has transitions from all states $\langle \vee, l, r, val, d \rangle$ with $l = 0$. In addition, the final states have self loops.

The set of observable events of S' is $\{\swarrow, \searrow, 0, 1\}$. We label an internal state by $\swarrow$ or $\searrow$ according to its direction element. For example, the node $\langle \wedge 100 \swarrow \rangle$ is labeled $\{\swarrow\}$. We label a final state by its value and direction. For example, the node $\langle 1 \searrow \rangle$ is labeled $\{1, \searrow\}$. We define the initial states of S' to be the internal existential states with $val = 1$. This completes the definition of S'. Clearly, it's size is fixed.

Given a particular alternating Turing machine T, we now define the system S_T such that $S_T \leq S'$ iff T accepts the empty tape. In general, the construction of S_T is similar to that in the proof of Theorem 7. The main difference is that while T there was deterministic, T here is alternating, and it branches in each of its transitions. Therefore, moving from configuration to configuration we must take extra care to ensure that either all components move according to the left branch, or all components move according to the right branch.

Let $T = \langle \Gamma, Q_u, Q_e, \mapsto, q_0, F_{acc}, F_{rej} \rangle$ be an $s(n)$-space-bounded alternating Turing machine as described above. The concurrent transition system S_T has $s(n)$ components, one for each tape cell. For each $1 \leq i \leq s(n)$, the component S_i is defined as follows:

- $O_i = \{\swarrow, \searrow, 0, 1\}$. Thus, $O_i = O$, and the sets of observable events are thus independent of i. As we shall see below, the events $\swarrow$ and $\searrow$ guarantee that all components move according to the same branch.
- $W_i = ((Q \times \Gamma) \cup \Gamma) \times \{\swarrow, \searrow\} \times \{i\}$. A state of the form (q, a, d, i) is called a *head-content state*; it indicates that T is in state q, the head is at cell i, whose content is a, and q was reached by taking a d branch. A state of the form (a, d, i) is called a *content state*; it indicates that the content of cell i is a, the head is not at cell i, and if the previous state of S_i was a head-content state then the current state has become a content state as a result of a taking a d branch. For both types of states, we call d the *direction element*. The direction element of a content state is determined once there is a transition from some head-content state to it. The direction element is guaranteed to maintain the directionality of the branch taken in this transition only for the next configuration. Later, this direction element may be changed.

• S_T's transitions are induced by the transitions of T as follows (in the following description we ignore the borderline cases of $i = 1$ or $i = s(n)$, which are essentially the same but require a little more attention).
- Each transition $(q, a) \mapsto \langle (q_l, b_l, \Delta_l), (q_r, b_r, \Delta_r) \rangle$ of T induces the following transitions in δ_i:

1. Unconditional transitions that correspond to the head moving from cell i to cell $i+1$ or $i-1$. These transitions also determine the direction element of the new state of S_i. For $d \in \{\swarrow, \searrow\}$, we have $\langle (q, a, d, i), \textbf{true}, (b_l, \swarrow, i) \rangle \in \delta_i$ and $\langle (q, a, d, i), \textbf{true}, (b_r, \searrow, i) \rangle \in \delta_i$.
2. Transitions that correspond to the head moving from cell $i+1$ or $i-1$ to cell i, as a result of taking a left branch. This includes a transition from every state $(z, d, i) \in \Gamma \times \{\swarrow, \searrow\} \times \{i\}$ to the state $(q_l, z, \swarrow, i)$, with the following conditions: If $\Delta_l = R$, then the current state of S_{i-1} must be $(q, a, d', i-1)$ for some $d' \in \{\swarrow, \searrow\}$; i.e., if $\Delta_l = R$, then $\langle (z, d, i), \{(q, a, \swarrow, i-1), (q, a, \searrow, i-1)\}, (q_l, z, \swarrow, i) \rangle \in \delta_i$. Dually, if $\Delta_l = L$, then $\langle (z, d, i), \{(q, a, \swarrow, i+1), (q, a, \searrow, i+1)\}, (q_l, z, \swarrow, i) \rangle \in \delta_i$.
3. Dual transitions that correspond to the head moving from cell $i+1$ or $i-1$ to cell i, as a result of taking a right branch.

- For each transition $(q, a) \mapsto (q, a, H)$ of T, and for all $d \in \{\swarrow, \searrow\}$, we have an unconditional transition that corresponds to looping in the final state q; i.e., $\langle (q, a, d, i), \textbf{true}, (q, a, d, i) \rangle \in \delta_i$.
- In addition, we have transitions that correspond to "passive" cells; that is, cells to which or from which the head does not move. We allow these cells to change their direction elements, so they can adjust themselves to the new configurations. This includes transitions from each state $(z, d, i) \in \Gamma \times \{\swarrow, \searrow\} \times \{i\}$ to states (z, d', i) for $d' \in \{\swarrow, \searrow\}$, with the condition that if a d' branch is currently taken, the head is not moving to cell i.
• The set of initial states of S_i is a singleton that corresponds to the initial content of cell i with (the arbitrarily chosen) direction element $\swarrow$. Thus, this set will be $\{(q_0, \beta, \swarrow, 1)\}$ for $i = 1$, and $\{(\beta, \swarrow, i)\}$ for $1 < i \leq s(n)$.
• The labeling function L_i is defined as follows:

– $\swarrow \in L_i(w)$ iff $w \in ((Q \times \Gamma) \cup \Gamma) \times \{\swarrow\} \times \{i\}$.
– $\searrow \in L_i(w)$ iff $w \in ((Q \times \Gamma) \cup \Gamma) \times \{\searrow\} \times \{i\}$.
– $0 \in L_i(w)$ iff $w \in F_{rej} \times \Gamma \times \{\swarrow, \searrow\} \times \{i\}$.
– $1 \in L_i(w)$ iff $w \in F_{acc} \times \Gamma \times \{\swarrow, \searrow\} \times \{i\}$.

We claim that the unwinding of the system S_T corresponds to the AND-OR graph induced by the possible computations of T on the empty tape and thus, T accepts the empty tape iff $S_T \preceq S'$. To see this, observe that each reachable configuration of S_T has exactly one component S_i in a head-content state. All other components are in content states. It is true that often two components can move into head-content states, but then, by the definition of δ_i, they will have different direction elements, which implies, by the definition of L_i, that there will be disagreement on the labeling of $\swarrow$ and $\searrow$. Thus, each reachable configuration of S_T corresponds to a legal configuration of T. Also, each configuration of S_T either corresponds to a universal or existential state of T (in which case it has exactly two possible successors, one for each possible branch) or corresponds to a final state of T (in which case it has one possible successor and is labeled by either 0 or 1). □

Theorem 9. *The implementation complexity of simulation for concurrent transition systems is EXPTIME–complete.*

Proof: Membership in EXPTIME follows from Theorem 8. Since the transition system S' used there is fixed, the proof of Theorems 8 provides an EXPTIME lower bound also for the implementation complexity of the simulation problem. □

5 The Fair-Containment and the Fair-Simulation Problems

So far, we saw that when we consider non-fair transition systems, verification of concurrent transition systems is exponentially harder than verification of sequential transition systems. We now turn to consider fair transition systems.

Theorem 10. *The fair-containment problem for concurrent transition systems is EXPSPACE–complete.*

Proof: Membership in EXPSPACE follows from Theorems 1 and 4. Hardness in EXPSPACE follows from Theorem 6. □

Theorem 10 shows that, as in the case of sequential transition systems, the trace-based approach to verification extends to fair systems at no cost. Indeed, the complexities of containment and fair containment coincide. An exception to this phenomenon are strongly fair transition systems. By Theorem 4, the implementation complexity of fair containment for strongly-fair systems is higher than the implementation complexity of containment. We now show that strongly-fair transition systems are exceptional also in their concurrent behavior: The implementation complexity of fair containment for concurrent strongly-fair systems is not exponentially harder than that of sequential strongly-fair systems.

Theorem 11. *The implementation complexity of the fair-containment problem for concurrent transition systems is PSPACE–complete.*

Proof (sketch): Hardness in PSPACE follows from Theorem 7. For unconditionally-fair and weakly-fair concurrent transition systems, membership in PSPACE follows from Theorems 1 and 4. For strongly-fair systems, a straightforward application of Theorems 1 and 4 results in an algorithm with exponential running time and space. To get the PSPACE bound, we suggest the following algorithm. Let S and S' be strongly fair concurrent transition systems, and let D be a strongly-fair sequential transition system equivalent to S. For each component S_i of S, let k_i and m_i denote the number of states and the number of pairs in the fairness condition of S_i, respectively. Assume that S has n components. Then, following the construction described in the proof of Theorem 1, the system D has $k = k_1 \cdot k_2 \cdots k_n$ states and $m = m_1 + m_2 + \cdots + m_k$ pairs in its fairness condition. By [KV96], we can translate D to an unconditionally-fair sequential transition system U with $k \cdot 2^{O(m)}$ states. Thus, the size of U is exponential in the size of S. We can also translate S' to an unconditionally-fair sequential transition system U' (this also involves an exponential blow up, which, as S' is fixed, is irrelevant to our proof). By Theorem 7, checking the containment of U in U' can be done nondeterministically in space logarithmic in U, thus polynomial in S, and we are done. □

Theorem 12. *The fair-simulation problem for concurrent transition systems is EXPSPACE–complete.*

Proof (sketch): Membership in EXPSPACE follows from Theorems 5 and 1. To prove hardness in EXPSPACE we do a reduction from exponential-space-bounded Turing machines. Our reduction is similar to the reduction described in [KV96] for proving a lower bound to the

