G_2. In addition, all states with location P that satisfy B have a transition into I_1 and those that do not satisfy B have a transition into I_2. Finally, states from $end(P_1)$ and $end(P_2)$ are unified with S_E. When G is a partition graph of $P =$ "while B do P_1 od", the structure of G contains the structure of G_1, and also transitions from I (states with location P) to I_1 if B holds and to S_E if B does not hold (and P terminates).

a structure $M = \langle S, R, I \rangle$ we define $M^{\uparrow P} = \langle S^{\uparrow P}, R^{\uparrow P}, I^{\uparrow P} \rangle$. For a set of structures $\mathcal{V}$ and a set of graph edges $\mathcal{E}$, $\mathcal{V}^{\uparrow P}$ and $\mathcal{E}^{\uparrow P}$ are obtained by putting in context each element in the set.

Definition A.3: A Partition Graph of a program P is a directed graph $G = (\mathcal{V}, \mathcal{E}, In, Out)$, where $\mathcal{V}$ is a finite set of nodes, each node V is a Kripke structure $\langle S_V, R_V, I_V \rangle$, $\mathcal{E}$ is a finite set of edges, $In \in \mathcal{V}$ is the entry node and $Out \in \mathcal{V}$ is the exit node. Each edge in $\mathcal{E}$ is either a null edge $M_1 \rightarrow M_2$, a yes-edge $B \stackrel{yes}{\rightarrow} M$ or a no-edge $B \stackrel{no}{\rightarrow} M$.

We use the following notations: $G_i = (\mathcal{V}_i, \mathcal{E}_i, In_i, Out_i)$ for any i, $S_P = \{(P, \sigma) \mid \sigma \in D^n\}$, $M_P = \langle S_P, \emptyset, S_P \rangle$, $S_E = \{(E, \sigma) \mid \sigma \in D^n\}$ and $M_E = \langle S_E, \emptyset, S_E \rangle$. The set $pg(P)$ of all Partition Graphs of P is defined inductively as follows [7] :

1. $(\{M\}, \emptyset, M, M) \in pg(P)$ where $M = struct(P)$.
2. If $P = P_1; P_2$ then for every $G_1 \in pg(P_1)$ and $G_2 \in pg(P_2)$, $G = (\mathcal{V}, \mathcal{E}, In, Out) \in pg(P)$ s.t.
   $\mathcal{V} = \mathcal{V}_1^{\uparrow P_2} \cup \mathcal{V}_2 \quad \mathcal{E} = \mathcal{E}_1^{\uparrow P_2} \cup \mathcal{E}_2 \cup \{Out_1^{\uparrow P_2} \rightarrow In_2\}$
   $In = In_1^{\uparrow P_2}, Out = Out_2$ (See figure 2).
3. If $P =$ "if B then P_1 else P_2 fi", then for every $G_1 \in pg(P_1)$ and $G_2 \in pg(P_2)$, $G = (\mathcal{V}, \mathcal{E}, In, Out) \in pg(P)$ s.t.
   $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2 \cup \{M_B, M_E\}$
   $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2 \cup \{M_B \stackrel{yes}{\rightarrow} In_1, M_B \stackrel{no}{\rightarrow} In_2, Out_1 \rightarrow M_E, Out_2 \rightarrow M_E\}$
   $In = M_B, Out = M_E$
4. If $P =$ "while B do P_1 od", then for all $G1 \in pg(P_1)$, $G = (\mathcal{V}, \mathcal{E}, In, Out) \in pg(P)$ s.t.
   $\mathcal{V} = \mathcal{V}_1 \cup \{M_B, M_E\} \quad \mathcal{E} = \mathcal{E}_1 \cup \{M_B \stackrel{yes}{\rightarrow} In_1, M_B \stackrel{no}{\rightarrow} M_E, Out_1 \rightarrow M_B\}$
   $In = M_B, Out = M_E$

The next step is to define the semantics of a partition graph, by means of one Kripke structure. The goal is that for any $G \in pg(P)$ the semantics of G and P will be the same.

Definition A.4: Given a step-edge $M_B \stackrel{yes}{\rightarrow} M_1$ (or $M_B \stackrel{no}{\rightarrow} M_1$) we define $step(M_B \stackrel{yes}{\rightarrow} M_1)$ to be the set of transitions induced by this edge. Assume that l is the location of the command that evaluates the boolean condition B, which means that l is the location of the states in M_B. Assume also that l' is the location of the states in $init(M_1)$. Then we define:
$step(M_B \stackrel{yes}{\rightarrow} M_1) = \{((l, \sigma), (l', \sigma)) \mid \sigma \models B\}$
$step(M_B \stackrel{no}{\rightarrow} M_1) = \{((l, \sigma), (l', \sigma)) \mid \sigma \not\models B\}$

Definition A.5: Let $G = (\mathcal{V}, \mathcal{E}, In, Out)$ be a partition graph of some program P, s.t. $\mathcal{V} = \{V_1, \ldots, V_n\}$ and for every $1 \leq i \leq n$, $V_i = \langle S_i, R_i, I_i \rangle$. $struct(G) = \langle S, R, I \rangle$ where [8]:
$S = \bigcup_{1 \leq i \leq n} S_i$,
$I = I_{In}$ (where I_{In} is the set of initial states of the entry node In of G) and
$R = (\bigcup_{1 \leq i \leq n} R_i) \cup (\bigcup_{V_i \stackrel{yes}{\rightarrow} V_j \in \mathcal{E}} step(V_i \stackrel{yes}{\rightarrow} V_j)) \cup (\bigcup_{V_i \stackrel{no}{\rightarrow} V_j \in \mathcal{E}} step(V_i \stackrel{no}{\rightarrow} V_j))$.

Lemma A.1: For any program P and graph $G \in pg(P)$, $struct(P) = struct(G)$.

Intuitively, when G is a partition graph of $P = P_1; P_2$, the structure of G is the union of the structures of G_1 and G_2, except that G_1 is "put in context" of P_2. This is done by using $S_1^{\uparrow P_2}$ and $R_1^{\uparrow P_2}$ instead of S_1 and R_1 when creating G. When G is a partition graph of $P =$ "if B then P_1 else P_2 fi", the structure of G contains the union of the structures of G_1 and

[7] All unions of nodes and edges are assumed to be disjoint unions, possibly requiring additional labels to differentiate between nodes or edges that happen to have the same name. We ignore the change in names whenever there is no doubt as to what we are referring to.

[8] In this definition, unions are *not* disjoint. If there are states appearing in more than one node then when the sets of states of these nodes are unioned, parts of the corresponding structures are unified.

A Appendix

A.1 The semantics of NWP programs

We start by defining the *Remainder set* of a program, which will be used as the set of locations in the program.

Definition A.1: The remainder set of a program P, denoted $Remain(P)$, is the set including all sub-programs of P that can appear as the remaining program to run:

- $P \in Remain(P)$
- If P ends with the command "fin" then "fin" $\in Remain(P)$, otherwise $E \in Remain(P)$
- If $P_1; P_2 \in Remain(P)$ and $P_1' \in Remain(P_1)$ then $P_1'; P_2 \in Remain(P)$.
- If "if B then P_1 else P_2 fi" $\in Remain(P)$ then $P_1 \in Remain(P)$ and $P_2 \in Remain(P)$.
- If "while B do P_1 od" $\in Remain(P)$ then P_1; while B do P_1 od $\in Remain(P)$.

