Incremental Model Checking in the Modal Mu-Calculus*

Oleg V. Sokolsky Scott A. Smolka
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400
{oleg,sas}@sbcs.sunysb.edu

Abstract. We present an incremental algorithm for model checking in the alternation-free fragment of the modal mu-calculus, the first incremental algorithm for model checking of which we are aware. The basis for our algorithm, which we call *MCI* (for Model Checking Incrementally), is a linear-time algorithm due to Cleaveland and Steffen that performs global (non-incremental) computation of fixed points. *MCI* takes as input a set Δ of *changes* to the labeled transition system under investigation, where a change constitutes an inserted or deleted transition; with virtually no additional cost, inserted and deleted states can also be accommodated. Like the Cleaveland-Steffen algorithm, *MCI* requires time linear in the size of the LTS in the worst case, but only time linear in Δ in the best case. We give several examples to illustrate *MCI* in action, and discuss its implementation in the Concurrency Factory, an interactive design environment for concurrent systems.

1 Introduction

The Concurrency Factory [CGL+94] is a joint project between the State University of New York at Stony Brook and North Carolina State University to develop an integrated toolset for the specification, verification, and implementation of concurrent and distributed systems. Like the Concurrency Workbench [CPS93], the Factory employs bisimulation, preorder, and model checking as its main avenues of analysis.

A major underlying goal of the project is that the Factory be suitable for industrial application. One manner in which we are striving to achieve such applicability is through the use of *incremental computation*, which is basically an attempt to avoid repeating lengthy analyses of a system specification after the specification has undergone some relatively minor change.

This current paper is concerned with the incrementalization of the model checking routine of the Concurrency Factory, or, more generally, *incremental model checking in the modal mu-calculus*. Model checking [CE81, CES86] is the process of verifying whether the system has a property specified by a formula in some temporal logic (in other words, provides a model for the formula). The modal mu-calculus [Koz83] is a highly expressive logic that can be used to specify safety and liveness properties of concurrent systems represented as labeled transition systems (LTSs). Our focus here is on the *alternation-free* fragment of the modal mu-calculus [EL86] which, intuitively, means that the "level" of mutually recursive greatest and least fixed-point operators is one.

* Research supported in part by NSF Grants CCR-9120995 and CCR-9208585, and AFOSR Grant F49620-93-1-0250DEF.

Our main result is an incremental algorithm for model checking in the alternation-free modal mu-calculus, which we call *MCI* (for Model Checking Incrementally). To our knowledge, *MCI* is the first incremental algorithm for model checking, of any logic, to be proposed in the literature. The basis for *MCI* is a linear-time algorithm due to Cleaveland and Steffen (henceforth referred to as the *CS* algorithm) which performs global (non-incremental) computation of fixed points.

MCI takes as input a set Δ of *changes* to the LTS under investigation. An element of Δ corresponds to an inserted or deleted transition, although with virtually no additional cost, inserted and deleted states can also be accommodated. Its output is a *variable assignment*, representing the desired fixed-point solution.

The main technique utilized by *MCI* is to first compute the immediate effects of Δ on the results of the previous computation and then restart the fixed-point iteration. As part of the correctness proof of *MCI*, we show that it is safe to restart the iterations only after making certain adjustments to the current variable assignment — raising it sufficiently high in the lattice of all variable assignments when computing greatest fixed points, and, dually, lowering it sufficiently when computing least fixed points.

The required adjustments to the variable assignment are realized by making certain *assumptions* about the connectivity of nodes in the *product graph*, a data structure capturing all dependencies between pairs of the form $\langle s, X_i \rangle$, for LTS state s and logical variable X_i. We show that it is the presence of strongly connected components in the product graph that leads to the existence of distinct greatest and least fixed point solutions. *MCI* later checks that the assumptions it made were correct, and undoes the effects of any that turned out to be invalid.

In terms of its computational complexity, *MCI*'s worst-case behavior is asymptotically the same as that of *CS*. This is to be expected for it is easy to construct an example in which the value of every variable changes as the result of adding a transition to the LTS. Thus, every node of the product graph must be visited during the incremental run. In fact we prove, via a reduction from SS-REACHABILITY (see [Ram93]), that model checking is an *unbounded* problem, meaning that the running time of an incremental update cannot, in general, be expressed solely in terms of Δ.

In the best case, however, *MCI* requires time linear only in the size of Δ, which is typically constant with respect to the size of the LTS. We show that *MCI* exhibits this kind of performance on an incremental computation involving Milner's scheduler [Mil80], an oft-used benchmark for verification tools.

The closest related work we are aware of is that of Ryder et al. [RMP88] which treats incremental solutions to graph problems in a very general setting. However, they only give sufficient conditions to ensure it is safe to restart iterations of the original algorithm after an incremental update. In practice, these conditions are very restrictive and, in general, an additional computation is needed before iterations can be safely restarted. An informal discussion of the role of cycles in fixed-point incremental computation on graphs is presented in [PS89].

The structure of the rest of the paper is as follows. Section 2 defines the syntax and semantics of the modal mu-calculus and the corresponding model checking problem. Section 3 contains our description of the *CS* algorithm, while Section 4 presents our *MCI* algorithm. Section 5 proves the correctness of *MCI* and analyzes its complexity. Section 6 discusses our implementation of *MCI* in the Concurrency Factory and illustrates the algorithm in action through examples. Finally, Section 7 concludes and outlines directions for future work.

