Model Checking of Hierarchical State Machines

RAJEEV ALUR
University of Pennsylvania and Bell Laboratories
and
MIHALIS YANNAKAKIS
Bell Laboratories

Model checking is emerging as a practical tool for detecting logical errors in early stages of system design. We investigate the model checking of sequential hierarchical (nested) systems, i.e., finite-state machines whose states themselves can be other machines. This nesting ability is common in various software design methodologies and is available in several commercial modeling tools. The straightforward way to analyze a hierarchical machine is to flatten it (thus, incurring an exponential blow up) and apply a model checking tool on the resulting ordinary FSM. We show that this flattening can be avoided. We develop algorithms for verifying linear-time requirements whose complexity is polynomial in the size of the hierarchical machine. We address also the verification of branching-time requirements and provide efficient algorithms and matching lower bounds.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods; state diagrams*; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods; model checking*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*mechanical verification; specification techniques*

General Terms: Algorithms, Design, Theory, Verification

Additional Key Words and Phrases: Hierarchical state machines, model checking, temporal logic, statecharts

1. INTRODUCTION

Finite-state machines (FSMs) are widely used in the modeling of systems in various areas. Descriptions using FSMs are useful to represent the flow of control (as

A preliminary version of this paper appeared in the *Proceedings of the Sixth ACM Symposium on Foundations of Software Engineering* (FSE), pp. 175–188, 1998. The research of the first author was supported in part by NSF CAREER award CCR97-34115, NSF grant CCR99-70925, SRC contract 99-TJ-688, Alfred P. Sloan Faculty Fellowship, and by DARPA/ITO Mobies program.
Authors' addresses: R. Alur, Department of Computer and Information Science, 200 South 33rd Street, University of Pennsylvania, Philadelphia, PA 19104; email: alur@cis.upenn.edu; URL: `http://www.cis.upenn.edu/~alur`. M. Yannakakis, Bell Labs, Lucent Technologies, 600 Mountain Avenue, Murray Hill, NJ 07974; email: mihalis@research.bell-labs.com; URL: `http://cm.bell-labs.com/who/mihalis/`.

opposed to data manipulation) and are amenable to formal analysis such as model checking. In the simplest setting, an FSM consists of a labeled graph whose vertices correspond to system states and edges correspond to system transitions. In practice, to describe complex systems using FSMs, several extensions are useful. We focus on *hierarchical FSMs* (or, *nested FSMs*) in which vertices of an FSM can be ordinary states or *superstates* which are FSMs themselves.

The notion of hierarchical FSMs was popularized by the introduction of Statecharts [Harel 1987], and exists in many related specification formalisms such as Modecharts [Jahanian and Mok 1987] and RSML [Leveson et al. 1994]. It is a central component of various object-oriented software development methodologies developed in recent years, such as OMT [Rumabaugh et al. 1991], ROOM [Selic et al. 1994], and the Unified Modeling Language (UML [Booch et al. 1997]). Hierarchical modeling is commonly available also in commercial computer-aided software engineering tools such as Statemate (by i-Logix), ObjecTime Developer (by ObjecTime), and RationalRose (by Rational).

The nesting capability is useful also in formalisms and tools for the requirements and testing phases of the software development cycle. On the requirements side, it is used to specify scenarios (or *use cases* [Jacobson 1992]) in a structured manner. For instance, the new ITU standard Z.120 (MSC'96) for message sequence charts [Rudolph et al. 1996] formalizes scenarios of distributed systems in terms of hierarchical graphs built from basic MSCs. FSMs are also used to model systems for the purpose of test generation, and again the nesting capability is useful to model large systems. For example, Teradyne's commercial tool TestMaster [Apfelbaum 1995] is based on a hierarchical FSM model, and so is an internal Lucent test tool developed over many years for the testing of a large enterprise switch. Although these models are primarily developed for test generation, they can be used also for formal analysis. This is useful for systems with informal and incomplete requirements and design documentation, as is often the case, and especially for software that was developed and evolved over a long period of time, when the test models are updated for continued regression testing as the system evolves.

As a simple example of a hierarchical FSM, consider a specification of a digital clock. The top-level machine consists of a cycle though 24 superstates, one per hour of the day. Each such state, in turn, is a hierarchical state machine consisting of a cycle containing 60 superstates counting minutes, each of which, in turn, is an (ordinary) state machine consisting of a cycle counting seconds. As illustrated by this example, hierarchical state machines have two descriptive advantages over ordinary FSMs. First, superstates offer a convenient structuring mechanism that allows us to specify systems in a stepwise refinement manner, and to view them at different levels of granularity. Such structuring is particularly essential for specifying large FSMs via a graphical interface. Second, by allowing sharing of component FSMs (for instance, the 24 superstates of the top-level FSM of digital clock are mapped to the *same* hierarchical FSM corresponding to an hour), we need to specify components only once and then can reuse them in different contexts, leading to modularity and succinct system representations. In fact, as shown in a recent paper [Alur et al. 1999], there is an *exponential* gap between ordinary and hierarchical FSMs as generators of regular languages.

In this paper, we consider algorithms for model checking when the description is given as a hierarchical state machine. Model checking is emerging as a practical method for automated debugging of complex reactive systems such as embedded controllers and network protocols (see [Clarke and Kurshan 1996] and [Clarke and Wing 1996] for surveys). Commercial tools for verification of hardware systems have appeared in the market in the last two years (e.g. FormalCheck, marketed originally by Lucent, and now by Cadence). On the software side, model checkers such as Spin [Holzmann 1997] have been shown to be useful in the design and analysis of software in telecommunication and other areas. In model checking, a high-level description of a system is compared against a logical correctness requirement to discover inconsistencies [Clarke and Emerson 1981; Queille and Sifakis 1982]. Given a hierarchical FSM, one can obtain an ordinary FSM by flattening it, that is, by recursively substituting each superstate with its associated FSM. Such a flattening, however, can cause a blow-up, particularly when there is a lot of sharing. For instance, the hierarchical description of the digital clock has $24+60+60=144$ vertices, while the flattened description has $24*60*60=86,400$ vertices.[1] Thus, if we first flatten the machine, and then employ the existing model-checking algorithms, the worst-case complexity would be exponential in the original description of the structure. Our results establish that such a flattening is unnecessary by providing polynomial-time algorithms.

In this paper we consider only *sequential* hierarchical machines. To capture the full range of modeling features of design languages such as Statecharts, we would have to consider two additional, orthogonal extensions of FSMs: (1) multiple FSMs operating in parallel and communicating with each other, and (2) extended FSMs whose transitions involve reading and writing of variables. The impact of both these extensions on the analysis problems has been well understood: both the extensions, by themselves, cost an exponential, leading to the so-called *state-explosion* problem. By considering sequential hierarchical machines, we can focus on the impact of hierarchy on the analysis problems. It should be noted that the models used in testing and hierarchical MSCs are sequential, and the model we use is representative of such models.

Our first result concerns the invariant verification problem, that is, the problem of establishing that all reachable states are included within the region of states satisfying the specified invariant. Invariant verification is the most common model checking problem in practice, and can model safety requirements such as mutual exclusion and absence of deadlocks. We show, that even though some FSM may appear repeatedly in different contexts, it needs to be searched just once. We give a depth-first search algorithm that performs the reachability analysis with time complexity linear in the size of the hierarchical structure. While reachability is in Nlogspace for ordinary FSMs, we establish that reachability problem for hierarchical FSMs is P-complete.

Our second verification problem concerns verification of linear-time requirements [Pnueli 1977; Vardi and Wolper 1986] such as eventual reception. The commonly used formalisms for specifying requirements of system behaviors are automata and

[1] Alternatively, we can model the system as a collection of three communicating FSMs, one corresponding to the hours, one for the minutes, and one for the seconds. Analysis, then, would require constructing the product of these three machines, leading to the same blow-up.

linear temporal logic. In the automata-theoretic formulation, we are given a hierarchical FSM K and a Büchi automaton A that accepts undesirable behaviors, and we wish to check whether or not the languages of K and A have a nonempty intersection. We show that this problem can be solved in time $O(|K| \cdot |A|^3)$ (if K were an ordinary FSM, this complexity would be $O(|K| \cdot |A|)$). When the linear-time specification is given by a formula φ of (propositional) linear temporal logic (LTL), using the known translations from LTL to Büchi automata [Vardi and Wolper 1986], we get an algorithm for LTL model checking with time complexity $O(|K| \cdot 8^{|\varphi|})$. We note that usually the formulas ϕ and automata A that specify correctness properties are very small (few temporal operators or states), while the system model K is very large.

Our third verification problem concerns branching-time requirements specified in the logic CTL [Clarke and Emerson 1981; Queille and Sifakis 1982]. The logic CTL can express both existential and universal path properties. For model checking of CTL, due to nesting of quantifiers, it is necessary to compute all the states that satisfy a particular subformula, and the fact that the same FSM can appear in different contexts has a greater impact on the resulting verification problem. In fact, the complexity depends on the number of exit-nodes of an FSM (exit-nodes of a hierarchical FSM K are the states that are connected directly to the states of a higher-level hierarchical FSM in which K is embedded; some systems restrict to one exit-node, while other systems allow multiple exit-nodes). We give an algorithm for verifying that a hierarchical FSM K satisfies a CTL formula φ with time complexity $O(|K| \cdot 2^{|\varphi| d})$, where each of the machines has at most d exit-nodes. We prove matching lower bounds by establishing that (1) the problem is Pspace-complete in the size of the formula for single-exit machines, and (2) there is a fixed CTL formula for which the problem is Pspace-complete in the size of the machine, when multiple exits are allowed.

1.1 RELATED WORK

Model checking for ordinary finite-state machines was first introduced in [Clarke and Emerson 1981], and has been studied extensively since then. The complexity of analysis of *concurrent* finite-state machines is also well understood (see, for instance, [Drusinsky and Harel 1994]); concurrency makes the analysis problem exponentially harder. The standard way of applying model checking to hierarchical state machines is by translating them to ordinary state machines by flattening. There have been some attempts to develop heuristics based on the hierarchical structure in symbolic model checking, for example, for choosing variable ordering or for exploiting locality of variables [Chan et al. 1998; Behrmann et al. 1999; Alur et al. 2000; Ball and Rajamani 2000]. However, there has been no systematic study of the impact of hierarchical descriptions with sharing on the analysis problem. It is worth noting that hierarchical FSMs can be viewed as a special case of pushdown automata, where the size of the stack is bounded a priori by a constant. Model checking of pushdown automata is known to be decidable, both for linear-time and branching-time requirements in Exptime [Burkart and Steffen 1992; Boujjani et al. 1997].

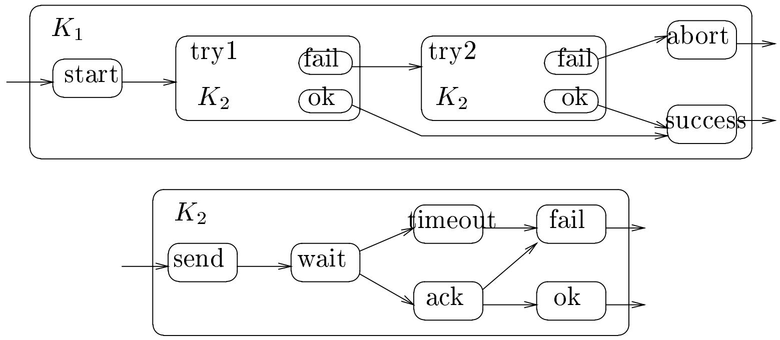

Fig. 1. A sample hierarchical structure.

2. HIERARCHICAL STATE MACHINES

There are many variants of finite-state machines. We choose to present our results in terms of Kripke structures due to their prevalent use in the model checking literature [Clarke and Emerson 1981]. Analogous results hold for the other variants of FSMs such as Mealy- and Moore-type FSMs where inputs and/or outputs occur on the states or transitions.

Kripke structures are state-transition graphs whose states are labeled with atomic propositions. Formally, given a finite set P of atomic propositions, a (flat) *Kripke structure* M over P consists of

(1) a finite set W of states,
(2) an initial state $in \in W$,
(3) a set R of transitions, where each transition is a pair of states, and
(4) a labeling function L that labels each state with a subset of P.

