Modular Model Checking of Software

Karen Laster and Orna Grumberg

Computer Science Department

The Technion

Haifa 32000, Israel

email: {laster,orna}@cs.technion.ac.il

phone: 972-4-8294327 fax: 972-4-8221128

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 on an external memory (files). 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 implemented the ideas described in this paper in a prototype tool called *SoftVer* and applied it to a few small examples. We have achieved reduction in both space and time requirements.

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

1 Introduction

This work presents a new modular approach that makes temporal logic model checking applicable to large non-deterministic 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 [17], 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 [7] can be checked by efficient algorithms, called model checking [5, 19, 14, 2]. *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 [15] were successful in finding subtle bugs in real-life designs [16, 6] and are currently in use by the hardware industry in the verification process of newly developed hardware designs [1, 12].

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 [18, 11, 9]. 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, following 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 states. Moreover, the truth of a formula at an ending state depends only on the paths leaving it, 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 that determines the truth of temporal formulas based on a 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. Handling loops in the program is more complicated but follows a similar intuition.

The suggested 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. Often it also saves time, since the model checking task is performed on substantially smaller models.

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 are [8] that verifies concurrent systems written in C, and the SPIN system [10] that validates protocols. However, their approaches are not modular. Another related work is [3], 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.

We implemented the ideas described in this paper in a prototype tool called *SoftVer*. We applied the tool to a few small examples, each with different partitions and compared the space and time requirements needed for model checking with the space and time used when the program is unpartitioned. In all cases,

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

a substantial space reduction has been achieved. Furthermore, in five out of six cases a significant reduction in time has been obtained as well. In the concluding section we summarize these results and explain them.

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. Finally, Section 6 describes the implementation and suggests directions for future research.

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 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 $\pi = s_0, s_1, \dots$ s.t. $\forall i[s_i \in S \text{ and } (s_i, s_{i+1}) \in R]$. 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 $\pi = s_0, s_1, \dots, s_{n-1}$ and $|\pi| = \infty$ if π is infinite.

Definition 2 For a Kripke structure $M = \langle S, R, I \rangle$ we define the set of ending states to be: $\text{end}(M) = \{s \in S \mid \neg \exists s'. (s, s') \in R\}$. We also use $\text{init}(M)$ 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 [4].

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 $\mathbf{A}$ ("for all paths") or $\mathbf{E}$ ("for some path"), and a temporal operator $\mathbf{X}$ or $\mathbf{U}$. Intuitively, the operator $\mathbf{X}$ 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* [7, 4] 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 $\text{end}(M)$, so the function may be defined only over some subset of S that includes $\text{end}(M)$.

Definition 3 *The closure of a formula ψ , $\text{cl}(\psi)$, is the set of all the sub-formulas of ψ (including itself).*

Definition 4 *An assumption function for a Kripke structure $M = \langle S, R, I \rangle$ is a function $As : \text{cl}(\psi) \rightarrow (2^{S'} \cup \{\perp\})$ where $S' \subseteq S$. We require that $\text{end}(M) \subseteq S'$ and that $\forall \varphi \in \text{cl}(\psi)$, if $As(\varphi) \neq \perp$ then $\forall \varphi' \in \text{cl}(\varphi)$, $As(\varphi') \neq \perp$.*

For every $\varphi \in \text{cl}(\psi)$, if $As(\varphi) \neq \perp$ 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 assume that $\neg\varphi$ holds. When $As(\varphi) = \perp$ 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 5 *Let $M = \langle S, R, I \rangle$ be a Kripke structure and As an assumption function over M . For every $\varphi \in \text{cl}(\psi)$:*

If $As(\varphi) = \perp$ then $s \models_{As} \varphi$ is not defined.