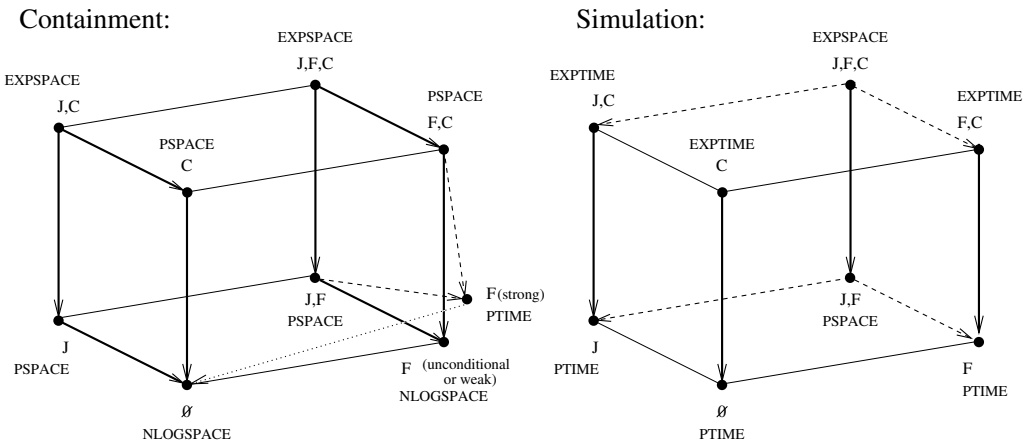
fair-simulation problem for sequential transition systems. The only change is that while there the Turing machines are polynomial-space bounded, yielding a PSPACE lower bound, here the machines are exponential-space bounded, yielding an EXPSPACE lower bound. Using bounded concurrency, we can handle the exponential size of the tape by n components that count to 2^n. The details of the reduction are given in the full paper. □

Theorem 13. *The implementation complexity of the fair-simulation problem for concurrent transition systems is EXPTIME–complete.*

Proof: Membership in EXPTIME follows from Theorems 5 and 1. Hardness in EXPTIME follows from Theorem 9. □

6 Discussion

Our results are illustrated by the cube figures below, in the style of [Har89, DH94]. All the complexities denote tight bounds. We use J to denote joint complexity (and its omission to denote implementation complexity), F to denote fair transition systems (and its omission to denote non-fair ones), and C to denote concurrent transition systems (and its omission to denote sequential ones). A bold arrow represents an exponential gap between the complexity classes, a dashed arrow represents a transition from a certain space-complexity class to the same time-complexity class, and a dotted line represents a transition from a certain time-complexity class to the space-complexity class it subsumes.



This paper considered the upper planes of the boxes. The vertical bold arrows illustrate the state-explosion problem, which is unavoidable. The protruding vertex on the lower level of the containment cube illustrates the anomaly of the strong fairness condition.

How robust are our results? Examining our lower-bound proofs, one can observe that they employ only a very humble kind of cooperation between the components. Indeed, in all the reductions, the conditions used in the transitions of a certain component S_i refer only to states of the components S_{i-1} and S_{i+1}. This suggests that a very weak, and local, model of concurrency is sufficient in order to cause the state-explosion problem. In particular, our results hold for the concurrency models presented in CSP and CCS.

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 ICALP*, Jerusalem, 1994.

[BBLS92] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Proc. 4th CAV*, LNCS 663, Montreal, 1992.

[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 *Proc. 6th CAV*, LNCS 818, pages 142–155, Stanford, 1994.

[CGL93] E.M. Clarke and O. Grumberg and D. Long. Verification tools for finite-state concurrent systems. In *Decade of Concurrency – Reflections and Perspectives*, LNCS 803, pages 124–175, 1993.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28(1):114–133, January 1981.

[DH94] D. Drusinsky and D. Harel. On the power of bounded concurrency I: Finite automata. *Journal of the ACM*, 41(3):517–539, 1994.

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

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comp. Prog.*, 8:231–274, 1987.

[Har89] D. Harel. A thesis for bounded concurrency. In *Proc. 14th MFOCS*, LNCS 379, pages 35–48, New York, 1989.

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

[HH94] T. Hirst and D. Harel. On the power of bounded concurrency II: Pushdown automata. *Journal of the ACM*, 41(3):540–554, 1994.

[HRV90] D. Harel, R. Rosner, and M.Y. Vardi. On the power of bounded concurrency iii: Reasoning about programs. In *Proc. 5th LICS*, Philadelphia, 1990.

[Koz77] D. Kozen. Lower bounds for natural proof systems. In *Proc. 18th FOCS*, pages 254–266, 1977.

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

[KV96] O. Kupferman and M.Y. Vardi. Verification of fair transition systems. In *Proc. 8th CAV*, LNCS 1102, pages 372–382. Rutgers, 1996.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th POPL*, pages 97–107, New Orleans, 1985.

[LPS81] D. Lehman, A. Pnueli, and J. Stavi. Impartiality, justice, and fairness – the ethics of concurrent termination. In *Proc. 8th ICALP*, LNCS 115, pages 264–277. 1981.

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

[Mil71] R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd IJCAI*, pages 481–489, 1971.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, LNCS 92, Springer Verlag, Berlin, 1980.

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

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

[SVW87] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

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