A *hierarchical Kripke structure* K over a set P of atomic propositions is a tuple $\langle K_1, \ldots K_n \rangle$ of structures, where each K_i has the following components:

(1) A finite set N_i of *nodes*.
(2) A finite set B_i of *boxes* (or *supernodes*). The sets N_i and B_i are all pairwise disjoint.
(3) An initial node $in_i \in N_i$.
(4) A subset O_i of N_i, called *exit-nodes*.
(5) A labeling function $X_i : N_i \mapsto 2^P$ that labels each node with a subset of P.
(6) An indexing function $Y_i : B_i \mapsto \{i+1 \ldots n\}$ that maps each box of the ith structure to an index greater than i. That is, if $Y_i(b) = j$, for a box b of structure K_i, then b can be viewed as a reference to the definition of the structure K_j.
(7) An edge relation E_i. Each edge in E_i is a pair (u, v) with source u and sink v:
—source u either is a node of K_i, or is a pair (w_1, w_2), where w_1 is a box of K_i with $Y_i(w_1) = j$ and w_2 is an exit-node of K_j
—sink v is either a node or a box of K_i.

The edges connect nodes and boxes with one another. Edges entering a box implicitly connect to the unique entry-node of the structure associated with that box. On the other hand, edges exiting a box need to explicitly specify the identity of the exit-node among the possible exit-nodes of the structure associated with that box. An example of a hierarchical Kripke structure is shown in Figure 1. The top-level structure K_1 has two boxes, *try1* and *try2*, both of which are mapped to K_2. The structure K_2 represents an attempt to send a message. The attempt fails if a timeout occurs before the receipt of an acknowledgment, or if the acknowledgment is not consistent with the message sent. In the structure K_1, if the first attempt fails, a second attempt is tried.

With each hierarchical structure, we can associate an ordinary flat structure by recursively substituting each box by the structure indexed by the box. Since different boxes can be associated with the same structure, each node can appear in different contexts. The expanded structure corresponding to the hierarchical structure of Figure 1 is shown in Figure 2. The expanded flat structure will be denoted K_1^F. Note that each node of K_2 appears twice in K_1^F: for instance, the node *send* appears as $(\mathit{try1}, \mathit{send})$ and $(\mathit{try2}, \mathit{send})$. In general, a state of the expanded structure is a vector whose last component is a node, and the remaining components are boxes that specify the context.

Now we proceed to a formal definition of expansion of a hierarchical Kripke structure $K = \langle K_1, \ldots K_n \rangle$. For each structure K_i,

(1) The set W_i of *states* of K_i^F is defined inductively:
 —every node of K_i belongs to W_i
 —if u is a box of K_i with $Y_i(u) = j$, and v is a state of K_j^F, then (u, v) belongs to W_i.

(2) The set R_i of transitions of K_i^F is defined inductively:
 —for $(u, v) \in E_i$, if the sink v is a node then $(u, v) \in R_i$, and if v is a box with $Y_i(v) = j$ then $(u, (v, in_j)) \in R_i$
 —if w is a box of K_i with $Y_i(w) = j$, and (u, v) is a transition of K_j^F, then $((w, u), (w, v))$ belongs to R_i.

(3) The labeling function $L_i : W_i \mapsto 2^P$ of K_i^F is defined inductively:
 —if w is a node of K_i, then $L_i(w) = X_i(w)$
 —if $w = (u, v)$, where u is a box of K_i with $Y_i(u) = j$, then $L_i(w)$ equals $L_j(v)$.

The structure $\langle W_i, in_i, R_i, L_i \rangle$ is a flat Kripke structure over P, and is called the *expanded structure* of K_i, denoted K_i^F. The structure K_1^F is also denoted K^F, the expanded structure of K.

The *size* of K_i, denoted $|K_i|$, is the sum of $|N_i|$, $|B_i|$, and $|E_i|$. The size of K is the sum of the sizes of K_i. The *nesting depth* of K, denoted $nd(K)$, is the length of the longest chain $i_1, i_2, \ldots i_j$ of indices such that a box of K_{i_l} is mapped to i_{l+1}. Observe that each state of the expanded structure is a vector of length at most the nesting depth, and the size of the expanded structure K^F can be exponential in the nesting depth, and is $O(|K|^{nd(K)})$.

Note that by our definition the last component of every state is a node, and the propositional labeling of the last component determines the propositional labeling of the entire state. This implies that the state assertions cannot refer to the context, and this choice is important for the algorithms and the complexity bounds.

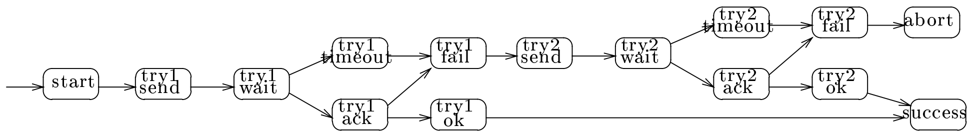

Fig. 2. The expanded structure.

Other variants of this definition are possible. First, we can allow multiple entry-nodes in the definition. Such a structure can be captured by our definition by replacing a structure with k entry-nodes with k structures, each with a single entry-node. Note that each box within a structure also may need to be repeated k times, and consequently, the blow-up in the total size due to the translation would be a factor of k^2 in the worst case. We allow explicitly multiple exit-nodes because the number of exit-nodes dramatically affects the complexity of the analysis problems to be studied (particularly, for branching time). Second, we can allow edges of the form (u, v) where u is a box meaning that the edge may be taken whenever the control is inside the box u. That is, for an edge (u, v), the expanded structure has an edge from every state with first component u to v. Such a definition is useful for modeling interrupts, and can be captured by our definition by introducing a dummy exit-node. With these two extensions, the definition would closely resemble the hierarchical state machines in UML-RT without the features of concurrency and variables.

Finally, the basic (unnested) nodes of the hierarchical structure could themselves be other types of objects; for example, they could be basic message sequence charts, in which case the hierarchical structure specifies a hierarchical (or high-level) MSC [Holzmann et al. 1997; Rudolph et al. 1996]. In this case there is concurrency within each basic MSC, but the hierarchical graph provides a global view of the system. The propositions on the nodes reflect properties for the basic objects (e.g., basic MSCs), from which we can infer properties of the executions of the whole hierarchical system.

3. LINEAR-TIME PROPERTIES

3.1 Reachability

For a hierarchical structure K, a state v is reachable from state u if there is a path from state u to state v in the expanded structure K^F. The input to the reachability problem consists of a hierarchical structure K, and a subset $T \subseteq \cup_i N_i$ of nodes, called the *target region*. Given (K, T), the reachability problem is to determine whether or not some state whose last component is in the target region T is reachable from the top-level entry-node in_1. The target region is usually specified implicitly, for instance, by a propositional formula over P. We assume, that given a node u, the test $u \in T$ can be performed in $O(1)$ time. The reachability problem can be used to check whether a hierarchical structure satisfies an invariant.

3.1.1 *Algorithm for Reachability.* The reachability algorithm is shown in Figure 3. We assume that the sets of nodes and boxes of all the structures are disjoint.

```
Input: A hierarchical structure K and a target region T.
Output: The answer to the reachability problem (K,T).

visited: set of nodes and boxes, initially empty.

procedure DFS(u)
  if u ∈ T then
    print ("Target is reachable");
    break     // terminate the execution of the entire algorithm
  fi ;
  visited := visited ∪ {u};
  if u ∈ N then
    foreach (u,v) ∈ E do
      if v ∉ visited then DFS(v) fi
    od
  else
    i := Y(u);
    if in_i ∉ visited then DFS(in_i) fi ;
    foreach ((u,v),w) ∈ E do
      if v ∈ visited and w ∉ visited then DFS(w) fi
    od
  fi
end DFS.

DFS(in_1);
print("Target is not reachable").
```

Fig. 3. Reachability Algorithm.

We use N to denote $\cup_i N_i$, E to denote $\cup_i E_i$, etc. The algorithm performs a depth-first search using the global data structure *visited* to store the nodes and boxes. While processing a box b with $Y(b) = i$, the algorithm checks if the entry-node in_i of the ith structure was visited before. The first time the algorithm visits some box b with index i, it searches the structure K_i by invoking $DFS(in_i)$. At the end of this search, the set of exit-nodes of K_i that are reachable from in_i will be stored in the data structure *visited*. If the algorithm visits subsequently some other box c with index i, it does not search K_i again, but simply uses the information stored in *visited* to continue the search. It is easy to verify that Algorithm 3 invokes, for every $u \in N \cup B$, $DFS(u)$ at most once. The cost of processing a node or a box u equals the number of edges with source u. Consequently, the running time of the algorithm is linear in the size of the input structure.

Theorem 1. (Reachability). *The depth-first search algorithm of Figure 3 correctly solves the reachability problem* (K,T) *with time complexity* $O(|K|)$.

Proof. First note that for any structure K_i, if the depth-first search $DFS(in_i)$ is invoked, then at the time of invocation, the set *visited* does not contain any node or box of K_i. Second, the search $DFS(in_i)$ does not encounter any node or box of a structure K_j with $j < i$. Now, we prove by backward induction on i that for each K_i, when $DFS(in_i)$ is invoked, if a node in the target set is reachable from in_i, the routine terminates after printing "Target is reachable"; else the routine terminates after inserting all the nodes reachable from in_i in the set *visited*.

The structure K_n has no boxes, and is like an ordinary Kripke structure. Since *visited* contains no node of K_n when $DFS(in_n)$ is invoked, it behaves like a standard depth-first search. By the correctness of the standard search, $DFS(in_n)$ either aborts by encountering a node in the target set, or visits all the nodes reachable from in_n.

Now consider a structure K_i, with $i < n$, with a possibly nonempty set of boxes. By the semantics of hierarchical structures, for the purpose of the search, K_i can be viewed as an ordinary Kripke structure where each box b with index j is replaced by the entry and exit-nodes of K_j, and edges that connect the entry-node with every exit-node reachable from it. The routine $DFS(in_i)$ behaves like the standard depth-first search on this structure.

Each structure is searched at most once. The running time follows from the fact that the standard depth search requires time proportional to the number of nodes and edges. □

3.1.2 *Complexity of Reachability.* For flat Kripke structures, deciding reachability between two states is in NLOGSPACE. For hierarchical structures, however, the reachability problem becomes PTIME-hard even if we require a single exit for every structure.

THEOREM 2. (REACHABILITY: COMPLEXITY). *The reachability problem* (K,T) *is Ptime-complete.*

PROOF. The proof is by reduction from the alternating reachability problem. Consider an AND-OR graph, i.e., a graph with two disjoint sets V_A and V_O of vertices, called AND and OR vertices, respectively, and a set E of edges. Given a set T of target vertices, and an initial vertex v, consider the following 2-player game. The initial game state is vertex v. Whenever the game state is an OR-vertex, player-1 chooses the successor vertex by following an edge in E, and whenever the game state is an AND-vertex, player-2 chooses the successor vertex by following an edge in E. If the game state is a vertex in the target set, the player-1 wins, and if the game continues forever, player-2 wins. The problem of determining whether player-1 has a winning strategy is known to be Ptime-complete. Note that if player-1 has a winning strategy, then it has a memoryless winning strategy that ensures victory within at most $n-1$ steps, where n is the total number of vertices.

Given an AND-OR graph and a target set, we construct a hierarchical Kripke structure as follows. Without loss of generality, assume that each node has precisely two successors. Suppose the graph has n vertices. Then for each $1 \leq i \leq n$, and each vertex u, we construct a hierarchical Kripke structure $K_{u,i}$. Each $K_{u,i}$ has two nodes, the entry-node $in_{u,i}$, and the single exit-node $out_{u,i}$.

The structures $K_{u,n}$ have no boxes. If u is in the target set, then there is an edge from $in_{u,n}$ to $out_{u,n}$; otherwise there are no edges.