Otherwise, we differentiate between ending states and other states. If $s \in \text{end}(M)$ then $s \models_{As} \varphi$ iff $s \in As(\varphi)$. If $s \in S \setminus \text{end}(M)$ 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 $(s \models_{As} \varphi_1 \text{ or } s \models_{As} \varphi_2)$.
- $s \models_{As} \neg\varphi_1$ iff $s \not\models_{As} \varphi_1$.
- $s \models_{As} \mathbf{A}\mathbf{X}\varphi_1$ iff $\forall s'.(s, s') \in R \Rightarrow s' \models_{As} \varphi_1$.
- $s \models_{As} \mathbf{E}\mathbf{X}\varphi_1$ 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 $\pi = s_0, s_1, \dots$ from s there is a number $i < |\pi|$ s.t. (either $s_i \models_{As} \varphi_2$ or $s_i \in \text{end}(M) \wedge s_i \models_{As} \mathbf{A}(\varphi_1 \mathbf{U}\varphi_2)$), and $\forall 0 \leq j < i [s_j \models_{As} \varphi_1]$.

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

- $s \models_{As} E(\varphi_1 U \varphi_2)$ iff there exist a (maximal) path $\pi = s_0, s_1, \dots$ from s and a number $i < |\pi|$ s.t. (either $s_i \models_{As} \varphi_2$ or $s_i \in \text{end}(M) \wedge s_i \models_{As} E(\varphi_1 U \varphi_2)$) 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 6 *Given an assumption function As over a structure M we define a function $MC[M, As] : cl(\psi) \rightarrow (2^S \cup \{\perp\})$ so that for any $\varphi \in cl(\psi)$, if $As(\varphi) = \perp$ then $MC[M, As](\varphi) = \perp$. 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* [5, 19, 2], after adapting it to the semantics under assumptions.

3 The Programming Language

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

Definition 7 *We assume a fixed set of program variables over some finite domain D . A program fragment is one of $x := \{e_1, \dots, e_k\}$, *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 $x := \{e_1, \dots, e_k\}$ is a non-deterministic 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 $Prog; \text{fin}$ 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 $P \in \text{NWP}$ 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 can be found in [13].

Let $P \in \text{NWP}$ be a program such that $x_1, \dots, x_n$ are the program variables. An assignment to the program variables is some $\sigma \in D^n$. We create a Kripke structure $\text{struct}(P)$ 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 $\text{init}(P)$ as the set of states with location P . The set of ending states, $\text{end}(P)$, is the set of states in $\text{struct}(P)$ that have no outgoing transition. If P is a full program then $\text{end}(P) = \emptyset$. 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 $\mathcal{L} : S \rightarrow 2^{AP}$ is extended accordingly.

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

Fig. 1. An example of an NWP program.

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 $\text{struct}(P')$. 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 .

There are three types of edges: null-edges, yes-edges, and no-edges, denoted $M_1 \rightarrow M_2$, $M_1 \xrightarrow{yes} M_2$ and $M_1 \xrightarrow{no} M_2$ 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 $P = \text{"if } B \text{ then } P_1 \text{ else } P_2 \text{ fi"}$, 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 $P = \text{"while } B \text{ do } P_1 \text{ od"}$, we create a partition graph $G_1 \in pg(P_1)$ and again a node 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).

The formal definition of partition graphs and their semantics (given as Kripke structures) can be found in [13]. 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.

Fig. 2. Creation of partition graphs

Fig. 3. An example partition graph for the program $P;\text{fin}$, where P is the program from figure 1.

Given a partition graph G we define $\text{init}(G)$ to be the set of initial states in $\text{struct}(G)$ and $\text{end}(G)$ to be the set of ending states in $\text{struct}(G)$. 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 $e = M_B \xrightarrow{yes} M_1$ or $e = M_B \xrightarrow{no} M_1$ (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 8 Let $e = M_B \xrightarrow{yes} M_1$ be an edge in a partition graph G s.t. $M_1 = \langle S_1, R_1, I_1 \rangle$ and $M_B = \langle \hat{S}_B, \emptyset, S_B \rangle$, and let $As : cl(\psi) \rightarrow (2^{S_1} \cup \{\perp\})$ be an assumption function over M_1 . $\mathcal{T}(e, As) = As'$ s.t. $As' : cl(\psi) \rightarrow (2^{S_T} \cup \{\perp\})$. 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 $As(\varphi) = \perp$ then $As'(\varphi) = \perp$. Otherwise, $As'(\varphi)$ is defined as follows³

- 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 $M_B \xrightarrow{no} M_1$ 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 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

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

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. Special care should be taken when dealing with loops in the partition graph.