2 Syntax and Semantics of the Modal Mu-Calculus

A *Labeled Transition System* (LTS) is a 4-tuple $\langle \mathcal{S}, Act, \rightarrow, s_0 \rangle$ where $\mathcal{S}$ is the set of *states*, *Act* is the set of *actions*, $\rightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$ is the *transition relation*, and s_0 is the *start state*.

We next give the syntax and semantics of a version of the alternation-free modal mu-calculus defined in [CS93], which we refer to as *CS-logic*. Formulas in CS-logic are of two types: basic formulas and equational blocks. The syntax of *basic formulas* is given by the following grammar:

$$\Phi ::= A \mid X \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid [a]\Phi \mid \langle a \rangle \Phi$$

where $A \in \mathcal{AP}$, a fixed set of atomic propositions, and $X \in Var$, a countably infinite set of variables.

Basic formulas are interpreted with respect to an LTS $\mathcal{L} = \langle \mathcal{S}, Act, \rightarrow, s_0 \rangle$, a *valuation mapping* $\mathcal{V} : \mathcal{AP} \rightarrow \mathcal{P}(\mathcal{S})$, relating every atomic proposition A to the set of states in which A holds, and an *environment* $e : Var \rightarrow \mathcal{P}(\mathcal{S})$, mapping each variable X to the set of states that satisfy X. For a fixed environment e, the meaning of basic formulas is given by the semantical function $[\![\cdot]\!]e : \Phi \rightarrow \mathcal{P}(\mathcal{S})$, defined in Figure 1.

An *equational block* B is formed by applying operator *min* or *max* to a set E of mutually recursive equations of the form

$$\begin{aligned} X_1 &= \Phi_1 \\ &\vdots \\ X_n &= \Phi_n, \end{aligned}$$

where each Φ_i is a basic formula and the X_i are pairwise distinct. Operators *min* and *max* are understood respectively as the least and greatest fixed points of E. Following [CS93], we assume that the Φ_i are *simple*, i.e., an atomic proposition, or constructed by the application of exactly one operator to variables. Every formula can be made simple with at most a linear blow-up in size.

Semantically blocks are understood as functions from environments to environments. Let a block B contain a set of equations E with variables $X_1, \ldots, X_n$ defined as left-hand sides. Let $\overline{S} = \langle S_1, \ldots, S_n \rangle \in (2^{\mathcal{S}})^n$ and let $e_{\overline{S}} = e[X_1 \mapsto S_1, \ldots, X_n \mapsto S_n]$. Then the function

$$f_E^e(\overline{S}) = \langle [\![\Phi_1]\!]e_{\overline{S}}, \ldots, [\![\Phi_n]\!]e_{\overline{S}} \rangle,$$

defined on the lattice of tuples of sets of states ordered by point-wise set inclusion is monotonic. By the Tarski-Knaster fixed-point theorem, f_E^e has both least and greatest fixed points given by:

$$\begin{aligned} \nu f_E^e &= \bigcup \{ \overline{S} \mid \overline{S} \subseteq f_E^e(\overline{S}) \} \\ \mu f_E^e &= \bigcap \{ \overline{S} \mid f_E^e(\overline{S}) \subseteq \overline{S} \} \end{aligned}$$

Blocks can now be interpreted in the following fashion:

$$\begin{aligned} [\![maxE]\!]e &= e_{\nu f_E^e} \\ [\![minE]\!]e &= e_{\mu f_E^e}. \end{aligned}$$

Finally, a *formula* $\mathcal{B} = \{B_1, \ldots, B_m\}$ is a set of blocks, with the following syntactic restrictions: all variables appearing on the left-hand sides in the set of blocks are distinct,

$$
\begin{aligned}
&[\![A]\!]e = \mathcal{V}(A)\\
&[\![X]\!]e = e(X)\\
&[\![\Phi_1 \wedge \Phi_2]\!]e = [\![\Phi_1]\!]e \cap [\![\Phi_2]\!]e\\
&[\![\Phi_1 \vee \Phi_2]\!]e = [\![\Phi_1]\!]e \cup [\![\Phi_2]\!]e\\
&[\![[a]\Phi]\!]e = \{s \mid \forall s'.s \xrightarrow{a} s' \Rightarrow s' \in [\![\Phi]\!]e\}\\
&[\![\langle a\rangle\Phi]\!]e = \{s \mid \exists s'.s \xrightarrow{a} s' \wedge s' \in [\![\Phi]\!]e\}
\end{aligned}
$$

Fig. 1. Semantics of basic formulas.

and the formula's block graph is acyclic. The *block graph* of $\mathcal{B}$ is the directed graph with nodes $B_1, \ldots, B_m$ and edges $\langle B_i, B_j\rangle$ whenever a variable appearing as a left-hand side of an equation in B_i is used in B_j (we say that B_j *depends* on B_i in this case). Restricting the block graph to be acyclic ensures that no alternating fixed points [EL86] can occur.

The meaning $[\![\mathcal{B}]\!]e$ of the formula $\mathcal{B}$ containing blocks $B_1, \ldots, B_m$, topologically sorted by the dependency relation, can be computed through a sequence of environments

$$
\begin{aligned}
e_1 &= [\![B_1]\!]e\\
&\vdots\\
e_m &= [\![B_m]\!]e_{m-1}
\end{aligned}
$$

with $[\![\mathcal{B}]\!]e = e_m$. Due to the acyclicity restriction on block graphs, we are ensured that $[\![\mathcal{B}]\!]e_m = e_m$.