For $i < n$, the structures $K_{u,i}$ are shown in Figure 4. If u is in the target set, then there are no boxes, and an edge connecting the entry-node to the exit-node. Otherwise, there are two boxes, mapped to $K_{v,i+1}$ and $K_{w,i+1}$, corresponding to the two successors v and w of u. The edges are as shown in Figure 4: if u is an OR-vertex, then the exit-node can be reached after visiting one of the boxes, while for an AND-vertex, the exit-node can be reached only after visiting both the boxes.

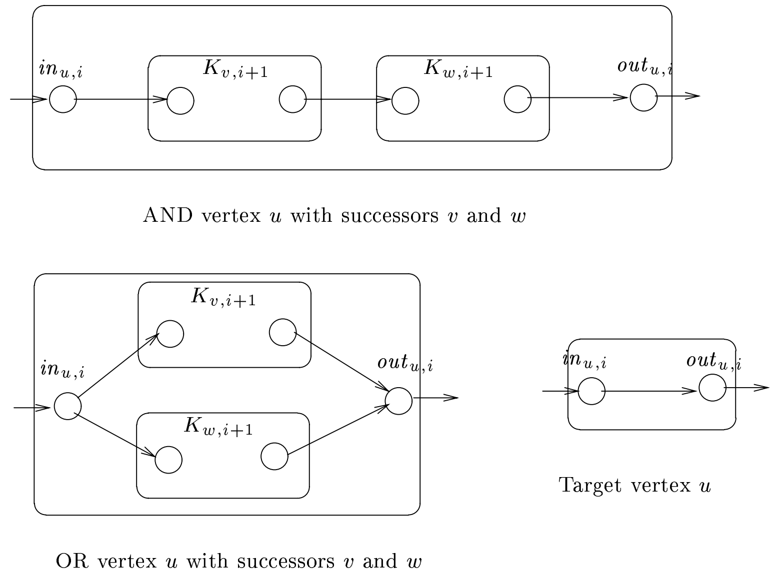

Fig. 4. Reduction from AND-OR reachability.

The correctness of the reduction is captured by the following claim: for each $1 \leq i \leq n$, the exit-node of $K_{u,i}$ can be reached from its entry-node iff the player-1 has a winning strategy to reach a target vertex starting from the vertex u in at most $n-i$ steps. This claim is proved by induction on i, and the proof is straightforward. Consequently, existence of a winning strategy from an initial vertex u reduces to reachability of $out_{u,1}$ from $in_{u,1}$.

Note that the size of each $K_{u,i}$ is constant, and thus, the total size of the hierarchical structure is quadratic in the size of the input AND-OR graph. □

3.2 Cycle Detection

The basic problem encountered during verification of liveness requirements is to check whether a specified state can be reached repeatedly [Vardi and Wolper 1986; Holzmann 1991]. As in the reachability problem, the input to the *cycle-detection problem* consists of a hierarchical Kripke structure K, and a target region $T \subseteq N$. Given (K, T), the cycle-detection problem is to determine whether there exists a state u whose last component is in the target region T such that (1) u is reachable from the entry-node in_1, and (2) u is reachable from itself.

The cycle-detection algorithm is shown in Figure 5. The algorithm involves two searches, a primary and a secondary. The algorithm uses a global data structure $visited_P$ to store the states encountered during the primary search, and $visited_S$ to store the states encountered during the secondary search. The primary search is performed by the procedure DFS_P. When the search backtracks from a node in the target region, it initiates a secondary search. The secondary search is performed by the procedure DFS_S, and it searches for a cycle. Since the stack stores a path leading to the node or the box from which the secondary search was initiated, the

Input: A hierarchical structure K and a target region T.
Output: The answer to the cycle-detection problem (K, T).

```
visited_P, visited_S: sets of nodes and boxes, initially empty
Stack: Stack of nodes and boxes, initially empty

procedure DFS_P(u)
  Push(u, Stack);
  visited_P := visited_P ∪ {u};
  if u ∈ N then
    foreach (u,v) ∈ E do
      if v ∉ visited_P then DFS_P(v) fi
    od ;
    if u ∈ T and u ∉ visited_S then DFS_S(u) fi
  else
    i := Y(u);
    if in_i ∉ visited_P then DFS_P(in_i) fi ;
    foreach ((u,v),w) ∈ E do
      if v ∈ visited_P and w ∉ visited_P then DFS_P(w) fi ;
      if v ∈ visited_S then
        if w ∈ Stack then
          print("Cycle found"); break
        fi ;
        if w ∉ visited_S then DFS_S(w) fi ;
      fi ;
    od ;
  fi ;
  Pop(Stack);
end DFS_P.

procedure DFS_S(u)
  visited_S := visited_S ∪ {u};
  if u ∈ N then
    foreach (u,v) ∈ E do
      if v ∈ Stack then
        print("Cycle found");break
      fi ;
      if v ∉ visited_S then DFS_S(v) fi
    od
  else
    i := Y(u);
    foreach ((u,v),w) ∈ E do
      if v ∈ visited_P then
        if w ∈ Stack then
          print("Cycle found"); break
        fi ;
        if w ∉ visited_S then DFS_S(w) fi
      fi
    od
  fi
end DFS_S.

DFS_P(in_1);
print("Cycle not found").
```

Fig. 5. Cycle-detection algorithm.

secondary search terminates with a positive response when it encounters a node or a box on the stack. The interleaving of the two searches is similar to the cycle-detection algorithm for ordinary Kripke structures [Courcoubetis et al. 1992], except that we have both boxes and nodes here. When the primary search backtracks from a box b, it invokes a secondary search from an exit-node v of the box b if there is a path from the entry-node to v involving a node in the target region, equivalently, when v has already been encountered during the secondary search.

THEOREM 3. (CYCLE DETECTION). *The nested depth-first search algorithm of Figure 5 correctly solves the cycle-detection problem* (K,T) *with time complexity* $O(|K|)$.

PROOF. Let us call a path/cycle to be a T-path/T-cycle if it contains a state whose last component is in T. We establish, that for every i, if the primary search $DFS_P(in_i)$ is invoked by the algorithm of Figure 5, then

(1) If the expanded structure K_i^F contains a T-cycle reachable from in_i, then $DFS_P(in_i)$ detects such a cycle and terminates by printing "Cycle found."

(2) If the expanded structure K_i^F does not contain a reachable T-cycle, then $DFS_P(in_i)$ terminates, and upon termination
   (a) a node or a box of K_i belongs to $visited_P$ iff it is reachable from in_i in K_i^F,
   (b) a node or a box of K_i belongs to $visited_S$ iff it is reachable from node in_i in K_i^F along a T-path.

This claim is proved by induction on the index i from n down to 1.

First, consider the invocation $DFS_P(in_n)$. At this point, the sets $visited_P$ and $visited_S$ do not contain any node of K_n. Since K_n has no boxes, the primary search behaves like a standard depth-first search. When the primary search from a node u terminates, the stack contains a trajectory from in_n to u. If u belongs to T, a secondary search is initiated, and if this search encounters a node on the stack, we can conclude that there is a cycle containing u.

The proof that if K_n has a reachable T-cycle, then $DFS_P(in_n)$ detects it, is similar to the proof in [Courcoubetis et al. 1992]. Let us order the nodes of K_n according to the termination times of the primary search: if $DFS_P(u)$ terminates before $DFS_P(v)$, then u gets a lower number than v. Suppose K_n contains a reachable T-cycle. Let $u_0, \ldots u_k$ be the ordering of reachable nodes in T according to our numbering. Let u_i be the first node in this list that belongs to a cycle. Then, no node of this cycle can be reachable from u_j for $j < i$ (else there will be a cycle containing u_j). Consequently, when the primary search from u_i is over, the set $visited_S$ does not contain any node of the cycle, which guarantees that $DFS_S(u_i)$ will discover the cycle.

When K_n has no reachable T-cycle, DFS_P is invoked from every node reachable from in_n, and all these nodes will be inserted in $visited_P$. The secondary search is initiated from every reachable node in T, and $visited_S$ will contain nodes that are reachable from such nodes.

Now consider the claim for $i < n$. Again, when $DFS_P(in_i)$ is invoked, the sets $visited_P$ and $visited_S$ do not contain any elements of K_i. Whenever the primary search is invoked from a node or a box of K_i, the stack contains a trajectory from in_i to that element. Since the claim holds for $j > i$, by definition of the hierarchical structure, a box b mapped to index j can be replaced by its entry-node, exit-nodes, edges connecting the entry-node to those exit-nodes reachable from it. Furthermore, if an exit-node v is in the set $\mathit{visited}_S$, K_j has a T-path from the entry-node to v. It follows, that if the secondary search encounters a node on the stack, there is a T-cycle, reachable from in_i.

We proceed to establish, that if K_i has a reachable T-cycle, then $DFS_P(in_i)$ discovers it. The proof is quite similar as in the case of K_n except that now we need to consider boxes also. Let us order the reachable boxes and nodes of K_i according to the termination times of the primary search. Let u be the first element in this ordering such that u is a node in T that belongs to a cycle, or u is a box mapped to j such that there is a cycle that contains (u, in_j) and some node in T. If u is a node, then by an argument as in the base case, when $DFS_S(u)$ is invoked, none of the elements in the cycle containing u are in $\mathit{visited}_S$, and hence, $DFS_S(u)$ will discover the cycle. Suppose u is a box mapped to index $j > i$. If K_j contains a reachable T-cycle, by induction hypothesis, $DFS_P(in_j)$ should discover it. Suppose that K_j does not contain a reachable T-cycle. Then the cycle contains a T-path from in_j to an exit-node v of K_j. There may be multiple choices of v, and in that case, consider the least one according to their order in the list of edges of the form $((u, v), w)$. By induction hypothesis, v will be in $\mathit{visited}_S$. Consider the edge $((u, v), w)$ that participates in the cycle (if there are multiple choices for w, consider the least one according to the order in the list of edges). Either w will be on stack, or the call $DFS_S(w)$ will discover the cycle.

For complexity analysis observe, that for every $u \in N \cup B$, $DFS_P(u)$ is invoked at most once, and $DFS_S(u)$ is invoked at most once. This gives linear-time complexity. □

Note that if all nodes of a hierarchical Kripke structure have self-loops then the cycle-detection problem is the same as the reachability problem. Consequently, the cycle-detection problem is also Ptime-hard, and thus, Ptime-complete.

3.3 Automata Emptiness

Let $M = \langle W, in, R, L\rangle$ be a Kripke structure over proposition set P. For a state $w \in W$, a source-w trajectory of M is an infinite sequence $w_0 w_1 w_2 \ldots$ of states in W such that $w_0 = w$ and $w_i R w_{i+1}$ for all $i \geq 0$. An initialized trajectory is a source-in trajectory. The trace corresponding to the trajectory $w_0 w_1 w_2 \ldots$ is the infinite sequence $L(w_0)L(w_1)\ldots$ over 2^P obtained by replacing each state with its label. The language $\mathcal{L}(M)$ consists of all the traces corresponding to the initialized trajectories of M.

A Büchi automaton A over P consists of a Kripke structure M over P and a set T of accepting states. An accepting trajectory of A is an initialized trajectory $w_0 w_1 w_2 \ldots$ of M such that $w_i \in T$ for infinitely many i. The language $\mathcal{L}(A)$ consists of all traces corresponding to accepting trajectories of A.

The input to the *automata-emptiness* problem consists of a hierarchical structure K over P and an automaton A over P. Given (K, A), the automata-emptiness problem is to determine whether the intersection $\mathcal{L}(A) \cap \mathcal{L}(K^F)$ is empty. This is the automata-theoretic approach to verification: if the automaton A accepts undesirable or bad behaviors, checking emptiness of $\mathcal{L}(A) \cap \mathcal{L}(K^F)$ corresponds to

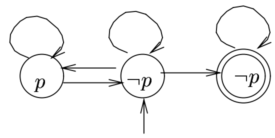

Fig. 6. A sample Büchi automaton.

ensuring that the model has no bad behaviors.

As an example, consider the 3-state automaton of Figure 6 over the single proposition p. The only accepting state is the rightmost state, and thus any accepting run must eventually keep cycling on this state. Thus, the language of the automaton contains all infinite words in which p is true only finitely many times. If infinite repetition of the property p is the desired requirement, then the automaton captures all the bad behaviors (see [Vardi and Wolper 1986], [Kurshan 1994], [Thomas 1990], and [Holzmann 1991] for more details on ω-automata and their role in formal verification).