Note that, if for instance $P = P_1; P_2$ then P_2 will be in the remainder of P, but P_1 will not.

Let $P \in$ NWP be a program such that $x_1, \ldots, x_n$ are the program variables, all of them over some *finite domain* D.

Definition A.2: The *meaning of a program* P is a Kripke structure $struct(P) = \langle S, R, I \rangle$. The set of states is $S = Remain(P) \times D^n$. For a state (l, σ) we refer to l as the *location* of the state. The set of initial states is $I = \{(P, \sigma) \mid \sigma \in D^n\}$. The transition relation R is defined inductively (using the notation $struct(P_i) = \langle S_i, R_i, I_i \rangle$).

For $P = E$: $R = \emptyset$.

For $P =$ "fin": $R = \{(s, s) | s \in S\}$

For $P =$ "skip": $R = \{((P, \sigma), (E, \sigma)) \mid \sigma \in D^n\}$

For $P =$ "x := $\{e_1, \ldots, e_k\}$": $R = \{((P, \sigma), (E, \sigma[x \leftarrow e_i])) \mid 1 \leq i \leq k, \sigma \in D^n\}$.

For $P = P_1; P_2$: $R = \{((Q; P_2, \sigma), (Q'; P_2, \sigma')) \mid ((Q, \sigma), (Q', \sigma')) \in R_1\} \cup R_2$.

For $P =$ "if B do P_1 else P_2 fi": $R = \{((P, \sigma), (P_1, \sigma)) \mid \sigma \models B, \sigma \in D^n\} \cup \{((P, \sigma), (P_2, \sigma)) \mid \sigma \not\models B, \sigma \in D^n\} \cup R_1 \cup R_2$.

For $P =$ "while B do P_1 od": $R = \{((P, \sigma), (P_1; P, \sigma)) \mid \sigma \models B, \sigma \in D^n\} \cup \{((P, \sigma), (E, \sigma)) \mid \sigma \not\models B, \sigma \in D^n\} \cup \{((Q; P, \sigma), (Q'; P, \sigma')) \mid ((Q, \sigma), (Q', \sigma')) \in R_1\}$.

Recall that we define E so that for any program P, $E; P = P$ and therefor the states $(E; P_2, \sigma)$ and (P_2, σ) are the same state. This is significant in the case of sequential composition.

A.2 Partition graphs for NWP programs

The set $pg(P)$ contains all possible partition graphs of P, representing different ways of partitioning P into sub-programs.

The following notation is required for the formal definition of partition graphs. It allows us to take a Kripke structure of a sub-program and put it "in context" of the whole program. This is done by changing the locations of the sub-program to match them to their locations in the whole program. For any set of states S, and program P, the set $S^{;P}$ is obtained from S by replacing each state (l, σ) in S by the state $(l;P, \sigma)$. The transition relation $R^{;P}$ is obtained from R by replacing every pair $((l_1, \sigma_1), (l_2, \sigma_2))$ in R by $((l_1; P, \sigma_1), (l_2; P, \sigma_2))$. For

[6] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, January 1983.

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

[9] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, March 1995.

[10] E. A. Emerson and J. Y. Halpern. "Sometimes" and "Not Never" revisited: On branching time versus linear time. *Journal of the ACM*, 33:151–178, 1986.

[11] P. Godefroid. Model checking for programming languages using VeriSoft. In *Principle of Programming Languages*, January 1997.

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

[13] B. Josko. Verifying the correctness of AADL-modules using model checking. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*. Springer-Verlag, May 1989.

[14] G. Kamhi, O. Weissberg, L. Fix, Z. Binyamini, and Z. Shtadler. Automatic datapath extraction for efficient usage of HDD. In *Proc. of the 9th International Conference on Computer Aided Verification, LNCS vol. 1254*, pages 95–106. Springer, June 1997.

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

[16] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[17] K. L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *Proceedings of the 1991 International Symposium on Shared Memory Multiprocessors*, April 1991.

[18] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[19] A. Pnueli. In transition for global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series F*. Springer-Verlag, 1984.

[20] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.

the inner one. This reflects the tradeoff between space and time complexity. As larger programs can be handled (by applying more refined partitions) the time complexity grows.

6 Conclusions and future development

The algorithm presented in this work can be considered as a framework into which any model checking algorithm for Kripke structures can be integrated. Since our method uses a given model checking algorithm as a procedure, whenever a better algorithm is developed it can immediately be plugged into ours.

Thus, our method can work well with both explicit-state model checking and BDD-based model checking. The former expects the model of the checked system to be given explicitly as a graph (e.g. by an adjacency list). The latter is based on BDD representation [2] of the system model. Each has its advantage for certain areas of applications and each can be made modular using our approach.

An important notion suggested in this work is that of partition graphs. In this work, they were used to partition the model checking task into several sub-tasks. They also maintained the flow of information (by means of assumption functions) between the sub-tasks.

Partition graphs can further be used for top-down design of systems. They may enable to verify a system at an early stage of development, when some of the components have not yet been written. In such cases, the assumption functions will actually play the role of assumptions about components that are yet unknown. The use of partition graphs in that context should be further investigated.

Choosing the right partition graph is crucial to the effectiveness of our method. As presented here, the algorithm is given a specific partition graph, but it may be possible to develop some heuristics that will allow automatic creation of the partition graph.

We will also explore various extensions of our method, to deal with other aspects of software such as procedures, data structures, templates, and parallel composition.

We are currently working on an implementation of our method using BDDs. We intend to use it to verify several example programs and compare performance when different partitions are used.

References

[1] I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver, P. Paanah, Y. Rodeh, and Y. Wolfstahl. Rulebase: Model checking at IBM. In *Proc. of the 9th International Conference on Computer Aided Verification, LNCS vol. 1254*, pages 480–484. Springer, June 1997.

[2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[4] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*. IEEE, IEEE Computer Society Press, June 1990.

[5] Olaf Burkart and Bernhard Steffen. Pushdown processes: Parallel composition and model checking. LNCS 836, pages 98–113. Springer, 1994.

$\underline{\varphi_k \in \{\mathbf{A}(\varphi_l \mathbf{U} \varphi_m), \mathbf{E}(\varphi_l \mathbf{U} \varphi_m)\}}$:

(a) Set $i \leftarrow 1$.
$Init_B \leftarrow As'(\varphi_m) \cap \{s \in init(G) \mid s \models B\}$.
The initial assumption function is $As^0 \leftarrow As'$.
Initialize the value for φ_k: $As^0(\varphi_k) \leftarrow As_{\neg B}(\varphi_k) \cup Init_B$.

(b) do:
- Tmp =CheckGraph(G_1, As^{i-1})
- $As^i_B \leftarrow \mathcal{T}(M_B \stackrel{yes}{\rightarrow} in_1, Tmp)$
- Define As^i so that for all $j < k$, $As^i(\varphi_j) \leftarrow As^0(\varphi_j)$ and $As^i(\varphi_k) \leftarrow As^i_B(\varphi_k) \cup As_{\neg B}(\varphi_k)$
- $i \leftarrow i + 1$

as long as $As^i \neq As^{i-1}$.