If $\mathcal{B}$ is a closed formula, i.e., every variable mentioned in the right-hand side of some equation appears on the left-hand side of an equation in one of the blocks, then for every two environments e and e', we have $[\![\mathcal{B}]\!]e = [\![\mathcal{B}]\!]e'$. Now, for every variable X defined in the formula we can compute the set of states in which X holds as $[\![X]\!][\![\mathcal{B}]\!]$. When the LTS is finite-state, f_E^e is continuous and the fixed points also have iterative characterizations which are used by the *CS* and *MCI* algorithms to compute fixed points.

The problem of *model checking in CS-logic* can now be defined as follows: given an LTS $\mathcal{L} = \langle \mathcal{S}, A, \rightarrow, s_0\rangle$ and a CS-logic formula $\mathcal{B}$ with a designated variable X defined within it, determine whether $s_0 \in [\![X]\!][\![\mathcal{B}]\!]$.

3 The Cleaveland-Steffen Model Checking Algorithm

The *CS* algorithm performs (non-incremental) global computation of fixed-points, i.e., the value of every variable is computed in every state. Due to the acyclicity restriction on block graphs (see Section 2), computation can proceed block-by-block: once the fixed point of a block is computed, the variable assignments in that block can no longer change due to dependencies on other blocks. Blocks are processed in the order resulting from topologically sorting the block graph.

The *CS* algorithm, as well as our incremental algorithm, uses an elaborate set of data structures to achieve its linear running time. To simplify its presentation, we describe the *CS* algorithm in terms of an intuitive structure called the "product graph" (cf. boolean graphs in [And94]). For efficiency reasons, the product graph is not computed by the algorithm explicitly, although its construction would not affect the asymptotic complexity. The correspondence between the product-graph-based presentation and the original *CS* algorithm is straightforward.

The *product graph* of an LTS $\mathcal{L} = \langle \mathcal{S}, Act, \rightarrow, s_0 \rangle$ and a mu-calculus formula $\mathcal{B}$ is a directed graph with set of vertices $\{\langle s, X_i \rangle \mid s \in \mathcal{S}, X_i \in Var\}$ and set of edges given by the following rules:

- if $X_i = X_j \vee X_k$ or $X_i = X_j \wedge X_k$, then for every $s \in \mathcal{S}$, $\langle s, X_j \rangle \rightarrow \langle s, X_i \rangle$ and $\langle s, X_k \rangle \rightarrow \langle s, X_i \rangle$
- if $s \xrightarrow{a} s'$ and $X_i = \langle a \rangle X_j$ or $X_i = [a] X_j$, then $\langle s', X_j \rangle \rightarrow \langle s, X_i \rangle$.

If operator $\vee$ or $\langle a \rangle$ is used to define X_i, the node $\langle s, X_i \rangle$ is called an *or*-node of the product graph; otherwise, it is called an *and*-node. Note that the direction of edges is reversed compared to the LTS. The intuition for this comes from the fact that the truth of a variable in a node of the product graph is determined by truth of its immediate predecessors in the product graph, which, according to the semantics of the modal operators, is dependent on the immediate successors of the current state in the LTS.

For each node $\langle s, X_i \rangle$ of the product graph, the *CS* algorithm maintains the following variables:

- A boolean variable indicating whether or not variable X_i is true of state s in the current stage of the analysis. We simply use the name of the product graph node, i.e, $\langle s, X_i \rangle$, as the name of this variable, and sometimes refer to it as the *value* of the node. $\langle s, X_i \rangle$ is initialized to true if X_i is defined in a *max*-block, and, dually, is initialized to false if X_i is defined in a *min* block (we refer to these initializations as *trivial*), with the following exceptions:
  - The right-hand side of the equation for X_i is an atomic proposition A. Then $\langle s, X_i \rangle =$ true if $s \in \mathcal{V}(A)$, and false otherwise.
  - If state s has no a-derivatives and X_i is defined by $\langle a \rangle X_j$, then $\langle s, X_i \rangle =$ false, and if X_i is defined by $[a] X_j$, then $\langle s, X_i \rangle =$ true.
- A counter $C_{\langle s, X_i \rangle}$ that keeps track of the immediate predecessors of $\langle s, X_i \rangle$ in the product graph: if X_i is defined in a *max*-block and $\langle s, X_i \rangle$ is an *or*-node, then $C_{\langle s, X_i \rangle}$ records how many immediate predecessors of $\langle s, X_i \rangle$ are currently true. $C_{\langle s, X_i \rangle}$ is used dually in the case that X_i is defined a *min*-block and $\langle s, X_i \rangle$ is an *and*-node; i.e., it records how many immediate predecessors are currently false. In either case, $C_{\langle s, X_i \rangle}$ is initialized to the number of immediate predecessors of $\langle s, X_i \rangle$ in the product graph. These are the only cases in which counters are used.

Also, for every block B_j of the formula, a list M_j of nodes of the product graph is maintained, such that $\langle s, X_i \rangle$ is in M_j if X_i is defined in B_j, $\langle s, X_i \rangle$ recently changed its value, and the effect of this change on other nodes has yet to be determined. Initially, M_j contains all nodes $\langle s, X_i \rangle$ that were initialized non-trivially (see above).