We solve the automata-emptiness problem (K, A) by reduction to a cycle-detection problem for the hierarchical structure obtained by constructing the product of K with A as follows. Let $K = \langle K_1, \ldots K_n \rangle$ with $K_i = \langle N_i, B_i, in_i, O_i, X_i, Y_i, E_i \rangle$, and let $A = \langle W = \{w_1, \ldots w_m\}, w_1, R, L, T \rangle$. The product structure $K \otimes A$ is the hierarchical structure $\langle K_{11}, \ldots K_{1m}, \ldots K_{n1}, \ldots K_{nm} \rangle$ with nm structures. Each structure K_i is repeated m times, one for every possible way of pairing its entry-node with a state of A. Each structure K_{ij} has the following components:

(1) A node of K_{ij} is a pair (u, w), where $u \in N_i$, $w \in W$, and $X(u) = L(w)$; the label of (u, w) equals $X(u)$.

(2) A box of K_{ij} is a pair (b, w), where $b \in B_i$ and $w \in W$; the index of (b, w) is $m(i' - 1) + j'$ if $Y(b) = i'$ and $w = w_{j'}$.

(3) The entry-node of K_{ij} is (in_i, w_j), and a node (u, w) is an exit-node of K_{ij} if $u \in O_i$.

(4) Consider a transition (w, x) of A. While pairing it with the edges in E_i, there are four cases to consider, depending on whether the edge connects a node to a node, a node to a box, a box to a node, or a box to a box.
 —For an edge (u, v) of K_i with $u, v \in N_i$, $X(u) = L(w)$, and $X(v) = L(x)$, there is an edge $((u, w), (v, x))$ in K_{ij}.
 —For an edge (u, b) of K_i with $u \in N_i$, $b \in B_i$, $X(u) = L(w)$, and $X(in_{Y(b)}) = L(x)$, there is an edge $((u, w), (b, x))$ in K_{ij}.
 —For an edge $((b, u), v)$ of K_i with $b \in B_i$, $u \in O_{Y(b)}$, $v \in N_i$, $X(u) = L(w)$, and $X(v) = L(x)$, for every $y \in W$, there is an edge $(((b, y), (u, w)), (v, x))$ in K_{ij}.
 —For an edge $((b, u), c)$ of K_i with $b, c \in B_i$, $u \in O_{Y(b)}$, $X(u) = L(w)$, and $X(in_{Y(c)}) = L(x)$, for every $y \in W$, there is an edge $(((b, y), (u, w)), (c, x))$ in K_{ij}.

Note that if the propositional labeling of in_i and w_j is different, then the entry-node of K_{ij} is not defined; the rules for edges ensure that a box mapped to such a structure is not reachable, and thus the structure K_{ij} can be omitted. The structures $K_{12}, \ldots K_{1m}$ are not reachable from the entry-node of K_{11}, and hence can also be omitted. The correctness of the product construction is captured by the following lemma which says that some trace of K^F is accepted by A iff there is a reachable cycle in $K \otimes A$ containing a node of the form (u, w) with $w \in T$.

LEMMA 4. (PRODUCT CONSTRUCTION). *The intersection $\mathcal{L}(K^F) \cap \mathcal{L}(A)$ is nonempty iff the answer to the cycle-detection question $(K \otimes A, N \times T)$ is* YES.

PROOF. The first step is to establish that the flattening of $K \otimes A$ is isomorphic to the product of K^F and A. This follows directly from the definitions of flattening and the product. The lemma follows from the standard property of the product construction: the language of the (ordinary) product of two automata is the intersection of their languages. □

Hence, we can solve the automata-emptiness question using the cycle-detection algorithm. Note that the algorithm allows us to explore the graph on-the-fly. That is, we do not need to construct $K \otimes A$ explicitly in advance, but rather its nodes and boxes can be generated on-the-fly when needed. In the construction of $K \otimes A$, the number of structures gets multiplied by the number of states of A. Within each structure, each node and box in the original hierarchical structure gets paired with a state of A in the worst case, and within each box each exit-node gets paired with a state of A. Consequently, the total number of edges in the structure K_{ij} is bounded by the product of the number of edges in K_i, the number of edges in A, and the number of states in A. This leads to the following bound:

THEOREM 5. (AUTOMATA EMPTINESS). *The automata emptiness question (K, A) can be solved by reduction to the cycle-detection problem in time $O(a^2 \cdot |A| \cdot |K|)$, where a is the number of states in A.*

3.4 Linear Temporal Logic

Requirements of trace-sets can be specified using the temporal logic LTL [Pnueli 1977; Manna and Pnueli 1991]. For instance, the requirement that "p should be repeated infinitely often" is specified by the LTL-formula $\Box\Diamond\, p$. A formula φ of LTL over propositions P is interpreted over an infinite sequence over 2^P. A hierarchical structure K satisfies a formula φ, written $K \models \varphi$, iff every trace in $\mathcal{L}(K^F)$ satisfies the formula φ. The input to the *LTL model-checking problem* consists of a hierarchical structure K and a formula φ of LTL. The model-checking problem (K, φ) is to determine whether or not K satisfies φ.

To solve the model-checking problem, we construct a Büchi automaton $A_{\neg\varphi}$ such that the language $\mathcal{L}(A_{\neg\varphi})$ consists of precisely those traces that do not satisfy φ. This can be done using one of the known translations from LTL to Büchi automata [Lichtenstein and Pnueli 1985; Vardi and Wolper 1986]. The number of states of $A_{\neg\varphi}$ is $O(2^{|\varphi|})$. Then, the hierarchical structure K satisfies φ iff $\mathcal{L}(K^F) \cap \mathcal{L}(A_{\neg\varphi})$ is empty. Thus, the model-checking problem reduces to the automata-emptiness problem. The complexity of solving the automata-emptiness problem, together with the cost of translating an LTL formula to a Büchi automaton, yields the following.

THEOREM 6. (LTL MODEL CHECKING). *The LTL model-checking problem*

(K, φ) can be solved in time $O(|K| \cdot 8^{|\varphi|})$.

An alternative approach to solve the LTL model-checking problem (K, φ) is to search for an accepting cycle in the product of the expanded structure K^F with the automaton $A_{\neg\varphi}$. This product has $|K|^{nd(K)} \cdot 2^{|\varphi|}$ states, and each state of this product can be represented in space $O(|K| \cdot |\varphi|)$. Transitions of the product can be computed efficiently using standard techniques, and consequently the search can be performed in space $O(|K| \cdot |\varphi|)$. This gives a Pspace upper bound on the LTL model-checking problem. It is known that the LTL model-checking problem for ordinary Kripke structures is Pspace-hard. This leads to the following

THEOREM 7. (LTL COMPLEXITY). *The LTL model-checking problem (K, φ) is Pspace-complete.*

4. BRANCHING-TIME PROPERTIES

Now we turn our attention to verifying requirements specified in the branching-time temporal logic CTL [Clarke and Emerson 1981]. Branching-time logics provide quantification over computations of the system allowing specification of requirements such as "along some computation, eventually p" and "along all computations, eventually p." Formally, given a set P of propositions, the set of CTL formulas is defined inductively by the grammar

$$\varphi := p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists\bigcirc\varphi \mid \exists\Box\varphi \mid \varphi\,\exists\mathcal{U}\varphi$$

where $p \in P$. For a Kripke structure $M = (W, in, R, L)$ and a state $w \in W$, the satisfaction relation $w \models \varphi$ is defined below:

$w \models p$	iff	$p \in L(w)$;
$w \models \neg\varphi$	iff	$w \not\models \varphi$;
$w \models \varphi \wedge \psi$	iff	$w \models \varphi$ and $w \models \psi$;
$w \models \exists\bigcirc\varphi$	iff	there exists a state u with wRu and $u \models \varphi$;
$w \models \exists\Box\varphi$	iff	there exists a source-w trajectory $w_0 w_1 \ldots$ such that $w_i \models \varphi$ for all $i \geq 0$;
$w \models \varphi\,\exists\mathcal{U}\psi$	iff	there exists a source-w trajectory $w_0 w_1 \ldots$ such that $w_k \models \psi$ for some $k \geq 0$, and $w_i \models \varphi$ for all $0 \leq i < k$.

Many additional temporal operators can be defined as derived connectives. Some standard derived operators include $\exists\Diamond\psi$ (along some trajectory, eventually ψ) for ($true$ $\exists\mathcal{U}\psi$), $\forall\Box\psi$ (ψ holds in all reachables states) for $\neg\exists\Diamond\neg\psi$, and $\forall\bigcirc\psi$ (ψ holds in all successor states) for $\neg\exists\bigcirc\neg\psi$.

The Kripke structure M satisfies the formula φ, written $M \models \varphi$, iff $in \models \varphi$. A hierarchical Kripke structure K satisfies the CTL formula φ iff $K^F \models \varphi$. The *CTL model-checking problem* is to decide, given a hierarchical Kripke structure K and a CTL formula φ, whether K satisfies φ.

4.1 Single-Exit Case

First, we consider the case when every structure has a single exit-node. In this case, the edges from boxes need not specify the exit-node of the structure associated with the box, and we assume that each E_i is a binary relation over $N_i \cup B_i$. We will use out_i to denote the unique exit-node of K_i.

Input: A hierarchical structure K and a CTL formula φ.
Output: The answer to the model-checking problem (K, φ).

$sub(\varphi)$:= list of subformulas of φ in increasing order of size.
foreach $\psi \in sub(\varphi)$ **do**
  case ψ:
    $\psi \in P$: skip;
    $\psi = \neg\chi$:
      foreach $u \in N$ **do**
        if $\chi \notin X(u)$ **then** $X(u) := X(u) \cup \{\psi\}$ **fi**
      od ;
    $\psi = \psi_1 \wedge \psi_2$:
      foreach $u \in N$ **do**
        if $\psi_1 \in X(u)$ and $\psi_2 \in X(u)$ **then** $X(u) := X(u) \cup \{\psi\}$ **fi**
      od ;
    $\psi = \exists\bigcirc\,\chi$: $K := CheckNext(K, \chi)$;
    $\psi = \psi_1\, \exists\mathcal{U} \psi_2$: $K := CheckUntil(K, \psi_1, \psi_2)$
    $\psi = \exists\Box\chi$: $K := CheckAlways(K, \chi)$;
od

if $\varphi \in X(in_1)$ **then print** ("φ satisfied") **else print** ("φ not satisfied") **fi**.

Fig. 7. CTL model checking.

4.1.1 *Model-Checking Algorithm.* The main loop of the model-checking algorithm is shown in Figure 7. At the beginning, for each node u, the labeling set $X(u)$ is initialized to contain the atomic propositions that are true at the node u. The algorithm considers subformulas of φ starting with the innermost subformulas, and extends the label $X(u)$, at each node u, with the subformulas that are satisfied at u. A node can appear in multiple contexts, and whether it satisfies a subformula can depend on the context. Consequently, the algorithm repeatedly transforms the hierarchical structure K, and is designed to maintain the following property:

> After the algorithm processes a subformula ψ, if a node u is labeled with ψ then in the current expanded structure K^F, *every* state with the last component u satisfies ψ, and if a node u is not labeled with ψ then in K^F, *no* state with the last component u satisfies ψ.

At the end, if the entry-node of the first structure is labeled with φ, then the original structure satisfies φ. Processing of atomic propositions and subformulas of the form $\neg\chi$ and $\psi_1 \wedge \psi_2$ is straightforward. Handling of temporal operators requires modifying the structure; each subformula can double the number of structures. We will consider the cases corresponding to the temporal operators.