(c) $As'(\varphi_k) \leftarrow As^i(\varphi_k)$

4. Return As'.

Theorem 5.2: For any full program P, a CTL formula ψ, a partition graph $G \in pg(P)$ and an empty assumption function $As : cl(\psi) \rightarrow \{\emptyset\}$, if As' =CheckGraph(G, As) then for every $\varphi \in cl(\psi)$ and $s \in init(G)$, $s \in As'(\varphi) \Leftrightarrow s \models \varphi$.

This theorem states that if we run the algorithm on a full program, with an empty assumption function, the resulting function will give us full knowledge about which formulas in $cl(\psi)$ hold in the initial states of the program according to the standard semantics of CTL.

When implementing the above algorithm there are many optimizations that can be done, such as working on several formulas at the same time and keeping information from previous calculations.

The space complexity of our modular algorithm is clearly better than that of algorithms that need to have the full model in the direct memory. Our algorithm holds in the direct memory at any particular moment only the model for the subprogram under consideration at that time. In addition, it keeps an assumption function, which at its largest holds the results of performing model checking on this subprogram. This of course is equivalent to any model checking algorithm that must keep its own results.

The time complexity is harder to analyze. It depends on the model checking algorithm used for a single node and on the partition graph. In most cases, our algorithm is of the same time complexity as algorithms designed to model check unpartitioned models. A case that might require a significant amount of additional computation occurs when an Until formula ψ (either $A(\phi_1 U \phi_2)$ or $E(\phi_1 U \phi_2)$) is checked on a while program in which the body is partitioned.

Let G be the partition graph of the while program. The number of iterations through the body of the while is bounded by the number of initial states of G, $Init(G)$. At each iteration, every node in G is model checked with respect to ψ. Such a case occurs when at each iteration exactly one state from $Init(G)$ is added to the set of states satisfying ψ. This means that there is a a computation that executes the body of the while $|init(G)|$ times without passing through the same state twice. Note that in such a case a regular model checking also has to traverse the same path.

Additional overhead occurs when while loops are nested and each is partitioned. Time complexity then grows since every pass through the outer loop requires several passes through

for a specific state s_0. In order to find out that $s_0 \models \mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$ it is necessary to traverse the path that proves this. This path is revealed only in the second traversal of the loop.

Following is the recursive definition of the algorithm. Given a partition graph $G \in pg(P)$ of a program P, and a consistent assumption $As : cl(\psi) \rightarrow (2^{end(G)} \cup \{\bot\})$, CheckGraph$(G, As)$ returns an assumption $As' : cl(\psi) \rightarrow (2^{init(G)} \cup \{\bot\})$.

CheckGraph(G,As):

The base case is for a single node M, in which case we return As' s.t. $\forall \varphi \in cl(\psi)$, if $As(\varphi) = \bot$ then $As'(\varphi) = \bot$, otherwise $As'(\varphi) = MC[M, As](\varphi) \cap init(M)$.

The three possible recursive cases are the ones depicted in Figure 2. We assume that in_1 (in_2) is the entry node of G_1 (G_2), M_B is the structure in a "B" node, and M_E is the structure in an "E" node.

- For a sequential composition $P_1; P_2$ (Figure 2 A) perform:
  1. $As_1 \leftarrow$CheckGraph(G_2, As).
  2. $As' \leftarrow$CheckGraph(G_1, As_1).
  3. Return(As')
- For a graph of $P =$ "if B then P_1 else P_2 fi" (Figure 2 B) perform:
  1. $As_1 \leftarrow$CheckGraph(G_1, As).
  2. $As_2 \leftarrow$CheckGraph(G_2, As).
  3. $As_B \leftarrow \mathcal{T}(M_B \stackrel{yes}{\rightarrow} in_1, As_1)$
  4. $As_{\neg B} \leftarrow \mathcal{T}(M_B \stackrel{no}{\rightarrow} in_2, As_2)$
  5. For every formula $\varphi \in cl(\psi)$, if $As_B(\varphi) = \bot$ then define $As'(\varphi) = \bot$ [4]. Otherwise, $As'(\varphi) = As_B(\varphi) \cup As_{\neg B}(\varphi)$ (Notice that the images of As_B and $As_{\neg B}$ are disjoint).
  6. Return As'
- For a graph of $P =$ "while B do P_1 od" (Figure 2 C) perform:
  1. $As_{\neg B} \leftarrow \mathcal{T}(M_B \stackrel{no}{\rightarrow} M_E, As)$
  2. Find an ordering $\varphi_1, \varphi_2, \ldots, \varphi_n$ of the formulas in $cl(\psi)$ such that each formula appears after all of its sub-formulas. Set $As'(\varphi_i) = \bot$ for all i. For $k = 1, \ldots, n$ perform step 3 to define $As'(\varphi_k)$ [5].
  3. Perform one of the following, according to the form of φ_k:
     $\underline{\varphi_k \in AP}$: $As'(\varphi_k) \leftarrow \{s \in init(G) \mid \varphi_k \in \mathcal{L}(s)\}$
     $\underline{\varphi_k = \neg \varphi_l}$: $As'(\varphi_k) \leftarrow \{s \in init(G) \mid s \notin As'(\varphi_l)\}$.
     $\underline{\varphi_k = \varphi_l \vee \varphi_m}$: $As'(\varphi_k) \leftarrow As'(\varphi_l) \cup As'(\varphi_m)$.
     $\underline{\varphi_k \in \{\mathbf{AX}\,\varphi_l, \mathbf{EX}\,\varphi_l\}}$:
     (a) $Tmp \leftarrow$CheckGraph(G_1, As')
     (b) Let $R_{M_B \stackrel{yes}{\rightarrow} in_1}$ be the set of transitions induced by the edge $M_B \stackrel{yes}{\rightarrow} in_1$.
     $As'(\varphi_k) \leftarrow As_{\neg B}(\varphi_k) \cup \{s \in init(G) \mid \forall s' \in init(G_1), (s, s') \in R_{M_B \stackrel{yes}{\rightarrow} in_1} \Rightarrow (s \models B \wedge s' \in Tmp(\varphi_l))\}$ [6].

[4] Since As_B and $As_{\neg B}$ both originate from the same assumption function As, it holds that $As_B(\varphi) = \bot iff As_{\neg B}(\varphi) = \bot$.

[5] Notice that when working on φ_k we have already calculated As' for all of its sub-formulas.

[6] The definition is the same for $\mathbf{AX}\,\varphi_l$ and $\mathbf{EX}\,\varphi_l$ because for each state $s \in init(G)$ there is exactly one state $s' \in init(G_1)$ s.t. $(s, s') \in R_{M_B \stackrel{yes}{\rightarrow} in_1}$.

G we first check G_1 and G_2, and then compute 'backwards' over the step-edges (using the function $\mathcal{T}$) to get the result for the initial states of G.