The *CS* algorithm is captured by the following procedure, where $\mathcal{L}$ is a finite-state LTS and $\mathcal{B}$ is a CS-logic formula.

procedure $CS(\mathcal{L}, \mathcal{B})$
  topologically sort the blocks of $\mathcal{B}$
  initialize the $\langle s, X_i \rangle$, $C_{\langle s, X_i \rangle}$, and M_j as described above
  for each $B_j \in \mathcal{B}$ in topological order **do**
    if B_j is of type *max* **then** $MAX(B_j)$
    else $MIN(B_j)$

Procedure MAX, invoked on a block B_j, proceeds as follows:

procedure $MAX(B_j)$
  while M_j not empty **do**
    delete some $\langle s, X_i \rangle$ from M_j
    $DOWN(\langle s, X_i \rangle)$
  for each $\langle s, X_i \rangle$ such that X_i is defined in B_j and $\langle s, X_i \rangle$ is true **do**
    $UP(\langle s, X_i \rangle)$

Procedure *MIN* is dual to it, with all occurrences of *UP* and *DOWN*, as well as true and false, interchanged. *MAX* first propagates the changes to the variable assignment for block B_j recorded in M_j by repeatedly calling the procedure *DOWN* (given below). When M_j is finally empty, the greatest fixed point for B_j will have been computed. *MAX* then invokes *UP* on each variable $\langle s, X_i \rangle$ such that X_i is defined in B_j and $\langle s, X_i \rangle$ was *not* falsified during the preceding while loop. This is necessary because there may exist variables $\langle s', X_j \rangle$ in *min*-blocks, trivially initialized to false and dependent on $\langle s, X_i \rangle$, whose values should now be true. The calls to *UP* will produce the desired effect.

Procedure *DOWN* takes as a parameter a product graph node that has just changed its value from true to false, and checks whether any of its successors are affected by the change.[2]

procedure $DOWN(\ \langle s, X_i \rangle\ )$:
  for each *or*-node $\langle s', X_j \rangle$ such that $\langle s, X_i \rangle \rightarrow \langle s', X_j \rangle$ and $\langle s', X_j \rangle$ is true **do**
    decrement $C_{\langle s', X_j \rangle}$ by 1
    if $C_{\langle s', X_j \rangle} = 0$, **then**
      $\langle s', X_j \rangle :=$ false
      add $\langle s', X_j \rangle$ to M_k /* X_j is defined in B_k */
  for each *and*-node $\langle s', X_j \rangle$ such that $\langle s, X_i \rangle \rightarrow \langle s', X_j \rangle$ and $\langle s', X_j \rangle$ is true **do**
    $\langle s', X_j \rangle :=$ false
    add $\langle s', X_j \rangle$ to M_k /* X_j is defined in B_k */

The name of the procedure stresses the fact that the function δ on the environment (variable assignment) computed by *DOWN* satisfies $\delta(e) \leq e$; i.e., the resulting assignment moves *down* in the lattice of tuples of sets of states described in Section 2. Procedure *UP* is dual to *DOWN* and can be obtained by syntactically interchanging all occurrences of *and* and *or*, and true and false.

4 The Incremental Model Checking Algorithm

In this section, we modify the *CS* algorithm to obtain our *MCI* algorithm. *MCI* works incrementally in the following sense: Let $\mathcal{L}$ and $\mathcal{B}$ constitute a given instance of the model checking problem, and assume that the desired variable assignment has been previously computed (say, by an application of *CS*). Given a set Δ of *changes* to the LTS, where a change may correspond to either an inserted or deleted transition, *MCI* computes the new variable assignment by judiciously using the previously computed one as the "starting point" of the computation.

[2] Note that the change to $\langle s, X_i \rangle$ will affect $\langle s', X_j \rangle$ only if $\langle s, X_i \rangle$ is an immediate predecessor of $\langle s', X_j \rangle$ in the product graph and $\langle s', X_j \rangle$ was true. Collectively, these two conditions, and the fact that the block graph is acyclic, ensure that B_k, the block in which X_j is defined, is a *max*-block, as desired. See [CS93] for further details.

The top-level structure of MCI is basically the same as in CS: an initialization phase, in which the immediate effects of Δ on the previously computed variable assignment are ascertained, is followed by a for-loop in which blocks are processed in topological order of the block graph by calling modified MAX and MIN procedures. In the incremental case, however, it is necessary to start off the fixed-point computation of a block by making certain adjustments to the current variable assignment to ensure that the proper fixed point is computed.

As discussed in greater detail below, the adjustments will raise the variable assignment in the case of a *max*-block (by calling a modified UP procedure), and lower it in the case of a *min*-block (by calling a modified $DOWN$ procedure). Like before, iterations of the fixed-point computation will then lower the assignment in the case of a *max*-block, and raise it in the case of a *min*-block.

Since, for either type of block, we may need to shift the variable assignment up or down the lattice, it is no longer sufficient to provide procedures MAX and MIN, invoked on a block B_j, a single "work list" M_j, as in the non-incremental case. Rather, two such lists are now required, for both types of blocks: $down_j$, which records variables that change their values from true to false, and up_j, containing variables that change their values from false to true. Moreover, every product graph *and*- and *or*-node now has an associated counter, regardless of the type of the block they are defined in.

The initialization phase uses Δ to update both the product graph and the variable assignment of the previous computation.[3] Changes to the product graph reflect the semantics of basic formulas, in particular, the modal operators (the insertion and deletion of LTS transitions has no immediate effect on basic formulas constructed out of logical operators). When a transition $s \xrightarrow{a} s'$ is inserted into the LTS, for every pair of variables X_i, X_j such that $X_j = [a]X_i$ or $X_j = \langle a \rangle X_i$, the edge $\langle s', X_i \rangle \rightarrow \langle s, X_j \rangle$ is inserted into the product graph. Conversely, when a transition is deleted from the LTS, the corresponding set of edges is deleted from the product graph.