4.1.2 *CheckNext: Processing of* $\exists\bigcirc$. To illustrate the ideas, first consider a hierarchical structure $K = \langle K_1, K_2\rangle$. Consider a formula $\psi = \exists\bigcirc\, p$ for an atomic proposition p. We wish to compute the set of nodes at which ψ is satisfied. Consider a node u of K_2. Multiple boxes of K_1 may be mapped to K_2, and hence, u may appear in multiple contexts in the expanded structure (i.e., there may be multiple states whose last component is u). If u is not the exit-node, then the successors of u do not depend on the context. Hence, the truth of ψ is identical in all states corresponding to u, and can be determined from the successors of u within K_2: ψ holds at u if some successor node of u is labeled with p. If u is the exit-node of

K_2, then the truth of ψ may depend on the context. For instance, the truth of $\exists\bigcirc$ *abort* at the exit-node *fail* of the structure K_2 of Figure 1 is different in the two instances; the formula is false in (*try1*, *fail*) and is true in (*try2*, *fail*). In this case, we need to create two copies of K_2: K_2^0 and K_2^1. The superscript indicates whether or not the exit-node of K_2 has some successor that satisfies p and is outside K_2. The exit-node of K_2^1 is labeled with ψ. The exit-node of K_2^0 is labeled with ψ only if it has a successor within K_2 that satisfies p. The mapping of boxes of K_1 must be consistent with the intended meaning: a box of K_1 which has a successor satisfying p is mapped to K_2^1 and to K_2^0 otherwise.

Now we proceed to define the computation of *CheckNext* for the general case. The input to *CheckNext* is a hierarchical structure K and a subformula χ. Let $K = \langle K_1, \ldots K_n \rangle$ where each $K_i = \langle N_i, B_i, in_i, out_i, X_i, Y_i, E_i \rangle$. Assume that the nodes of K are already labeled with the formula χ, and let $\psi = \exists\bigcirc\chi$. For $u \in N_i \cup B_i$, define $u \models_i \psi$ if either (i) there exists a node v of K_i with $(u, v) \in E_i$ and $\chi \in X_i(v)$, or (ii) there exists a box b of K_i with index k such that $(u, b) \in E_i$ and $\chi \in X(in_k)$. Thus, $u \models_i \psi$ means that u has a χ-successor within K_i or substructures nested within K_i. For a given u, whether $u \models_i \psi$ holds can be determined simply by examining the edges out of u.

CheckNext(K, χ) returns a hierarchical structure $K' = \langle K_1^0, K_1^1, \ldots K_n^0, K_n^1 \rangle$ by replacing each K_i with two structures $K_i^0 = \langle N_i, B_i, in_i, out_i, X_i^0, Y_i', E_i \rangle$ and $K_i^1 = \langle N_i, B_i, in_i, out_i, X_i^1, Y_i', E_i \rangle$, where the revised mapping Y_i' of boxes to indices is defined as follows:

> For a box b of K_i with $Y_i(b) = j$, if $b \models_i \psi$ then $Y_i'(b) = 2j$, and otherwise, $Y_i'(b) = 2j - 1$.

The revised labeling functions X_i^0 and X_i^1 that extend X_i to include labeling of nodes with ψ are defined as follows:

(1) For a node u of K_i^0, if $u \models_i \psi$ then $\psi \in X_i^0(u)$, and otherwise, $\psi \notin X_i^0(u)$.

(2) For a node u of K_i^1, if $u \models_i \psi$ or if u is the exit-node of K_i, then $\psi \in X_i^1(u)$, and otherwise, $\psi \notin X_i^1(u)$.

Observe that if the exit-node $out_i \models_i \psi$ then K_i^0 and K_i^1 are identical, and in this case, we can delete one of them (i.e., there is no need to create two instances).

4.1.3 *CheckUntil: Processing of $\exists\mathcal{U}$.* Whether a node u of a structure K_i satisfies the until-formula $\psi = \psi_1 \exists\mathcal{U} \psi_2$ may depend on what happens after exiting K_i, and thus, different occurrences may assign different truth values to $\psi_1 \exists\mathcal{U} \psi_2$, requiring splitting of each structure into two. The input to *CheckUntil* consists of a hierarchical structure K and formulas ψ_1 and ψ_2. Let $K = \langle K_1, \ldots K_n \rangle$, where each $K_i = \langle N_i, B_i, in_i, out_i, X_i, Y_i, E_i \rangle$. The computation proceeds in two phases.

In the first phase, we partition the index-set $\{1, \ldots n\}$ into three sets, YES, NO, and MAYBE, with the following interpretation. An index i belongs to YES when the entry-node in_i satisfies the until-formula ψ within the expanded structure K_i^F. Then, in K^F, every occurrence of the entry-node in_i satisfies ψ. Now consider an index i that does not belong to YES. It belongs to MAYBE if within K_i^F there is a finite trajectory from in_i to $exit_i$ that contains only states labeled with ψ_1. In this case, it is possible that for some occurrences of K_i^F in K^F, the entry-node

YES:= $\emptyset$; NO:= $\emptyset$; MAYBE:= $\emptyset$;
for $i = n$ **downto** 1 **do**
  if $in_i \models_i (\psi_1 \vee \mathrm{MAYBE})\, \exists\mathcal{U}(\psi_2 \vee \mathrm{YES})$
  then YES := YES $\cup \{i\}$
  else
    if $in_i \models_i (\psi_1 \vee \mathrm{MAYBE})\, \exists\mathcal{U}(out_i \wedge \psi_1)$
    then MAYBE := MAYBE $\cup \{i\}$
    else NO := NO $\cup \{i\}$
    fi
  fi
od

Fig. 8. First phase of CheckUntil.

satisfies ψ depending on whether or not the corresponding exit-node satisfies ψ. In the last case, the index i belongs to NO, and in every occurrence of K_i^F, the entry-node does not satisfy the formula ψ. This happens when upon entering K_i^F, a trajectory is guaranteed to encounter a node violating $\neg\psi_1$ before visiting a node satisfying ψ_2 or the exit-node of K_i.

To express the computation succinctly, define the satisfaction relation $\models_i$ that considers the structure K_i to be an ordinary structure over vertices $N_i \cup B_i$. In this interpretation, a box is considered like an ordinary vertex, which is labeled with YES, NO, or MAYBE, according to the characterization of the index that it is mapped to. In addition, we modify the definition of the satisfaction so that an $\exists\mathcal{U}$-formula can be satisfied along a finite trajectory. For example,

$$u \models_i (\psi_1 \vee \mathrm{MAYBE})\, \exists\mathcal{U}(\psi_2 \vee \mathrm{YES}),$$

for a node or a box u of the structure K_i, translates to the requirement that there exists a finite path $v_0 v_1 \ldots v_k$ of vertices $v_l \in N_i \cup B_i$ such that

(1) v_0 equals u,
(2) $v_k \models_i (\psi_2 \vee \mathrm{YES})$ meaning that if v_k is a box such that $Y_i(v_k) = j$ then $j \in \mathrm{YES}$, and if v_k is a node then $\psi_2 \in X_i(v_k)$, and
(3) for $0 \leq l < k$, $v_l \models_i (\psi_1 \vee \mathrm{MAYBE})$ meaning that if v_l is a box such that $Y_i(v_l) = j$ then $j \in \mathrm{MAYBE}$, and if v_l is a node then $\psi_1 \in X_i(v_l)$.

The desired partitioning is computed by the procedure of Figure 8. The computation for each index i can be performed by a simple depth-first search over the nodes and boxes of K_i starting at the node in_i in time $|K_i|$.

For example, consider the hierarchical structure shown in Figure 9. The atomic propositions are p and q, and each node is labeled with the atomic propositions true at that node. The structure K_1 has two boxes both of which are mapped to the structure K_2. The formula of interest is $p\, \exists\mathcal{U}\, q$. In the first phase, we first consider the structure K_2, and conclude that it corresponds to the case MAYBE, since there is a path from its entry-node to exit-node that visits only the nodes satisfying p. Then, we process the structure K_1. At this step, the boxes are considered as vertices labeled with MAYBE (see Figure 10). Since the entry-node satisfies the formula $(p \vee \mathrm{MAYBE})\, \exists\mathcal{U}(q \vee \mathrm{YES})$, we conclude that the structure K_1 corresponds to the case YES. Thus, after executing the algorithm of Figure 8, the set MAYBE contains the index 2, and the set YES contains the index 1.

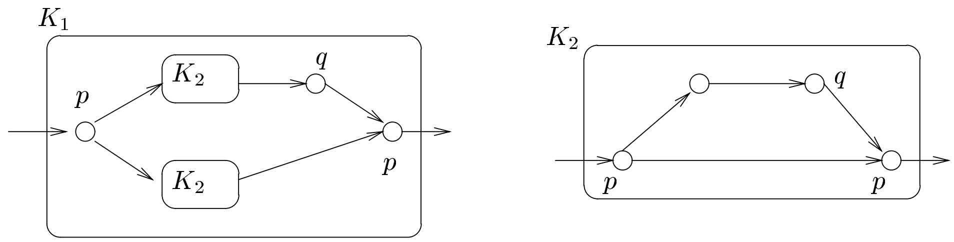

Fig. 9. Example for illustrating checking of $\exists\mathcal{U}$ formulas.

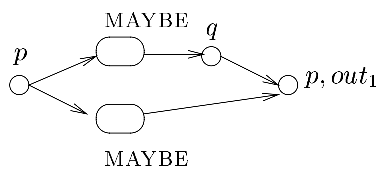

Fig. 10. Illustration of phase 1 of CheckUntil.

In the second phase, the new hierarchical structure K' along with the labeling of ψ is constructed. To obtain K', each structure K_i is split into two: K_i^0 and K_i^1. A box b that is previously mapped to K_i will be mapped to K_i^1 if there is a path starting at b that satisfies ψ, and otherwise to K_i^0. Consequently, nodes within K_i^1 can satisfy ψ along a path that exits K_i, while nodes within K_i^0 can satisfy ψ only if they satisfy it within K_i. The new structure $K' = \langle K_1^0, K_1^1, \ldots K_n^0, K_n^1\rangle$ is obtained by replacing each K_i of K with two structures $K_i^0 = \langle N_i, B_i, in_i, out_i, X_i^0, Y_i^0, E_i\rangle$ and $K_i^1 = \langle N_i, B_i, in_i, out_i, X_i^1, Y_i^1, E_i\rangle$, where the new components are defined as follows:

(1) For the indexing of boxes of K_i^0, consider a box b with $Y_i(b) = j$.
If $b \models_i \exists\bigcirc[(\psi_1 \vee \text{MAYBE})\, \exists\mathcal{U}(\psi_2 \vee \text{YES})]$ then $Y_i^0(b) = 2j$, else $Y_i^0(b) = 2j-1$.

(2) For the indexing of boxes of K_i^1, consider a box b with $Y_i(b) = j$.
If $b \models_i \exists\bigcirc[(\psi_1 \vee \text{MAYBE})\, \exists\mathcal{U}(\psi_2 \vee \text{YES} \vee (out_i \wedge \psi_1))]$, then $Y_i^1(b) = 2j$, else $Y_i^1(b) = 2j-1$.

(3) For labeling of a node u of K_i^0, if $u \models_i (\psi_1 \vee \text{MAYBE})\, \exists\mathcal{U}(\psi_2 \vee \text{YES})$ then $X_i^0(u)$ equals $X_i(u)$ with ψ added to it; else it equals $X_i(u)$.

(4) For labeling of a node u of K_i^1, if $u \models_i (\psi_1 \vee \text{MAYBE})\, \exists\mathcal{U}(\psi_2 \vee \text{YES} \vee (out_i \wedge \psi_1))$, then $X_i^1(u)$ equals $X_i(u)$ with ψ added to it; else it equals $X_i(u)$.