The most complicated part of the algorithm is for the partition graph G of a program $P =$ "while B do P_1 od", as in figure 2 C. We start from the node M_E, for which we have the assumption As. Walking backwards on the no-edge we use the function $\mathcal{T}$ to get an assumption $As_{\neg B}$ over the initial states of G that satisfy $\neg B$. We now demonstrate the computation of $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$. We assume that $As'(\varphi_1)$ and $As'(\varphi_2)$ were already calculated. The goal is to mark all states that satisfy $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$ (to create $As'(\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2))$. Standard model checking algorithms would start by marking all states that satisfy φ_2, and then repeatedly move backwards on transitions and mark every state that has a transition into a marked state, and satisfies φ_1 itself. We reconstruct this computation over the partition graph of P. For initial states of G that satisfy B we have no assumption regarding $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$, so we mark all those that satisfy B and φ_2 and keep them in $Init_B$. Together with $As_{\neg B}(\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2))$ we have an initial estimate for $As'(\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2))$ (kept in $As^0(\varphi_k)$). We now want to mark all the fathers in G of these states. Notice that $init(G) = end(G_1)$ so these fathers are inside G_1. Hence we continue from $end(G_1)$ backwards inside G_1 until we arrive at $init(G_1)$. At this point, only the marks on states of $init(G_1)$ are kept (in Tmp). The marks on all other states of G_1 are not preserved. Notice that G_1 itself may consist of more than one node, and the creation of Tmp is done by a recursive call to CheckGraph. From Tmp we can calculate a new estimate for $As'(\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2))$.

The whole process repeats itself since the body of a "while" loop can be executed more than once. Obviously, it is essential that the states satisfying φ_1 and φ_2 be known before this process can be performed. Therefore, we use the $\perp$ value for $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$ when working on the assumptions for φ_1 and φ_2. Only when calculations for all sub-formulas are completed, we may begin calculating the proper result for $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$.

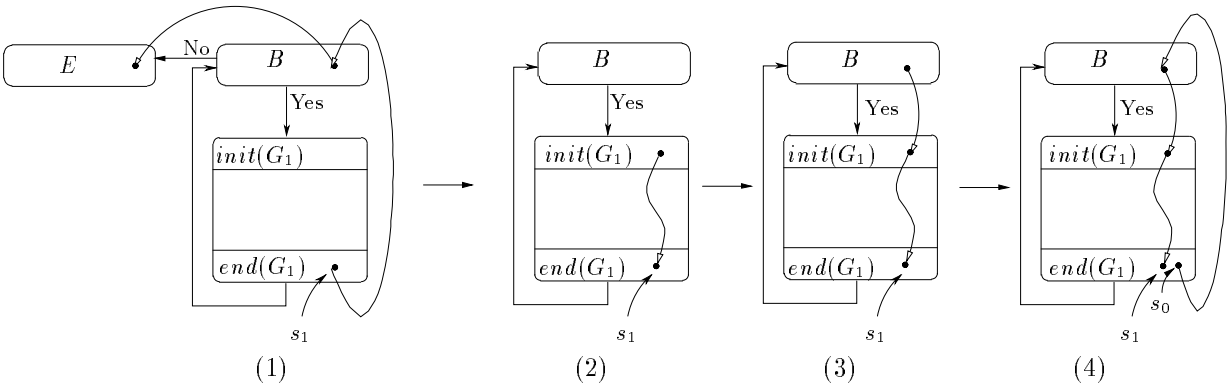


Figure 5: The steps taken by the algorithm in order to reveal that the state s_0 satisfies $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$

This process stops when the assumption calculated reaches a fix-point ($As^i = As^{i-1}$). Obviously, no new information will be revealed by performing another cycle. The set of states in $init(G)$ that are marked increases with each cycle, until all states that satisfy the formula are marked, at which point the algorithm stops. Figure 5 illustrates the algorithm

truth of formulas in the structure of G then the derived assumption also coincides with the truth of formulas in the structure. The proof of this is omitted due to space restrictions.

5.2 The Compositional Algorithm

Following, we give an algorithm to check a formula ψ on a partition graph G of a full program P. The result is an assumption function over the set of initial states of P that gives, for every sub-formula φ of ψ, the set of all initial states of P satisfying φ. We start with an intuitive description of the algorithm.

The algorithm works on G from the exit node upwards to the entry node. First the structure contained in a leaf node V of G is model checked under an "empty" assumption for $cl(\psi)$, an assumption in which all values are $\emptyset$. Since V is a leaf it must represent a full program and therefore all paths in it are infinite, and the assumption function has no influence on the result. The result of the model checking algorithm is an assumption function As' that associates with every sub-formula of ψ the set of all initial states of V that satisfy that sub-formula. Once we have As' on V we can derive a similar function As on the ending states of any node U, preceding V in G (that is, any node U from which there is an edge into V). Next, we model check U under the assumption As. Proceeding this way, each node in G can be checked in isolation, based on assumptions derived from its successor nodes.

Given a procedure that properly computes $MC[M, As]$, we define the algorithm CheckGraph that takes an assumption function As and a partition graph G and performs model checking under assumption resulting in an assumption As'. The answer to the model checking problem is $As'(\psi)$. CheckGraph is able to handle partially defined assumption functions, in which there are some $\perp$ values. For any sub-formula φ s.t. $As(\varphi) = \perp$ we get $As'(\varphi) = \perp$. CheckGraph is defined by induction on the structure of G. The base case handles a single node, that may contain the Kripke structure of any program, by using the given model checking procedure. To model check a partition graph G of $P = P_1; P_2$, as in figure 2 A, CheckGraph first checks G_2 under As (see Figure 4). As_1 is the result of this check (As_1 is over the set $init(G_2)$). It then uses As_1 as an assumption on the ending states of G_1 and checks P_1 w.r.t As_1 . The second check returns for all $\varphi \in cl(\psi)$ the set of all initial states of P_1 (also initial states of P) that satisfy φ, which is the desired result.

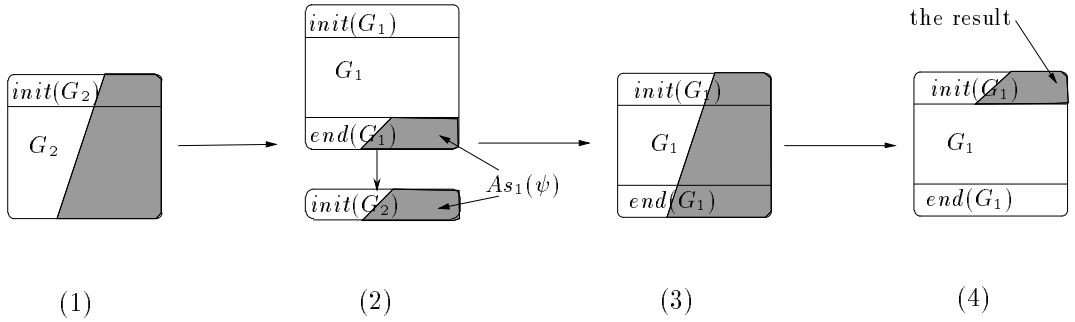


Figure 4: The operation of the algorithm on sequential composition. The gray area is the set of states that satisfy ψ.