In response to changes in the product graph, counters are updated as one would expect. If $\langle s', X_j \rangle$ is an *or*-node and $\langle s, X_i \rangle$ is true, then $C_{\langle s', X_j \rangle}$ is incremented by 1 if the edge $\langle s, X_i \rangle \rightarrow \langle s', X_j \rangle$ is inserted into the product graph, and is decremented by 1 if this edge is deleted. The situation is dual for *and*-nodes.

The following cases require changing the value of a variable as an immediate result of inserting or deleting a product graph edge, independent of the block type. In each case, assume that X_j is defined in block B_k.

- An edge $\langle s, X_i \rangle \rightarrow \langle s', X_j \rangle$ such that $\langle s', X_j \rangle$ is an *and*-node is added to the product graph. If $\langle s', X_j \rangle$ is true and $\langle s, X_i \rangle$ is false, then $\langle s', X_j \rangle$ is changed to false and $\langle s', X_j \rangle$ is added to $down_k$.
- An edge $\langle s, X_i \rangle \rightarrow \langle s', X_j \rangle$ such that $\langle s', X_j \rangle$ is an *or*-node is added to the product graph. If $\langle s', X_j \rangle$ is false and $\langle s, X_i \rangle$ is true, then $\langle s', X_j \rangle$ is changed to true and $\langle s', X_j \rangle$ is added to up_k.
- An edge $\langle s, X_i \rangle \rightarrow \langle s', X_j \rangle$ such that $\langle s', X_j \rangle$ is an *and*-node is deleted from the product graph. If $\langle s, X_i \rangle$ is false and the node $\langle s', X_j \rangle$ had only one false predecessor (this number is recorded in $C_{\langle s', X_j \rangle}$), then $\langle s', X_j \rangle$ is changed to true and $\langle s', X_j \rangle$ is added to up_k.
- An edge $\langle s, X_i \rangle \rightarrow \langle s', X_j \rangle$ such that $\langle s', X_j \rangle$ is an *or*-node is deleted from the product graph. If $\langle s, X_i \rangle$ is true and the node $\langle s', X_j \rangle$ had only one true predecessor,

[3] Because MCI accepts as input a *set* of changes to the LTS, updates to the variable assignment made during initialization can potentially "overlap." Care needs to be taken to avoid unnecessary computations.

then $\langle s', X_j \rangle$ is changed to false and $\langle s', X_j \rangle$ is added to $down_k$.

The initializations described above depend only on the semantics of basic formulas and are therefore independent of the type of fixed point being computed. Simply restarting the fixed-point iteration (e.g., calls to $DOWN$ in the case of a *max*-block) at this point would bring us to a fixed point, but not necessarily the required fixed point! Rather, we must conclude the initialization phase by making certain *assumptions* about the existence of strongly connected components (SCCs) in the product graph. Assumptions made during initialization and their subsequent propagation through the product graph, will serve to adjust the variable assignment to a level where fixed-point iteration can be safely restarted.

To motivate our use of assumptions, consider the following scenario. The insertion of a transition in the LTS has resulted in the formation of a new SCC, call it C, in the product graph. Assume that C is contained in the subgraph of the product graph pertaining to a *max*-block B_j, and according to the results of the previous fixed-point computation, all nodes in C are false. Further assume that C is free of "external interference," that is, there is no edge $\langle s, X_i \rangle \rightarrow \langle s', X_j \rangle$ entering C such that the value of $\langle s, X_i \rangle$ uniquely determines the value of $\langle s', X_j \rangle$. For example, if $\langle s', X_j \rangle$ is an *and*-node and $\langle s, X_i \rangle$ is false, then $\langle s, X_i \rangle$ would be a source of external interference.[4] Then it is not difficult to see that the variable assignment in which all nodes in C are uniformly set to true or false is a fixed point. The point is, however, that the required fixed point for C is the largest one, i.e., the one in which all nodes are assigned the value true.

We will therefore, in general, assume that when an edge $\langle s, X_i \rangle \rightarrow \langle s', X_j \rangle$ is added to the product graph such that $\langle s, X_i \rangle$ and $\langle s', X_j \rangle$ are defined in the same *max*-block B_k and both are false, that a new SCC, free of external interference, has been created. We record this assumption by setting $\langle s, X_i \rangle$ to true and adding it to up_k and a new list called $assumptions_k$.[5] Note that the counter $C_{\langle s, X_i \rangle}$ is not updated to reflect $\langle s, X_i \rangle$'s new value and thus an inconsistency is introduced. This is intentional and will be used later to determine whether the assumption was a valid one.

The case of a *min*-block is dual: when an edge is added between two true nodes corresponding to variables defined in the same *min*-block, we change one of them to false and update the block's *assumptions* list. The changed value is reflected in the *down* list.

Deleted edges can also cause us to make assumptions during initialization. Suppose that a *max*-block SCC had only one source of external interference, which was eliminated when an edge was deleted from the product graph. The desired variable assignment in this case has all nodes in the SCC uniformly set to true. We therefore assume that whenever an edge $\langle s, X_i \rangle \rightarrow \langle s', X_j \rangle$ is deleted from the product graph such that X_j is defined in a *max*-block B_k and both nodes are false, $\langle s, X_i \rangle$ constituted the only source of external interference in the SCC containing $\langle s', X_j \rangle$. We record this assumption by setting $\langle s', X_j \rangle$ to true and adding it to both up_k and $assumptions_k$.