The hierarchical structure K' computed by the phase 2 on the example of Figure 9 is shown in Figure 11. Each of the structures K_1 and K_2 are split into two. In K_2^0, the exit-node is assumed not to satisfy $p\,\exists\mathcal{U}q$, while in K_2^1, the exit-node is assumed to satisfy $p\,\exists\mathcal{U}q$, and the truth of $p\,\exists\mathcal{U}q$ at other nodes is determined based on these assumptions. In particular, the entry-node of K_2^1 is labeled with $p\,\exists\mathcal{U}q$,

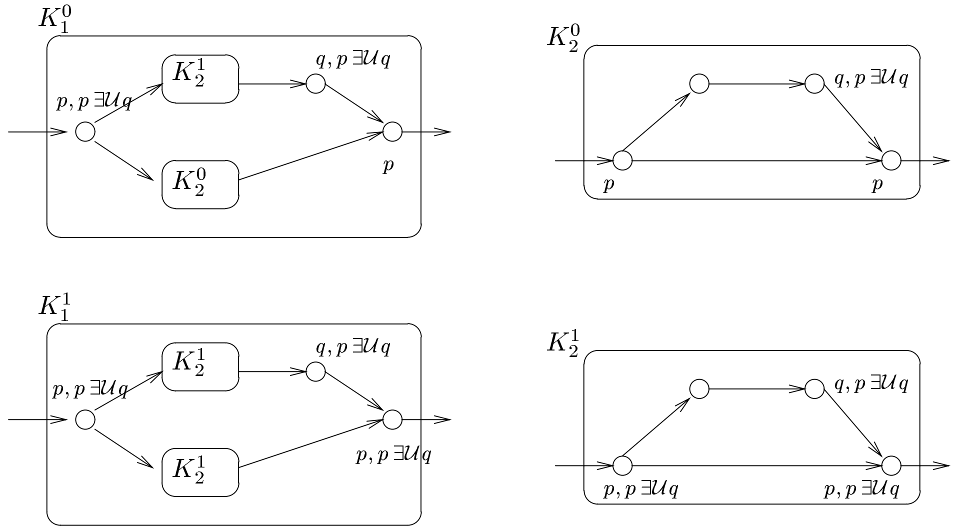

Fig. 11. Illustration of phase 2 of CheckUntil.

as it satisfies $(p \vee \text{MAYBE})\,\exists\mathcal{U}(q \vee \text{YES} \vee (out_2 \wedge p))$ (see rule (4) above). The labeling of nodes with $p\,\exists\mathcal{U}q$ in the two copies of K_1 is determined analogously. The mapping of boxes to structures is determined according to rules 1 and 2 above. For the upper box, the formula $p\,\exists\mathcal{U}q$ holds upon exit in K_1, and hence in both the copies K_1^0 and K_1^1, the upper box gets mapped to K_2^1. For the lower box, the truth of the formula $p\,\exists\mathcal{U}q$ upon exit coincides with its truth at the exit-node of K_1, and hence, the lower box is mapped to K_2^0 in K_1^0 and to K_2^1 in K_1^1.

The construction of K' is immediate if we have computed, for each K_i, the sets

$$\{u \in N_i \cup B_i \mid u \models_i (\psi_1 \vee \text{MAYBE})\,\exists\mathcal{U}(\psi_2 \vee \text{YES})\}$$

and

$$\{u \in N_i \cup B_i \mid u \models_i (\psi_1 \vee \text{MAYBE})\,\exists\mathcal{U}(\psi_2 \vee \text{YES} \vee (out_i \wedge \psi_1))\}.$$

Since the nodes of K_i are already labeled with ψ_1 and ψ_2, and the boxes are labeled with YES, MAYBE, or NO, these two sets can be computed in time $O(|K|)$, as in the standard labeling algorithm of CTL [Clarke and Emerson 1981].

4.1.4 *CheckAlways: Processing of* $\exists\Box$. Finally, we consider the processing of subformulas of the type $\exists\Box\chi$. The input to *CheckAlways* is a hierarchical structure K and a formula χ. Let $K = \langle K_1, \ldots K_n \rangle$ with $K_i = \langle N_i, B_i, in_i, out_i, X_i, Y_i, E_i \rangle$, and let $\psi = \exists\Box\chi$. The computation proceeds in two phases as in case of *CheckUntil*.

The first phase partitions the index-set $\{1, \ldots n\}$ into three sets, YES, NO, and MAYBE, with the following interpretation. An index i belongs to YES when the entry-node in_i satisfies the always-formula ψ within the expanded-structure K_i^F. An index i, that does not belong to YES, belongs to MAYBE if within K_i^F there is a (finite) trajectory from in_i to out_i that contains only states labeled with χ.

YES:= $\emptyset$; NO:= $\emptyset$; MAYBE:= $\emptyset$;
for $i = n$ **downto** 1 **do**
  if $in_i \models_i [(\chi \vee \mathrm{MAYBE})\, \exists\mathcal{U}\, \mathrm{YES}] \ \vee\ \exists\Box(\chi \vee \mathrm{MAYBE})$
  then YES := YES $\cup \{i\}$
  else
    if $in_i \models_i (\chi \vee \mathrm{MAYBE})\, \exists\mathcal{U}(out_i \wedge \chi)$
    then MAYBE := MAYBE $\cup \{i\}$
    else NO := NO $\cup \{i\}$
    fi
  fi
od .

Fig. 12. First phase of CheckAlways.

Remaining indices belong to NO. As before, define the satisfaction relation $\models_i$ that considers the structure K_i to be an ordinary structure over vertices $N_i \cup B_i$. The desired partitioning is computed by the procedure of Figure 12.

In the second phase, we construct two copies of each K_i. Nodes within K_i^1 can satisfy ψ along a path that exits K_i, while nodes within K_i^0 can satisfy ψ only if they satisfy it within K_i. The new structure $K' = \langle K_1^0, K_1^1, \ldots K_n^0, K_n^1 \rangle$ is obtained by replacing each K_i of K with two structures $K_i^0 = \langle N_i, B_i, in_i, out_i, X_i^0, Y_i^0, E_i \rangle$ and $K_i^1 = \langle N_i, B_i, in_i, out_i, X_i^1, Y_i^1, E_i \rangle$, where the new components are defined as follows:

(1) For the indexing of boxes of K_i^0, consider a box b with $Y_i(b) = j$. If $b \models_i \exists\bigcirc\, [(\chi \vee \mathrm{MAYBE})\, \exists\mathcal{U}\, \mathrm{YES} \ \vee\ \exists\Box(\chi \vee \mathrm{MAYBE})]$, then $Y_i^0(b) = 2j$; else $Y_i^0(b) = 2j - 1$.
(2) For the indexing of boxes of K_i^1, consider a box b with $Y_i(b) = j$. If $b \models_i \exists\bigcirc\, [(\chi \vee \mathrm{MAYBE})\, \exists\mathcal{U}(\mathrm{YES} \vee (out_i \wedge \chi)) \ \vee\ \exists\Box(\chi \vee \mathrm{MAYBE})]$, then $Y_i^1(b) = 2j$; else $Y_i^1(b) = 2j - 1$.
(3) For labeling of a node u of K_i^0, if $u \models_i \exists\bigcirc\, [(\chi \vee \mathrm{MAYBE})\, \exists\mathcal{U}\, \mathrm{YES} \ \vee\ \exists\Box(\chi \vee \mathrm{MAYBE})]$, then $X_i^0(u)$ equals $X_i(u)$ with ψ added to it; else it equals $X_i(u)$.
(4) For labeling of a node u of K_i^1, if $u \models_i \exists\bigcirc\, [(\chi \vee \mathrm{MAYBE})\, \exists\mathcal{U}(\mathrm{YES} \vee (out_i \wedge \chi)) \ \vee\ \exists\Box(\chi \vee \mathrm{MAYBE})]$, then $X_i^1(u)$ equals $X_i(u)$ with ψ added to it; else it equals $X_i(u)$.

The structure K' can be computed from K in time $O(|K|)$ using the standard CTL model-checking algorithm. This leads us to the following theorem:

THEOREM 8. (CTL MODEL CHECKING: SINGLE EXIT). *The algorithm of Figure 7 solves the CTL model-checking problem (K, φ), for a single-exit hierarchical structure K, in time $O(|K| \cdot 2^{|\varphi|})$.*

PROOF. For correctness, let K_ψ be the structure after processing of the subformula ψ. We claim, that for every node u that is reachable from the entry-node of the top-level structure of K_ψ, if u is labeled with ψ then in the expanded structure K_ψ^F *every* reachable state with the last component u satisfies ψ, and if u is not labeled with ψ then *no* reachable state with the last component u satisfies ψ. This can be established by induction on the length of ψ from the definitions of the temporal operators and the constructions in the algorithm. Note that the restriction to *reachable* states is needed. For instance, while processing an until-formula

$\psi = \psi_1 \, \exists\mathcal{U}\psi_2$, the nodes in the copy K_i^1 are labeled assuming that ψ is satisfied upon exit. The consistency of this assumption is verified for any box mapped to K_i^1. If no box is mapped to K_i^1, then there may be inconsistency, but in this case, nodes in K_i^1 are not reachable from the top-level entry-node.

A CTL-formula φ has at most $|\varphi|$ subformulas. Processing of each subformula requires time linear in the size of the current structure, as discussed in different cases. Processing each temporal subformula at worst doubles the size of the current structure. This leads to the complexity bound of $O(|K| \cdot 2^{|\varphi|})$, where $|K|$ is the size of the input structure. □

4.1.5 *Lower Bound.* It is known that deciding whether a flat Kripke structure M satisfies a CTL formula φ can be solved in space $O(|\varphi| \cdot log\ |M|)$ [Bernholtz et al. 1994]. For a hierarchical structure K, the size of the expanded structure K^F is $O(|K|^{nd(K)})$. The expanded structure need not be constructed explicitly. Each state of the expanded structure can be represented in space $O(nd(K) \cdot |K|)$. The number of successors of a state of K^F is only polynomial in the size of K: the number of successors of an expanded state whose last component is the node x of the structure K_i equals the number of successors of x if x is not an exit-node, and is bounded by the product of the number of edges in K and the number of boxes mapped to the index i, if x is an exit-node. The successors of any state of the expanded structure can be computed in space polynomial in the size of K. It follows that the space-efficient algorithm of [Bernholtz et al. 1994] requires space polynomial in the size of K, and consequently, CTL model checking for hierarchical structures is in Pspace. We now establish a Pspace lower bound for the case of single-exit structures.

THEOREM 9. (CTL COMPLEXITY: SINGLE EXIT). *Model checking of CTL formulas for single-exit hierarchical structures is Pspace-complete.*

PROOF. We establish Pspace-hardness by reduction from quantified boolean formulas. Consider a quantified formula $\phi = Q_1 x_1 . Q_2 x_2 . \ldots Q_n x_n . \psi(x_1, \ldots x_n)$, where ψ is a boolean formula over the boolean variables $x_1, \ldots x_n$, and each Q_i is a quantifier (universal or existential). Determining the truth of ϕ is Pspace-hard. Given such a formula ϕ, we construct a formula ϕ' of CTL, and a hierarchical structure K such that $K \models \phi'$ iff ϕ is true.

The formula ϕ' is defined to be the CTL-formula

$$Q_1 \bigcirc . Q_2 \bigcirc . \ldots Q_n \bigcirc . \psi(\exists\Diamond x_1, \exists\Diamond x_2 \ldots \exists\Diamond x_n).$$

That is, in the inside formula ψ, we replace each occurrence of the proposition x_i by the formula $\exists\Diamond x_i$, and in the quantifier prefix, we replace $\forall x_i$ by $\forall\bigcirc$ and $\exists x_i$ by $\exists\bigcirc$. The construction of the hierarchical structure K is illustrated in Figure 13. For $i = 1, \ldots n$, the structure K_i has one entry-node, one exit-node, and two boxes, $b_{i,0}$ and $b_{i,1}$, both mapped to K_{i+1}. The structure K_{n+1} has a single node. The entry-node of K_i has two successors. Intuitively, choosing $b_{i,1}$ corresponds to setting x_i to true, and the formula $\exists\Diamond x_i$ will be true at every subsequent node until K_i is exited. On the other hand, choosing $b_{i,0}$ corresponds to setting x_i to false, and the formula $\exists\Diamond x_i$ will be false at every subsequent node. Since we define our semantics with respect to only infinite trajectories, we add a self-loop on the exit-node of the structure K_1.

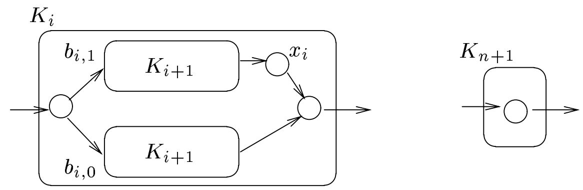

Fig. 13. Construction for Pspace complexity of Theorem 9.