Let G be a partition graph of $P =$ "if B then P_1 else P_2", as in figure 2 B. To check

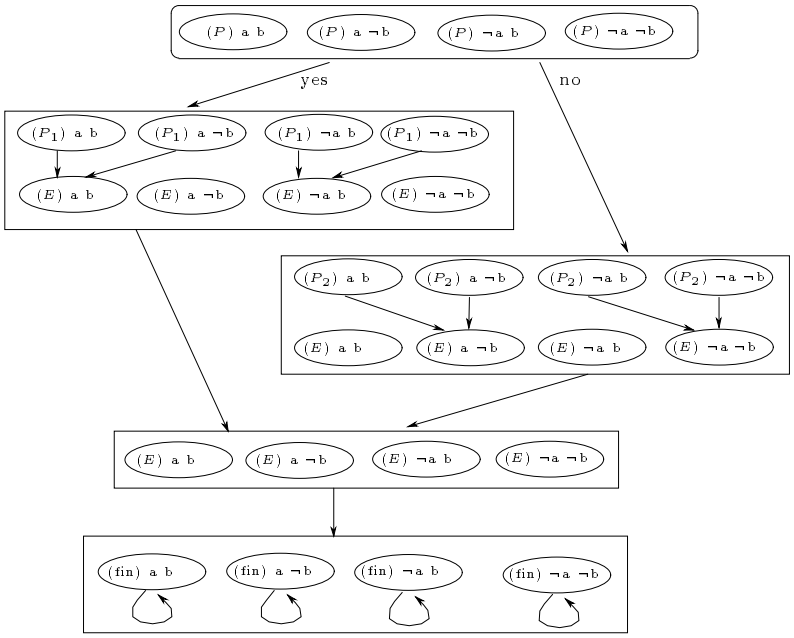


Figure 3: An example partition graph for the program P;fin, where P is the program from figure 1.

and $M_B = \langle S_B, \emptyset, S_B \rangle$, and let $As : cl(\psi) \rightarrow (2^{S_1} \cup \{\bot\})$ be an assumption function over M_1. $\mathcal{T}(e, As) = As'$ s.t. $As' : cl(\psi) \rightarrow (2^{S_T} \cup \{\bot\})$. The set $S_T \subseteq S_B$ is the set of states in S_B that satisfy the condition B. This is exactly the set of states from which there will be an edge into a state of M_1 in $struct(G)$. Moreover, assume that l is the location of all the states in S_B and l' is the location of the states in I_1. Then the definition of $struct(G)$ is such that from each state $s = (l, \sigma) \in S_B$ s.t. $\sigma \models B$ there is exactly one transition, into a state $s' = (l', \sigma)$. As a result, there is no difference between "for all paths" and "there exists a path" and therefore the operators $\mathbf{AX}\,\varphi$ and $\mathbf{EX}\,\varphi$ are handled in exactly the same way, and so are the operators $\mathbf{A}(\varphi_1 \mathbf{U} \varphi_2)$ and $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$.

If $As(\varphi) = \bot$ then $As'(\varphi) = \bot$. Otherwise, $As'(\varphi)$ is defined recursively as follows [3]

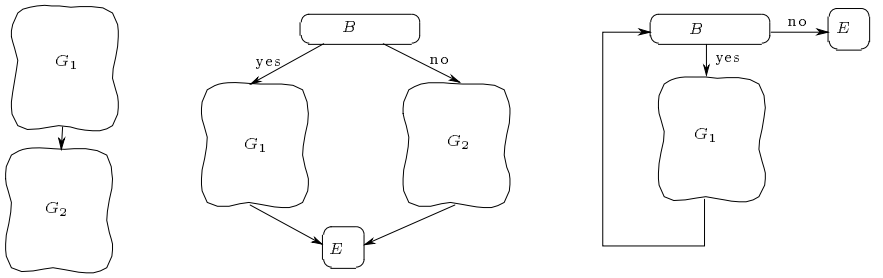
- For any $p \in AP$, $As'(p) = \{s \in S_T \mid p \in \mathcal{L}(s)\}$.
- $As'(\neg\varphi) = S_T \setminus As'(\varphi)$
- $As'(\varphi_1 \vee \varphi_2) = As'(\varphi_1) \cup As'(\varphi_2)$
- $As'(\mathbf{AX}\,\varphi) = As'(\mathbf{EX}\,\varphi) = \{(l, \sigma) \in S_T \mid (l', \sigma) \in As(\varphi)\}$
- $As'(\mathbf{A}(\varphi_1 \mathbf{U} \varphi_2)) = As'(\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)) = As'(\varphi_2) \cup (As'(\varphi_1) \cap \{(l, \sigma) \in S_T \mid (l', \sigma) \in As(\mathbf{A}(\varphi_1 \mathbf{U} \varphi_2))\})$

For a no-edge $M_B \stackrel{no}{\rightarrow} M_1$ the definition is the same, replacing every use of S_T by S_F which is the set of states that do not satisfy B.

An important feature of this operation is that if the original assumption coincides with the

[3] If $As(\varphi) \neq \bot$ then for all sub-formulas φ' of φ it holds that $As(\varphi') \neq \bot$.

for B, which is the entry node, and an E node as the exit node. The edges represent the semantics of the "while" loop. (Figure 2 C).



(A) $P = P_1; P_2$ (B) P = if B then P_1 else P_2 fi (C) P = while B do P_1 od

Figure 2: Creation of partition graphs

The formal definition of partition graphs and their semantics (given as Kripke structures) will be given in the full version. It is defined so that given any partition graph $G \in pg(P)$, the structure that defines its semantics, denoted $struct(G)$, is identical to $struct(P)$. Informally, $struct(G)$ is created out of the union of all Kripke structures in its nodes (with some adjustment of the program locations). Each step-edge induces a set of transitions from the states in the node representing the boolean expression, to initial states in the node that is pointed at by the edge. A yes-edge (no-edge) creates one transition from each state that satisfies (does not satisfy) the condition into the corresponding state (different location, same assignment to variables). A null-edge $M_1 \rightarrow M_2$ does not create transitions.

Given a partition graph G we define $init(G)$ to be the set of initial states in $struct(G)$ and $end(G)$ to be the set of ending states in $struct(G)$. Figure 3 gives an example of an actual partition graph.

5 Performing Modular Model Checking

Our algorithm for modular model checking is based on the notion of satisfaction under assumptions. Furthermore, the basic building block in the recursive definition of the algorithm is "model checking under assumptions". We do not give here an explicit algorithm to compute it, we just note that every standard model checking algorithm for CTL can easily be adapted to handle assumptions.

Before we present our modular algorithm we define a few operations on assumption functions that we use in the algorithm.

5.1 Operations on Assumption Functions

We first present an operation $\mathcal{T}$ that, given a step-edge $e = M_B \stackrel{yes}{\rightarrow} M_1$ or $e = M_B \stackrel{no}{\rightarrow} M_1$ (M_B is a structure representing a condition B), and an assumption function As over the initial states of M_1, results in an assumption function As' over M_B. As' is defined so that it represents all the knowledge that As gives, translated over the edge.