Consider now the propagation of the changes made to the variable assignment during initialization. These are recorded in up_j and $down_j$, for each block B_j. As before, blocks are processed in topological order of the block graph, by calling a modified MAX or MIN

[4] Because C is strongly connected, each node in C has at least one incoming edge and is therefore either an *and*- or *or*-node.

[5] A number of small optimizations can be made here to prevent obviously false assumptions, e.g. do not make the assumption if either of the nodes has no incoming edges or no outgoing edges, and thus cannot be on a cycle. We leave these to careful implementors.

procedure depending on the type of the block. MAX and MIN now commence with an *adjustment phase*, during which the variable assignment is shifted up (in the case of a *max* block) or down (in the case of a *min* block) in the lattice of assignments to ensure that the fixed-point computation can proceed normally. The new MAX procedure is given by:

```
procedure MAX(B_j)
  while up_j not empty do                      /* Adjustment Phase */
    delete some ⟨s, X_i⟩ from up_j
    UP(⟨s, X_i⟩)
  while assumptions_j not empty do             /* Check validity of assumptions */
    delete some ⟨s, X_i⟩ from assumptions_j
    if ⟨s, X_i⟩ is an and-node then
      if C_⟨s,X_i⟩ ≠ 0 then
        ⟨s, X_i⟩ := false
        add ⟨s, X_i⟩ to down_j
    if ⟨s, X_i⟩ is an or-node then
      if C_⟨s,X_i⟩ = 0 then
        ⟨s, X_i⟩ := false
        add ⟨s, X_i⟩ to down_j
  while down_j not empty do                    /* Iteration Phase */
    delete some ⟨s, X_i⟩ from down_j
    DOWN(⟨s, X_i⟩)
```

The adjustment phase makes as many variables true as possible by iteratively invoking UP. As shown in Section 5, the resulting variable assignment will be high enough in the lattice to contain every fixed point of the semantic function. At this point, the validity of any previously made assumptions is determined by checking whether the value of $\langle s, X_i\rangle$ is consistent with $C_{\langle s,X_i\rangle}$, for each node $\langle s, X_i\rangle$ on the list of assumptions. If an inconsistency is detected, i.e., according to $C_{\langle s,X_i\rangle}$, $\langle s, X_i\rangle$ should be false, we reset the variable and let $DOWN$ undo the effects of the assumption. When finished, the fixed-point iteration (applications of $DOWN$) can be safely restarted. Procedure $DOWN$ is modified as follows:

```
procedure DOWN( ⟨s, X_i⟩ ):
  for each or-node ⟨s', X_j⟩ such that ⟨s, X_i⟩ → ⟨s', X_j⟩, X_j defined in B_k do
    decrement C_⟨s',X_j⟩ by 1
    if ⟨s', X_j⟩ = true and C_⟨s',X_j⟩ = 0 then
      ⟨s', X_j⟩ := false
      add ⟨s', X_j⟩ to down_k   /* X_j is defined in B_k */
    else if C_⟨s',X_j⟩ ≠ 0, ⟨s', X_j⟩ = true and B_k is a min-block then
      ⟨s', X_j⟩ := false
      add ⟨s', X_j⟩ to down_k   /* X_j is defined in B_k */
      add ⟨s', X_j⟩ to assumptions_k
  for each and-node ⟨s', X_j⟩ such that ⟨s, X_i⟩ → ⟨s', X_j⟩, X_j defined in B_k do
    increment C_⟨s',X_j⟩ by 1
    if ⟨s', X_j⟩ = true then
      ⟨s', X_j⟩ := false
      add ⟨s', X_j⟩ to down_k   /* X_j is defined in B_k */
```

The overall structure of $DOWN$ is retained, except that counters are updated in both cases and an assumption is made when $DOWN$ encounters a true variable in a *min*-block. The intuition for the assumption is somewhat different from the one for assumptions made at initialization. Here we are assuming that $\langle s', X_j \rangle$ is a part of an SCC and the change to $\langle s, X_i \rangle$ eliminated a source of external interference for the SCC. As before, procedures *MIN* and *UP* are dual to those given above and are obtained by interchanging all occurrences of *UP* and *DOWN*, *up* and *down*, and *and* and *or*.

So far we have assumed that the changes to the LTS only concern transitions. States, however, can be added and deleted with almost no extra effort. The basic idea is to assume that the variable assignment for an isolated state, i.e., one devoid of incident transitions, is known — it can be computed during the first, nonincremental run of the algorithm. During incremental runs, state additions are processed before any other changes by setting variables of the form $\langle s, X_i \rangle$, where s is a new state, in accordance to the variable assignment of an isolated state. The processing of inserted and deleted transitions can now proceed as before. For state deletions, we assume that any incident transitions are deleted as well.

5 Correctness and Complexity

The proof of correctness of the *MCI* algorithm is given by the following theorem.