More precisely, observe that the node of K_{n+1} appears in 2^n contexts. Each context is given by a tuple $(b_{1,j_1}, b_{2,j_2}, \ldots b_{n,j_n})$, where each j_i can be 0 or 1. In this context the node satisfies the formula $\psi(\exists\Diamond x_1, \exists\Diamond x_2 \ldots \exists\Diamond x_n)$ iff ψ evaluates to true under the interpretation that assigns each variable x_i to the value j_i. For $i < n$, the entry-node of each structure K_{i+1} appears in 2^i contexts, where each context given by $(b_{1,j_1}, \ldots b_{i,j_i})$ with each j_k being 0 or 1. In this context, the entry-node satisfies the formula $Q_i \bigcirc \ldots . Q_n \bigcirc .\psi(\exists\Diamond x_1, \exists\Diamond x_2 \ldots \exists\Diamond x_n)$ iff $Q_i x_i . \ldots Q_n x_n . \psi$ evaluates to true under the interpretation that assigns each free variable x_k, for $k < i$, to the truth value j_k. This claim can be proved by induction. $\Box$

4.2 Multiple-Exit Case

Now consider the case when the hierarchical structure K has multiple exits. The model-checking algorithm is similar to the algorithm for the single-exit case, except now more splitting may be required. For instance consider a structure K_i with 2 exit-nodes u and v, and a formula $\psi = \exists\bigcirc p$. For different boxes mapped to K_i, whether the exit-node u satisfies ψ can vary, and similarly, whether the exit-node v satisfies ψ can vary. Consequently, we need to split K_i into four copies, depending whether both, only u, only v, or none, have a ψ-successor outside K_i. In general, if there are d exit-nodes, processing of a single temporal subformula can generate 2^d copies of each structure in the worst case. The modifications required to the algorithm of Section 4.1 are left to the reader.

THEOREM 10. (CTL MODEL CHECKING). *The CTL model-checking problem (K, φ) can be solved in time $O(|K| \cdot 2^{|\varphi|d})$, where each structure of K has at most d exit-nodes.*

The alternative approach of applying known model-checking procedures to the expanded structure gives a time bound of $O(|\varphi| \cdot |K|^{nd(K)})$, or alternatively, a Pspace bound. Note that in practice the size of the system (structure) K is orders of magnitude greater than the size of the formula ϕ; thus it is much preferable to have the formula size in the exponent than to have the structure. In fact, typically formulas to be checked are quite small in size. The next result states that the lower bound of Pspace applies even for some small fixed CTL formula. This suggests that the appearance of the number d of exit-nodes in the exponent in the running time of Theorem 10 is unavoidable, and CTL model checking becomes indeed harder with multiple exit-nodes.

THEOREM 11. (CTL COMPLEXITY). *The structure-complexity of CTL model checking for hierarchical structures is Pspace-complete.*

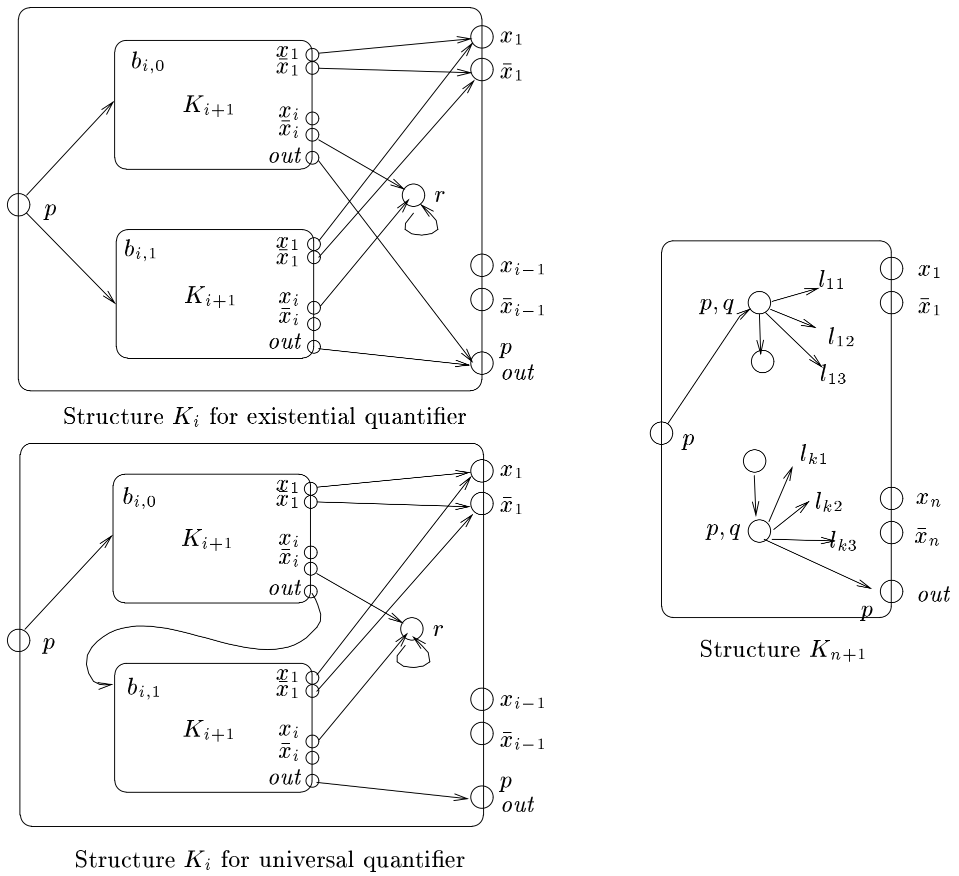

Fig. 14. Construction for Pspace complexity of Theorem 11.

PROOF. We again establish Pspace-hardness by reduction from quantified boolean formulas. Consider a quantified formula $\phi = Q_1x_1.Q_2x_2.\ldots Q_nx_n.\psi(x_1,\ldots x_n)$, where ψ is a boolean formula over the boolean variables $x_1,\ldots x_n$, and each Q_i is a quantifier (universal or existential). We assume that ψ is in conjunctive normal form, where each conjunct has 3 literals. That is, $\psi = \psi_1 \wedge \cdots \wedge \psi_k$, where each $\psi_j = l_{j1} \vee l_{j2} \vee l_{j3}$ such that each of the literals l_{j1}, l_{j2}, l_{j3} is either a boolean variable or its negation.

We build a hierarchical structure with $n+1$ substructures over 3 propositions p, q, and r (see Figure 14). The first n structures correspond to the n quantifiers and variables, while the last substructure K_{n+1} encodes the clauses of the formula ψ. For $i = 1,\ldots n$, each structure K_i has an entry-node, two boxes mapped to K_{i+1} (they are named $b_{i,0}$ and $b_{i,1}$ and correspond to setting variable x_i to 0 and to 1 respectively), one sink node, one exit-node called *out*, and $2(i-1)$ additional exit-nodes corresponding to the variables $x_1,\ldots x_{i-1}$ and their negations $\bar{x}_1,\ldots \bar{x}_{i-1}$.

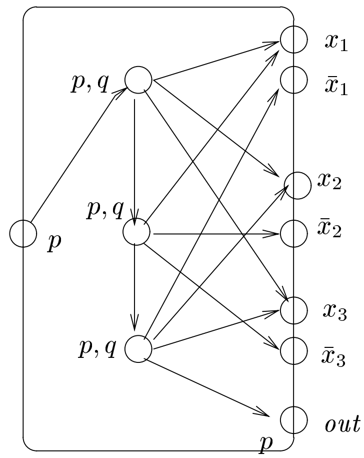

Fig. 15. Example of substructure K_{n+1}.

The entry-node and the exit-node *out* are labeled with (i.e. satisfy) p, while the sink node is labeled with r; the literal nodes x_j and $\bar{x}_j$, $j = 1, \ldots, i-1$ are not labeled with any proposition.

The edges of K_i, $i = 1, \ldots n$, are as shown in Figure 14, and depend on whether the quantifier Q_i is universal or existential. The edges incident to the exit-nodes x_j, $\bar{x}_j$ of the K_{i+1} boxes $b_{i,0}$, $b_{i,1}$ are the same in the two cases: for each $j = 1, \ldots, i-1$, the exit-node x_j (respectively, $\bar{x}_j$) of each of the two K_{i+1} boxes has an edge to the exit-node x_j (respectively, $\bar{x}_j$) of K_i. The exit-nodes $\bar{x}_i$ of box $b_{i,0}$ and x_i of $b_{i,1}$ have edges to the sink node (the node labeled r in the figure), while the exit-nodes x_i of $b_{i,0}$ and $\bar{x}_i$ of box $b_{i,1}$ have no outgoing edges. The other edges depend on the quantifier. If Q_i is an existential quantifier, then the entry-node of K_i has edges to the entries of both boxes $b_{i,0}$ and $b_{i,1}$ and the exit-nodes *out* of both boxes have edges to the exit-node *out* of K_i. If Q_i is a universal quantifier, then the entry-node of K_i has an edge only to the entry-node of $b_{i,0}$, the exit-node *out* of $b_{i,0}$ has an edge to the entry-node of $b_{i,1}$ and the exit-node *out* of $b_{i,1}$ has an edge to the exit node *out* of K_i. In addition, the sink nodes labeled r of all K_i, and the exit-node *out* of structure K_1 have a self-loop (not shown in Figure 14).

The bottom structure K_{n+1} has an entry-node, a series of k nodes corresponding to the clauses ψ_j of ψ, an exit-node *out*, and $2n$ exit-nodes corresponding to all possible literals (variables and their negations). The entry-node and the exit-node *out* are labeled p, and the clause-nodes are labeled with p as well as q; the literal nodes are not labeled with any proposition. There is an edge from the entry-node to the first clause node and an edge from the last clause node to the *out* node. Furthermore, each node corresponding to a clause ψ_j has three edges connecting it to the exit-nodes corresponding to its 3 literals l_{j1}, l_{j2}, and l_{j3}. In Figure 15 we show an example of the substructure K_{n+1} for the formula $\psi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3)$.

Now consider the CTL formula

$$\psi' \ = \ \exists\Box\,[\,p \ \wedge \ (\,q \rightarrow \ \exists\bigcirc \ (\neg p \wedge \exists\Diamond\, r\,)\,)\,].$$

We claim that the entry-node of K_1 satisfies ψ' iff the given quantified formula ϕ holds. In words, the CTL formula states that there is a path π (starting from the entry-node of K_1) each of whose nodes satisfies p, and each node of the path that is labeled q has a successor which does not satisfy p and which can reach a node labeled r. Note that the only nodes labeled p are the entry-nodes of the substructures, the exits nodes *out*, and the nodes of K_{n+1} corresponding to the clauses. Thus, a path π, along which p holds continuously, exits each structure K_i at the exit-node marked *out*. Each structure K_i has two occurrences of K_{i+1}: one box $b_{i,0}$ corresponding to setting x_i to false, and one box $b_{i,1}$ corresponding to setting x_i to true. The edges connecting *out* nodes ensure, that when x_i is universally quantified, both boxes must be entered by the path π, and when x_i is existentially quantified, only one box is entered.