Definition 5.1: Let $e = M_B \stackrel{yes}{\rightarrow} M_1$ be an edge in a partition graph G s.t. $M_1 = \langle S_1, R_1, I_1 \rangle$

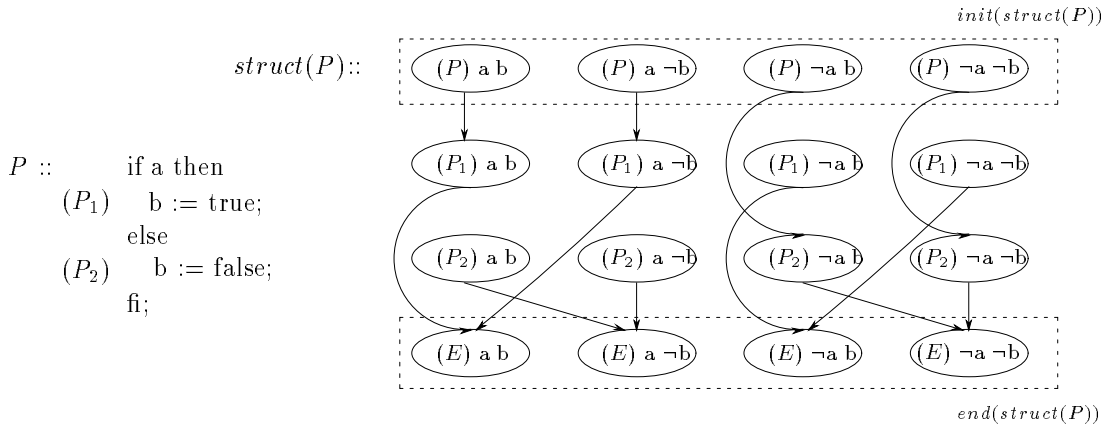


Figure 1: An example of an NWP program.

There are three types of edges: null-edges, yes-edges, and no-edges, denoted $M_1 \rightarrow M_2$, $M_1 \stackrel{yes}{\rightarrow} M_2$ and $M_1 \stackrel{no}{\rightarrow} M_2$ respectively. A null edge $M_1 \rightarrow M_2$, where $M_1 = struct(P_1)$ and $M_2 = struct(P_2)$, means that $init(P_2) = end(P_1)$. This happens when there is no step in the execution between the corresponding sub-programs, for example, when the program to be executed is $P_1; P_2$. Yes-edges and no-edges, called *step-edges*, are edges outgoing from a node representing a boolean expression. Execution from a state in this node continues through the yes-edge or the no-edge, depending on whether the expression evaluates to *true* or *false* in that state. These edges also represent a step in the execution. A partition graph also has two designated nodes: the *entry node*, from which execution starts, and the *exit node*, at which it stops.

The set $pg(P)$ contains all possible partition graphs of P, representing different ways of partitioning P into sub-programs. It is defined recursively, where at each step one may decide to break a given program according to its primary structure, or to create a single node out of it. Figure 2 shows the three different ways in which a program may be decomposed, according to the three structures by which programs are created. We use in_1 (in_2) for the entry node of G_1 (G_2) and out_1 (out_2) for the exit node.

1. If $P = P_1; P_2$ we may decompose it into two parts, by creating (recursively) partition graphs $G_1 \in pg(P_1)$ and $G_2 \in pg(P_2)$, and connecting them with a null edge from out_1 to in_2. The entry node of the resulting graph would be in_1, and the exit node would be out_2 (Figure 2 A).
2. If $P =$ "if B then P_1 else P_2 fi", we again create the two graphs $G_1 \in pg(P_1)$ and $G_2 \in pg(P_2)$ but also create two new nodes, one representing the boolean expression B and the other representing the empty program E. The Kripke structure representing E has no edges (an empty transition relation) and its set of states is the product of D^n and the location E. This node is used as the exit node, and the entry node is the B node. The edges connecting the different components are according to the semantics of the "if" command. Again, the edges entering G_1 and G_2 are to in_1 and in_2 and the edges exiting G_1 and G_2 are from out_1 and out_2. (Figure 2 B).
3. If $P =$ "while B do P_1 od", we create a partition graph $G_1 \in pg(P_1)$ and again a node

3 The Programming Language

Following we define the syntax and semantics of our programming language NWP (Nondeterministic While-Programs).

Definition 3.1: We assume a fixed set of program variables over some finite domain D. A *program fragment* is one of $x := \{e_1, \ldots, e_k\}$, skip, $Prog_1; Prog_2$, "if B then $Prog_1$ else $Prog_2$ fi" or "while B do $Prog_1$ od" s.t. $Prog_1$,$Prog_2$ are program fragments, B is a boolean expression over program variables and constants, x stands for any program variable, and e is an expression over program variables and constants. The meaning of x:=$\{e_1, \ldots, e_k\}$ is a nondeterministic assignment, $Prog_1; Prog_2$ is the sequential composition of $Prog_1$ and $Prog_2$, and the "if" and "while" structures have the same meaning as in all sequential programming languages.

A *full program* is of the form $Prog$;fin where $Prog$ is a program fragment. The meaning of "fin" is an infinite loop that does not change the values of program variables. We define E to be the empty program, such that for every $P \in$ NWP it holds that $P; E = E; P = P$. The set NWP is the set of all full programs.

From here on we use the word "program" to refer to either a full program, or a program fragment, unless stated otherwise.

The semantics of NWP programs is given by means of Kripke structures. We give here only an informal description, the formal definition will appear in the full version.

Let $P \in$ NWP be a program such that $x_1, \ldots, x_n$ are the program variables. An assignment to the program is be some $\sigma \in D^n$. We create a Kripke structure $struct(P)$ so that each state is a pair (l, σ) where l is a program location and $\sigma \in D^n$ is an assignment to the program variables. Each location is associated with the remaining program to be run from that point on. The transition relation is created in the intuitive way, following the usual semantics of the commands. Evaluating a boolean expression (in an "if" or "while" command) is considered a step in execution. We define the set of initial states $init(P)$ as the set of states with location P. The set of ending states, $end(P)$, is the set of states in $struct(P)$ that have no outgoing transition. If P is a full program then $end(P) = \emptyset$. If P is a program fragment then this is the set of states with the location E (which means that there is nothing more to run).

We add to AP the set $\{at\_l \mid l \text{ is a location in } P\}$. The new propositions are used to refer to a location in the program within the specification. The labeling function $\mathcal{L} : S \to 2^{AP}$ is extended accordingly so that $\mathcal{L}((l, \sigma)) = \{at\_l\} \cup \{p \in AP \mid \sigma \models p\}$ for every state (l, σ) in $struct(P)$.

Figure 1 includes an example of a NWP program, and its structure.

4 Partition Graphs

A *Partition Graph* of a program P is a finite graph representing a decomposition of P into several sub-programs while maintaining the original flow of control. The nodes are Kripke structures, each representing a sub-program of P or a boolean expression. A node representing a sub-program P' is of the form $struct(P')$. A node representing a boolean expression B, has the form $\langle S, R, I \rangle$ s.t. $R = \emptyset$ and $S = I = \{(l, \sigma) \mid \sigma \in D^n\}$ where l is the program location of the "if" or "while" command that evaluates B.

assume that $\neg\varphi$ holds. When $As(\varphi) = \bot$ it means that we have no knowledge regarding the satisfaction of φ in S'.