Theorem 1. *Let $\mathcal{L}$ be an LTS, $\mathcal{B}$ a CS-logic formula, Δ a set of changes to $\mathcal{L}$ in the form of inserted and deleted transitions, and $\mathcal{L}'$ the LTS obtained by applying Δ to $\mathcal{L}$. Furthermore, let e be the variable assignment obtained by algorithm* CS *on input $\mathcal{L}$ and $\mathcal{B}$, and, similarly, let e' be the variable assignment obtained by* CS *on input $\mathcal{L}'$ and $\mathcal{B}$. Then* MCI*, using e as the initial variable assignment, terminates on input $\mathcal{L}$, $\mathcal{B}$, and Δ with variable assignment e'.*

Proof Sketch: Consider a block B in $\mathcal{B}$. Without loss of generality, assume that B is a *max*-block; the case where B is a *min*-block is completely dual and therefore omitted. For O an arbitrary topological order of $\mathcal{B}$'s block graph, the proof is by induction on the position of B in O and proceeds in two main steps.

We first show that when B's *up* list is empty, the current variable assignment is higher in the lattice of variable assignments than any fixed point of B's recursive equations. In particular, it contains B's greatest fixed point. For this purpose it is convenient to define B's *subgraph*, the subgraph of the product graph induced by the set of nodes $\{\langle s, X_i \rangle \mid X_i \text{ is defined in } B\}$.

The proof now proceeds by induction on the topological order of the strongly connected components of B's subgraph (this is well defined since the acyclicity of the block graph guarantees that every SCC appears within one block). That is, fix an SCC C and assume the result for any SCC having edges leading into C. There are two cases to consider depending on whether or not C is a *trivial* SCC (consisting of one node). For the case when C is non-trivial, we have to worry about cycles of false nodes in it. The details of the case analysis are omitted but the crucial point is showing that no such cycle can exist unless some of its nodes are uniquely determined by the values of nodes outside C.

Now that we have established that when the *up* list has been emptied the current variable assignment contains B's greatest fixed point, the second step of the proof basically coincides with the proof of correctness of the *CS* algorithm. That is, we show that the processing of entries in B's *down* list monotonically lowers the variable assignment, and, when the list is empty, the greatest fixed point will have been reached (see [CS93] for details). □

Consider now the computational complexity of the *MCI* algorithm. In the worst case, its complexity is the same as that of the *CS* algorithm: linear in the product of the size of the LTS and the size of the formula, where the size of the LTS is taken to be the total number of states and transitions, and the size of the formula is the total number of equations over all blocks. The proof is similar to that of [CS93], and is based on the fact that a product-graph node $\langle s, X_i \rangle$ can appear at most once in each list up_j and $down_j$, for X_i defined in block B_j. We ensure this property by checking (in constant time) if a node is already present in a list, before attempting to add it to the list. Thus *UP* and *DOWN* can only be invoked on a node at most once each, and each such invocation traverses each outgoing edge once.

We have also shown that the problem of model checking in the alternation-free fragment of the modal mu-calculus falls into the category of *unbounded* problems, i.e., the running time of an incremental update cannot be expressed solely in terms of the size of the change to the input. The proof of the unboundedness of the model checking problem is via a reduction from the single-source reachability problem (SS-REACHABILITY): given a directed graph (V, E) and a fixed vertex $s \in V$, determine, for every vertex $v \in V$, whether v is reachable from s.

In [RR91], it is shown that SS-REACHABILITY is unbounded in the *locally persistent* model of computation [AHR+90], which, intuitively, comprises all incremental algorithms in which no global information is maintained between updates. It is straightforward to show that *MCI* is locally persistent, and it thus follows that the performance of the algorithm is the best one could hope for in an incremental setting.

6 Implementation and Examples

The *MCI* algorithm has been implemented as part of the Concurrency Factory project. We started with the implementation of *CS*, which we later modified to make use of incremental computation. Although the non-incremental version is still needed for the initial computation of fixed points, we were able to avoid unnecessary duplication of code. In particular, with only minor changes, the incremental versions of *UP* and *DOWN* produce correct results in the initial computation.

We now consider an example of *MCI* in action, which is intended to demonstrate the best-case behavior of the algorithm. The assumptions concerning the design process, however, seem to be realistic. The system in question is Milner's scheduler [Mil80], consisting of a circular chain of simple "cycler" processes $C_0, \ldots, C_{n-1}$. Milner's scheduler is often used as a benchmark for verification tools, partly because its state space grows exponentially with the size of the scheduler (number of cyclers).

Each C_i is initiated by the previous one in the chain by means of action g_i, after which it carries out the sequence of observable actions a_i, b_i "in parallel" with initiating C_{i+1}. The LTS for C_i is depicted in Figure 2. C_0 must be furnished with a transition labeled by action $\overline{start}$ (the dashed line in Figure 2) that allows it to be initiated by a separate starter process.

Imagine that the designer has completed the scheduler and checks it for the absence of deadlocks. The property "there is a reachable deadlocked state" is expressed by X_1 in the following *min*-block, where '$-$' stands for "any action:"

$$
\begin{aligned}
min\{\ & X_1 = X_2 \vee X_3 \\
& X_2 = \langle - \rangle X_1 \\
& X_3 = [-] X_4 \\
& X_4 = ff \qquad \}.
\end{aligned}
$$

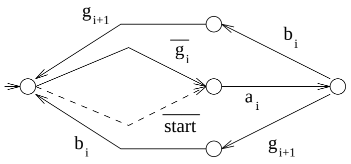

Fig. 2. The LTS for cycler C_i

The scheduler is correct and thus the formula is not satisfied. Imagine now that the designer, in an attempt to simplify the implementation, decides to omit the $\overline{start}$ transition from C_0 and then checks the scheduler again. The scheduler is now deadlocked in the start state, which disconnects the start state from the rest of the scheduler.