Suppose that the given quantified formula ϕ is true. Then the required path π starts at the entry-node of K_1 and can be formed incrementally as follows. Assume that the path π is at the entry-node of an instance of K_i. Note that this instance is nested inside boxes corresponding to the substructures K_t, with $t < i$. Let this context of nested boxes be represented by the sequence $(b_{1,j_1}, \ldots, b_{i-1,j_{i-1}})$ where each j_t is either 0 or 1. Suppose first that $i \leq n$, i.e., K_i corresponds to a variable x_i with quantifier Q_i. If Q_i is an existential quantifier, then choose a value for the variable x_i, given the values b_t for the variables x_t, for $t < i$, appropriately to satisfy the quantified formula ϕ, and let the path π proceed to the corresponding box $b_{i,0}$ or $b_{i,1}$ of K_i, exit at the *out* node of that box, and proceed to exit the structure K_i at the *out* node. If Q_i is a universal quantifier, then the path π proceeds to the box $b_{i,0}$, exits at its *out* node, proceeds to box $b_{i,1}$, exits at its *out* node, and then proceeds to exit K_i at the *out* node. Suppose that $i = n{+}1$, i.e., π enters an instance of the structure K_{n+1} in the context $(b_{1,j_1}, \ldots b_{n,j_n})$; then the context corresponds to an assignment for the variables that satisfies the formula ψ (from our choices in the path above for the existentially quantified variables). The path π traverses all the nodes that correspond to the clauses and exits the structure from node *out*. Clearly, all the nodes of the constructed path π satisfy p. The only nodes labeled q are the clause nodes in K_{n+1}. Since the variable assignment corresponding to the context satisfies all the clauses, each clause node has a successor l corresponding to a true literal, i.e., either the literal l is x_i and the context includes box $b_{i,1}$ (i.e., $j_i = 1$) or the literal l is $\bar{x}_i$ and the context includes box $b_{i,0}$. This successor node l satisfies $\neg p$ and can reach a node labeled r, by exiting each structure K_a, for $a \geq i$ at the exit-point corresponding to the same literal $l = x_i$ or $\bar{x}_i$; note, that because of the truth value of the variable x_i, the exit-node of the box b_{i,j_i} has an edge to the sink node of K_i that is labeled r.

Conversely, suppose that the CTL formula ψ' is satisfied. That is, there is an infinite path π all whose nodes satisfy p and every node labeled q has a successor that satisfies $\neg p \wedge \exists\Diamond r$. Since all nodes of π satisfy p, the path exits every box at the node *out*, and within structure K_{n+1} it traverses the nodes corresponding to the clauses. By the construction, whenever the path enters K_i, $i \leq n$, for an existential (respectively, universal) quantifier, it traverses one (respectively, both) of the boxes

	Ordinary Structure M	Hierarchical Structure K
Reachability	$O(\|M\|)$	$O(\|K\|)$
Automata-emptiness	$O(\|M\| \cdot \|A\|)$	$O(\|K\| \cdot \|A\|^3)$
LTL model checking	$O(\|M\| \cdot 2^{\|\varphi\|})$	$O(\|K\| \cdot 8^{\|\varphi\|})$
CTL model checking	$O(\|M\| \cdot \|\varphi\|)$	$O(\|K\| \cdot 2^{\|\varphi\|d})$

Table I. Summary of results.

corresponding to a truth assignment for x_i. Consider the traversal of the path π through an occurrence of the structure K_{n+1} in the context $(b_{1,j_1}, \dots b_{n,j_n})$, where each j_i is either 0 or 1 denoting the assignment to x_i. The proposition q holds at nodes corresponding to the clauses ψ_j. Each such node must satisfy $\exists\bigcirc(\neg p \wedge \exists\Diamond r)$, i.e., it has a successor which satisfies $\neg p$ and $\exists\Diamond r$. Hence the successor must be an exit-node corresponding to a literal $l = x_i$ or $\bar{x}_i$ of the clause. Since the node can reach a node labeled r, the literal l must be true in the context: note that the path following this literal node of K_{n+1} is completely forced; it goes through the corresponding literal exit-node l of the structures K_a, for $a \geq i$. If literal l was not true in the assignment corresponding to the context then the exit-node l of the K_i would not have any outgoing edge and thus would not reach r. It follows that the given quantified formula ϕ is true. $\square$

5. CONCLUSIONS

In this paper, we have established that verification of hierarchical machines can be done without flattening them first. Among the three popular extensions of state machines, communicating state machines, extended state machines, and hierarchical state machines, it is already known that the first two offer exponential succinctness at an exponential cost, but as our results show, hierarchical specifications offer exponential succinctness at a minimal price. We presented efficient algorithms, and matching lower bounds, for the model-checking problem for all the commonly used specification formalisms. Our results are summarized in Table I. For hierarchical structures, model checking of branching-time formulas seems more expensive than model checking of linear-time formulas. This difference is intuitively due to the different complexities of the *local* and *global* model-checking problems. In local model checking, we want to check whether some (or all) paths starting at a specified state (e.g., an initial state) satisfy a linear-time property, while in the global model checking, we want to compute all (reachable) states such that some (or all) paths starting at that state satisfy a linear-time property. For ordinary state machines, both local and global variants can be solved in time linear in the size of the structure (even though the local, or on-the-fly, algorithms for checking linear-time properties are preferred in practice). For hierarchical structures, the local variant can be solved efficiently by our algorithm that avoids repeated analysis of a shared substructure. The global variant is more expensive, as it requires splitting of a substructure because the satisfaction of formulas can vary from context to context. CTL model checking requires solving the global problem repeatedly, due to the nesting of path quantifiers in the formula.

Our results can be useful more generally in the analysis of hierarchical structures other than hierarchical state machines, i.e., in the analysis of structures that are defined hierarchically where the basic (unnested) nodes are other types of objects rather than simple states. From the given properties of the basic objects, our algorithms can be used to infer efficiently the properties of the hierarchical structure and of its executions. For example, a hierarchical MSC (HMSC) is just like a hierarchical state machine except that the basic objects are basic message sequence charts instead of states. It is shown in [Alur and Yannakakis 1999] how to use the algorithms of this paper to check, without flattening, an HMSC for properties such as boundedness of message buffers, divergence of processes, and for general LTL properties under a certain choice of semantics for the concatenation of message sequence charts; we refer the reader to [Alur and Yannakakis 1999] for more details.

In this paper, we have presented results about hierarchical structures in the absence of variables and concurrency. In the presence of variables, our algorithms can be adopted in a natural way by augmenting nodes with the values of the variables. In particular, suppose we have an extended hierarchical structure K with k boolean variables, and the edges have guards and assignments that read/write these variables. Then, we can construct a hierarchical structure of size at most $8^k \cdot |K|$: a blow-up of 2^k is immediate, as a node must be considered in combination with different values of the variables, and the additional blow-up arises from the facts that a box can be entered in 2^k possible ways, and translation from multi-entry case to single-entry case causes a factor quadratic in the number of entry-nodes. Thus, reachability problems for an extended hierarchical structure K with k boolean variables can be solved in $O(8^k \cdot |K|)$ (in contrast, reachability problems for an extended ordinary structure M with k boolean variables can be solved in $O(2^k \cdot |M|)$). In presence of concurrency, our algorithms are applicable in the case when one of the communicating systems is modeled as a hierarchical state machine and the rest are modeled as ordinary state machines, using the product construction of Section 3.3. If two or more systems are modeled as hierarchical machines, as recent results indicate [Alur et al. 1999], in the worst case, there is no better strategy than taking the product of flattened machines.

ACKNOWLEDGMENTS

We thank Kousha Etessami, Radu Grosu, and Sampath Kannan for fruitful discussions, and anonymous reviewers for careful comments.

REFERENCES

Alur, R. and Yannakakis, M. 1999. Model checking of message sequence charts. In *CONCUR'99: Concurrency Theory, Tenth International Conference*. LNCS 1664. Springer-Verlag, 114–129.

Alur, R., Grosu, R., and McDougall, M. 2000. Efficient reachability analysis of hierarchical reactive machines. In *Computer Aided Verification, 12th International Conference*. LNCS 1855. Springer, 280–295.

Alur, R., Kannan, S., and Yannakakis, M. 1999. Communicating hierarchical state machines. In *Automata, Languages and Programming, 26th International Colloquium*. Springer, 169–178.

Apfelbaum, L. 1995. Automated functional test generation. In *Proceedings of the IEEE Autotestcon Conference*.

Ball, T. and Rajamani, S. 2000. Bebop: A symbolic model checker for boolean programs. In *SPIN 2000 Workshop on Model Checking of Software*. LNCS 1885. Springer, 113–130.

Behrmann, G., Larsen, K., Andersen, H., Hulgaard, H., and Lind-Nielsen, J. 1999. Verification of hierarchical state/event systems using reusability and compositionality. In *TACAS '99:*

Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Software. LNCS 1579. Springer, 163–177.

BERNHOLTZ, O., VARDI, M., AND WOLPER, P. 1994. An automata-theoretic approach to branching-time model checking. In *Computer Aided Verification, Proc. 6th Int. Conference*. LNCS 818. Springer-Verlag, 142–155.

BOOCH, G., JACOBSON, I., AND RUMBAUGH, J. 1997. *Unified Modeling Language User Guide*. Addison Wesley.

BOUJJANI, A., ESPARZA, J., AND MALER, O. 1997. Reachability analysis of pushdown automata: Applications to model checking. In *CONCUR'97: Concurrency Theory, Eighth International Conference*. LNCS 1243. 135–150.

BURKART, O. AND STEFFEN, B. 1992. Model checking for context-free processes. In *CONCUR'92: Concurrency Theory, Third International Conference*. LNCS 630. 123–137.

CHAN, W., ANDERSON, R., BEAME, P., BURNS, S., MODUGNO, F., NOTKIN, D., AND REESE, J. 1998. Model checking large software specifications. *IEEE Transactions on Software Engineering 24,* 7, 498–519.

CLARKE, E. AND EMERSON, E. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*. LNCS 131. Springer-Verlag, 52–71.

CLARKE, E. AND KURSHAN, R. 1996. Computer-aided verification. *IEEE Spectrum 33,* 6, 61–67.

CLARKE, E. AND WING, J. 1996. Formal methods: State of the art and future directions. *ACM Computing Surveys 28,* 4, 626–643.

COURCOUBETIS, C., VARDI, M., WOLPER, P., AND YANNAKAKIS, M. 1992. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design 1,* 275–288.

DRUSINSKY, D. AND HAREL, D. 1994. On the power of bounded concurrency I: finite automata. *Journal of the ACM 41,* 3, 517–539.

FLORA-HOLMQUIST, A., O'GRADY, J., AND STASKAUSKAS, M. 1995. In *Proceedings of the International Switching Symposium*. 103–107.

HAREL, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8*, 231–274.

HOLZMANN, G. 1991. *Design and Validation of Computer Protocols*. Prentice-Hall.

HOLZMANN, G. 1997. The model checker SPIN. *IEEE Trans. on Software Engineering 23,* 5, 279–295.

HOLZMANN, G., PELED, D., AND REDBERG, M. 1997. Design tools for for requirements engineering. *Lucent Bell Labs Technical Journal 2,* 1, 86–95.

JACOBSON, I. 1992. *Object-oriented software engineering – A use case driven approach*. Addison-Wesley.

JAHANIAN, F. AND MOK, A. 1987. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers C-36,* 8, 961–975.

KURSHAN, R. 1994. *Computer-aided Verification of Coordinating Processes: the automata-theoretic approach*. Princeton University Press.

LEVESON, N., HEIMDAHL, M., HILDRETH, H., AND REESE, J. 1994. Requirements specification for process control systems. *IEEE Transactions on Software Engineering 20,* 9, 684–707.

LICHTENSTEIN, O. AND PNUELI, A. 1985. Checking that finite-state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*. 97–107.

MANNA, Z. AND PNUELI, A. 1991. *The temporal logic of reactive and concurrent systems: Specification*. Springer-verlag.

PNUELI, A. 1977. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*. 46–77.

QUEILLE, J. AND SIFAKIS, J. 1982. Specification and verification of concurrent programs in CESAR. In *Proceedings of the Fifth International Symposium on Programming*. LNCS 137. Springer-Verlag, 195–220.

RUDOLPH, E., GRAUBMANN, P., AND GABOWSKI, J. 1996. Tutorial on message sequence charts. *Computer Networks and ISDN Systems 28,* 12, 1629–1641.

RUMABAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. 1991. *Object-oriented Modeling and Design.* Prentice-Hall.

SELIC, B., GULLEKSON, G., AND WARD, P. 1994. *Real-time object oriented modeling and design.* J. Wiley.

THOMAS, W. 1990. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. B. Elsevier Science Publishers, 133–191.

VARDI, M. AND WOLPER, P. 1986. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic in Computer Science*. 332–344.

Received August 1999; revised December 2000; accepted April 2001