Satisfaction of a CTL formula φ in a state $s \in S$ under an assumption function As is denoted $(M, \mathcal{L}), s \models_{As} \varphi$ [2]. We define it so that it holds if either $M, s \models \varphi$ directly (by infinite paths only), or through the assumption. For example, $M, s \models_{As} \mathbf{E}(f\mathbf{U}g)$ if there exists an infinite path from s satisfying f in all states until a state satisfying g is reached, but it is also true if there is a finite path from s in which the last state, say s', satisfies $s' \in As(\mathbf{E}(f\mathbf{U}g))$, and all states until s' satisfy f. Formally:

Definition 2.5: Let $M = \langle S, R, I\rangle$ be a Kripke structure and As a consistent assumption function over M. For every $\varphi \in cl(\psi)$:
If $As(\varphi) = \bot$ then $s \models_{As} \varphi$ is not defined.
Otherwise, we differentiate between ending states and other states. If $s \in end(M)$ then $s \models_{As} \varphi$ iff $s \in As(\varphi)$. If $s \in S \setminus end(M)$ then $s \models_{As} \varphi$ is defined as follows:

- $s \models_{As} p$ iff $p \in \mathcal{L}(s)$ for every $p \in AP$.
- $s \models_{As} \varphi_1 \vee \varphi_2$ iff $(s \models_{As} \varphi_1$ or $s \models_{As} \varphi_2)$.
- $s \models_{As} \neg\varphi_1$ iff $s \not\models_{As} \varphi_1$.
- $s \models_{As} \mathbf{AX}\,\varphi_1$ iff $\forall s'.(s, s') \in R \Rightarrow s' \models_{As} \varphi_1$.
- $s \models_{As} \mathbf{EX}\,\varphi_1$ iff $\exists s'.(s, s') \in R \wedge s' \models_{As} \varphi_1$.
- $s \models_{As} \mathbf{A}(\varphi_1\mathbf{U}\varphi_2)$ iff for all (maximal) paths $\pi = s_0, s_1, \ldots$ from s there is a number $i < |\pi|$ s.t. (either $s_i \models_{As} \varphi_2$ or $s_i \in end(M) \wedge s_i \models_{As} \mathbf{A}(\varphi_1\mathbf{U}\varphi_2)$), and $\forall 0 \leq j < i[s_j \models_{As} \varphi_1]$.
- $s \models_{As} \mathbf{E}(\varphi_1\mathbf{U}\varphi_2)$ iff there exist a (maximal) path $\pi = s_0, s_1, \ldots$ from s and a number $i < |\pi|$ s.t. (either $s_i \models_{As} \varphi_2$ or $s_i \in end(M) \wedge s_i \models_{As} \mathbf{E}(\varphi_1\mathbf{U}\varphi_2)$) and $\forall 0 \leq j < i[s_j \models_{As} \varphi_1]$.

Note that, if the transition relation of M is total then the above definition is equivalent to the traditional definition of CTL semantics, because the assumption function is consulted only on states from which there are no outgoing transitions.

2.3 Model Checking Under Assumptions

The task of model checking is to find all initial states of a given structure that satisfy a formula ψ. We write $M \models_{As} \psi$ iff $\forall s \in I, [M, s \models_{As} \psi]$. From here on we assume that ψ is the formula to be checked on a structure $M = \langle S, R, I\rangle$ (or later, a program).

Definition 2.6: Given an assumption function As over a structure M we define a function $MC[M, As] : cl(\psi) \rightarrow (2^S \cup \{\bot\})$ so that for any $\varphi \in cl(\psi)$, if $As(\varphi) = \bot$ then $MC[M, As](\varphi) = \bot$. Otherwise, $MC[M, As](\varphi) = \{s \in S \mid M, s \models_{As} \varphi\}$.

Notice that $MC[M, As]$ results in an assumption function. Given M and As, this function can be calculated using any known model checking algorithm for CTL [7, 20, 4], after adapting it to the semantics under assumptions.

[2] Since we assume a fixed $\mathcal{L}$, we always omit $\mathcal{L}$. When no confusion may occur we also omit M.

2 Basic Definitions

2.1 Kripke Structures

Kripke structures are widely used for modeling finite-state systems. In this paper we use Kripke structures to model the behavior of a finite program.

Definition 2.1: A *Kripke Structure* is a tuple $M = \langle S, R, I \rangle$ s.t. S is a set of states, $R \subseteq S \times S$ is a *transition relation* and $I \subseteq S$ is a set of *initial states*. A *path* in M from a state s_0 is a sequence $\pi = s_0, s_1, \ldots$ s.t. $\forall i[s_i \in S$ and $(s_i, s_{i+1}) \in R]$. A *maximal* path in M is a path which is either infinite, or ends in a state with no outgoing transitions. Let π be a maximal path in M. We write $|\pi| = n$ if $\pi = s_0, s_1, \ldots, s_{n-1}$ and $|\pi| = \infty$ if π is infinite.

Definition 2.2: For a Kripke structure $M = \langle S, R, I \rangle$ we define the set of *ending states* to be $end(M) = \{s \in S \mid \neg \exists s'.(s, s') \in R\}$. We also use $init(M)$ to refer to the set I of initial states.

2.2 CTL

For our specification language we use the propositional branching-time temporal logic CTL. It allows us to specify a behavior of a program in terms of its computation tree [6].

We assume a set of *atomic propositions* AP and a labeling function that associates with each state in a structure the set of atomic propositions true at that state. Throughout the paper we assume a fixed labeling function $\mathcal{L} : S \rightarrow 2^{AP}$.

We define a CTL formula to be either q for each $q \in AP$, or $\neg f_1$, $f_1 \vee f_2$, $\mathbf{AX}\, f_1$, $\mathbf{EX}\, f_1$, $\mathbf{A}(f_1 \mathbf{U} f_2)$, and $\mathbf{E}(f_1 \mathbf{U} f_2)$ where f_1 and f_2 are CTL formulas. Each temporal operator in CTL is constructed of a path quantifier, either A ("for all paths") or E ("for some path"), and a temporal operator X or $\mathbf{U}$. Intuitively, the operator X means "at the next step", so the formula $\mathbf{AX}\, q$ states that in all the paths outgoing from a given state, the second state satisfies q. A path satisfies $p\, \mathbf{U}\, q$ (p "Until" q) if there exists a state along it that satisfies q and all the states preceding it satisfy p.

CTL formulas are usually interpreted over Kripke structures that have a total transition relation, so that all paths are infinite. We denote the standard semantics for CTL [10, 6] as $M, s \models \psi$ (meaning that the state s in the structure M satisfies ψ). In this paper we introduce an interpretation for CTL over a Kripke structure and an *assumption function* (defined below). The use of assumption functions enables us to give semantics (over infinite paths) in case of incomplete information. When a finite path occurs in a structure, we view it as a prefix of a set of infinite paths with unspecified continuations. The assumption function states which formulas are true over this absent continuation. We use this information only for states in $end(M)$, so the function may be defined only over some subset of S that includes $end(M)$.