The second, incremental run of *MCI* finds the deadlock. Intuitively, the effects of the update to C_0 should not propagate very far, and Table 1 reveals this to indeed be the case. There, three rows of results are presented: (1) execution times of our implementation of *MCI* on the original, deadlock-free scheduler for increasing numbers of cyclers; (2) execution times for incremental runs of *MCI* on the updated, deadlocked scheduler; and (3) execution times of our implementation of *CS* on the original scheduler. The second row shows that the verification of the updated scheduler can be performed incrementally in *constant time*, independent of the number of cyclers. The third row allows us to compare how the *MCI* and *CS* algorithms perform on the first, necessarily non-incremental verification of the scheduler. As can be seen, the difference in execution times is negligible and, thus, the extra information we maintain in the incremental case does not significantly affect the constant factors of the algorithm.

We have also considered an example of the worst-case behavior of the incremental computation, involving a linear chain of transitions which we again test for deadlock. The update to the LTS is to extend the chain with yet another transition. Obviously, the LTS still has a deadlock, but the assumption made during the initialization phase of the incremental computation results in a wave of changes to the values of the variables that reaches the start state. At this point, it is determined that the assumption was wrong, so the wave of changes reverses direction and traverses the whole length of the chain again. The second application of *MCI* ends up taking about 75% more time than the first application.

7 Conclusions and Future Work

We have modified the algorithm of [CS93] to obtain *MCI*, an incremental algorithm for model checking in the alternation-free fragment of the modal mu-calculus. *MCI* can be easily modified to handle various kinds of incremental updates to the logical formula, such as the inversion of logical and modal connectives (e.g., changing $[a]$ to $\langle a\rangle$). Such updates affect the semantics of basic formulas but not the strongly connected components of the product graph.

We believe that our results on the non-uniqueness of fixed points of functions on graphs and their applications to incremental computation have wider applicability than just model checking CS-logic. For example, the *MCI* algorithm can be easily generalized to perform (global) incremental evaluation of *boolean equation systems* [Lar92]. Furthermore, if we do not confine ourselves to boolean variables in each node of the graph, then it appears that a variety of graph problems can be accommodated, including those pertaining to data-flow analysis [Hec77] (the relationship between data-flow

No. of cyclers		2	3	4	5	6	7	8	9	10
incr.	1st pass	0.01	0.02	0.05	0.19	0.48	1.36	3.06	7.52	18.90
	2nd pass	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
non-incr.		0.01	0.02	0.07	0.19	0.47	1.22	3.07	7.50	18.65

Table 1. Execution times, in seconds, for the scheduler example

analysis and model checking is investigated in [Ste91]).

Other directions for future work include the pursuit of incremental algorithms for model checking in the full modal mu-calculus [EL86] and for local model checking [SW91].

References

[AHR+90] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. "Incremental Evaluation of Computational Circuits". In *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990.

[And94] H. R. Andersen. "Model Checking and Boolean Graphs". *TCS*, 126(1), 1994.

[CE81] E. M. Clarke and E. A. Emerson. *Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic.* LNCS 131, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications". *ACM TOPLAS*, 8(2), 1986.

[CGL+94] R. Cleaveland, J. N. Gada, P. M. Lewis, S. A. Smolka, O. V. Sokolsky, and S. Zhang. "The Concurrency Factory – Practical Tools for Specification, Simulation, Verification and Implementation of Concurrent Systems". In *Proceedings of the DIMACS Workshop on Specification Techniques for Concurrent Systems, Princeton, NJ.*, May 1994.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. "The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems". *ACM TOPLAS*, 15(1), 1993.

[CS93] R. Cleaveland and B. Steffen. "A Linear-Time Model Checking Algorithm for the Alternation-Free Modal Mu-Calculus". *Formal Methods in System Design*, 2, 1993.

[EL86] E. A. Emerson and C.-L. Lei. "Efficient Model Checking in Fragments of the Propositional Mu-Calculus". In *Proceedings LICS '86.* IEEE Computer Society Press, 1986.

[Hec77] S. M. Hecht. *Flow Analysis of Computer Programs.* Elsevier, North Holland, 1977.

[Koz83] D. Kozen. "Results on the Propositional Mu-Calculus". *Theoretical Computer Science*, 27:333–354, 1983.

[Lar92] K. G. Larsen. "Efficient Local Correctness Checking". In *Proceedings of CAV'92*, 1992.

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

[PS89] L. L. Pollock and M. L. Soffa. "An Incremental Version of Iterative Data Flow Analysis". *IEEE Trans. Software Engineering*, 15(12), 1989.

[Ram93] G. Ramalingam. *Bounded Incremental Computation.* PhD thesis, Computer Sciences Dept., University of Wisconsin-Madison, 1993.

[RMP88] B. G. Ryder, T. J. Marlowe, and M. C. Paull. "Conditions for Incremental Iteration: Examples and Counterexamples". *Sci. Program.*, 11(1), 1988.

[RR91] G. Ramalingam and T. Reps. "On the Computational Complexity of Incremental Algorithms". Technical Report TR-1033, Computer Sciences Dept., University of Wisconsin-Madison, 1991.

[Ste91] B. Steffen. "Data Flow Analysis as Model Checking". In *Proc. TACS'91.* LNCS 526, 1991.

[SW91] C. Stirling and D. Walker. "Local Model Checking in the Modal Mu-Calculus". *Theoretical Computer Science*, 89(1), 1991.

This article was processed using the LaTeX macro package with LLNCS style