Definition 2.3: The *closure* of a formula ψ, $cl(\psi)$, is the set of all the sub-formulas of ψ (including itself).

Definition 2.4: An *assumption function* for a Kripke structure $M = \langle S, R, I \rangle$ is a function $As : cl(\psi) \rightarrow (2^{S'} \cup \{\bot\})$ where $S' \subseteq S$. We require that $end(M) \subseteq S'$ and that $\forall \varphi \in cl(\psi)$, if $As(\varphi) \neq \bot$ then $\forall \varphi' \in cl(\varphi)$, $As(\varphi') \neq \bot$.

For every $\varphi \in cl(\psi)$, if $As(\varphi) \neq \bot$ then $As(\varphi)$ represents the set of all states in S' for which we assume (or know) that φ holds. For every state $s \in S'$ s.t. $s \notin As(\varphi)$ we

that determines the truth of temporal formulas based on the given assumption function. Only minor changes are needed in order to adapt a standard model checking algorithm so that it performs model checking under assumptions.

Given a procedure that performs model checking under assumptions, we develop a *modular model checking algorithm* that checks the program in parts. To illustrate how the algorithm works consider the program $P = P_1; P_2$. We notice that every path of P lies either entirely within P_1 or has a prefix in P_1 followed by a suffix in P_2. In order to check a formula ψ on P, we first model check ψ on P_2. The result does not depend on P_1 and therefore the algorithm can be applied to P_2 in isolation. We next want to model check P_1, but now the result does depend on P_2. In particular, ending states of P_1 have their continuations in P_2. However, each ending state of P_1 is an initial state of P_2 for which we have already the model checking result [1]. Using this result as an *assumption* for P_1, we can now model check P_1 in isolation. This scheme saves significant amounts of space since at any given time the memory contains only the model of the component under consideration, together with the assumption function that maps formulas to the ending states of that component.

Our modular algorithm can handle any finite-state while program with non-deterministic assignments. In addition to sequential composition, programs may include choices ("if-then-else") and while loops, nested in any way.

Works discussing model checking of programs written in a high level language are rare. The closest to our work is [11] that verifies concurrent systems written in C. However, their approach is not modular. Moreover, they do not handle a full temporal logic. Another related work is [5], in which they perform model checking on Pushdown Process Systems by considering the semantics of 'fragments', which are interpreted as 'incomplete portions' of the process. The model checking algorithm they propose calculates the *property transformer* of each fragment, which is a function that represents the semantics of a fragment with respect to alternation-free mu-calculus formulas. This algorithm, however, works on all fragments together. It should also be noted that Pushdown Process Systems are suitable for modeling (parallel) processes but they can hardly be considered as a high level programming language.

In contrast, our work applies model checking to programs written in a high level programming language, while exploiting their textual structure in order to reduce space requirements. We consider our work as a first step in making model checking applicable to realistic software systems.

The paper is organized as follows. In section 2 we introduce the temporal logic CTL and define its semantics under assumptions. Section 3 describes the syntax and semantics of our programming language and Section 4 defines partition graphs. Section 5 gives the modular model checking algorithm. Section 6 concludes with directions for future research.

[1] The result includes for each sub formula φ of ψ the set of states satisfying φ.

1 Introduction

This work presents a new modular approach that makes temporal logic model checking applicable to large sequential finite-state programs, written in some high level programming language.

Finite-state programs can be useful for describing, in some level of abstraction, many interesting systems. They can describe the behavior of communication protocols. They can be used to describe expert systems, provided that some of the inputs are mapped into a finite domain. Such programs (written in behavioral Hardware Description Languages) are being used to describe the high level behavior of hardware designs. All these examples are *reactive*, i.e., they continuously interact with their environment. They are also quite complex which makes their verification an important and non-trivial task. Furthermore, even though they are sequential they might be significantly large.

A first step in verification is choosing a specification language. Temporal logics [18], capable of describing behaviors over time, have proven to be most suitable for the specification of reactive systems. When restricted to finite-state systems, propositional temporal logic specifications [10] can be checked by efficient algorithms, called model checking [8, 20, 15, 3]. *Temporal logic model checking* procedures typically receive a system model by means of a state transition graph and a formula in the logic, and determine the set of states in the model that satisfy the formula. Tools based on model checking [16] were successful in finding subtle bugs in real-life designs [17, 9] and are currently in use by the hardware industry as part of the verification efforts of newly developed hardware designs [1, 14].

Unfortunately, similar applications of model checking to programs are very limited. One reason for this deficiency arises from the fact that large hardware systems are usually composed of many components working in parallel. Software systems, on the other hand, can be extremely large even when they consist of one sequential component. A useful approach to reducing space requirement is modularity. *Modular model checking* techniques treat each component in separation, based on an assumption about the behavior of its environment [19, 13, 12]. Existing techniques, however, are based on partitioning the system into processes that run in parallel.

Our work applies a modular approach to sequential programs. To do so, we suggest a way of partitioning the program into components that follows the program *text*. A given program may have several different partitions. A partition of the program is represented by a *partition graph*, whose nodes are models of the subprograms and whose edges represent the flow of control between subprograms.

Once the program is partitioned, we wish to check each part separately. However, verifying one component in isolation amounts to checking the specification formula on a model in which some of the paths are truncated, i.e. for certain states in the component we do not know how the computation proceeds (since the continuation is in another component). Such states are called *ending states*. We notice, however, that the truth of a formula at a state inside a component can be determined solely by considering the state transition graph of this component, and the set of formulas which are true at the ending state. Moreover, the truth of the formula at such states depends only on the paths leaving the state and not on the paths leading to it. This observation is the basis for our algorithm.

We define a notion of *assumption function* that represents partial knowledge about the truth of formulas at ending states. Based on that, we define a *semantics under assumption*

Modular Model Checking of Software

Karen Laster Orna Grumberg
Computer Science Department
The Technion
Haifa 32000, Israel
email: {laster,orna}@cs.technion.ac.il

July 17, 1997

Abstract

This work presents a modular approach to temporal logic model checking of software.

Model checking is a method that automatically determines whether a finite state system satisfies a temporal logic specification. Model checking algorithms have been successfully used to verify complex systems. However, their use is limited by the high space requirements needed to represent the verified system.

When hardware designs are considered, a typical solution is to partition the design into units running in parallel, and handle each unit separately. For software systems such a solution is not always feasible. This is because a software system might be too large to fit into memory even when it consists of a single sequential unit.

To avoid the high space requirements for software we suggest to partition the program text into *sequentially composed subprograms*. Based on this partition, we present a model checking algorithm for software that arrives at its conclusion by examining each subprogram in separation. The novelty of our approach is that it uses a decomposition of the program in which the interconnection between parts is sequential and not parallel. We handle each part separately, while keeping all other parts in an external memory. Consequently, our approach reduces space requirements and enables verification of larger systems.

Our method is applicable to finite state programs. Further, it is applicable to infinite state programs provided that a suitable abstraction can be constructed.

We consider this work as a first step towards making temporal logic model checking useful for software verification.