We now give an intuitive description of the algorithm **CheckGraph** (where variable names refer to variables in the algorithm). Given a procedure that properly computes $MC[M, As]$, **CheckGraph** 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.

Fig. 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 $P = \text{"if } B \text{ then } P_1 \text{ else } P_2\text{"}$, as in figure 2 B. To check G we first check G_1 and G_2 , and then compute 'backwards' over the step-edges (using the function 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 $P = \text{"while } B \text{ do } P_1 \text{ od"}$, 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 T to get an assumption As_{-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 $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$.

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, and the algorithm stops.

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

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 $As(\varphi) = \perp$ then $As'(\varphi) = \perp$, 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 $P = \text{"if } B \text{ then } P_1 \text{ else } P_2 \text{ fi"}$ (Figure 2 B) perform:
 1. $As_1 \leftarrow CheckGraph(G_1, As)$.
 2. $As_2 \leftarrow CheckGraph(G_2, As)$.

3. $As_B \leftarrow T(M_B \xrightarrow{yes} in_1, As_1)$
 4. $As_{\neg B} \leftarrow T(M_B \xrightarrow{no} in_2, As_2)$
 5. For every formula $\varphi \in cl(\psi)$, if $As_B(\varphi) = \perp$ then define $As'(\varphi) = \perp$ 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 $P = \text{"while } B \text{ do } P_1 \text{ od"}$ (Figure 2 C) perform:
1. $As_{\neg B} \leftarrow T(M_B \xrightarrow{no} M_E, As)$
 2. Find an ordering $\varphi_1, \varphi_2, \dots, \varphi_n$ of the formulas in $cl(\psi)$ such that each formula appears after all of its sub-formulas. Set $As'(\varphi_i) = \perp$ for all i . For $k = 1, \dots, n$ perform step 3 to define $As'(\varphi_k)$ ⁵.
 3. Perform one of the following, according to the form of φ_k :
 - $\varphi_k \in AP$: $As'(\varphi_k) \leftarrow \{s \in init(G) \mid \varphi_k \in \mathcal{L}(s)\}$
 - $\varphi_k = \neg\varphi_l$: $As'(\varphi_k) \leftarrow \{s \in init(G) \mid s \notin As'(\varphi_l)\}$.
 - $\varphi_k = \varphi_l \vee \varphi_m$: $As'(\varphi_k) \leftarrow As'(\varphi_l) \cup As'(\varphi_m)$.
 - $\varphi_k \in \{\mathbf{AX}\varphi_l, \mathbf{EX}\varphi_l\}$:
 - (a) $Tmp \leftarrow \text{CheckGraph}(G_1, As')$
 - (b) Let $R_{M_B \xrightarrow{yes} in_1}$ be the set of transitions induced by the edge $M_B \xrightarrow{yes} in_1$.
 $As'(\varphi_k) \leftarrow As_{\neg B}(\varphi_k) \cup \{s \in init(G) \mid \forall s' \in init(G_1), (s, s') \in R_{M_B \xrightarrow{yes} in_1} \Rightarrow (s \models B \wedge s' \in Tmp(\varphi_l))\}$ ⁶.
- $\varphi_k \in \{\mathbf{A}(\varphi_l \mathbf{U} \varphi_m), \mathbf{E}(\varphi_l \mathbf{U} \varphi_m)\}$:
- (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:
 - $Tmp = \text{CheckGraph}(G_1, As^{i-1})$
 - $As_B^i \leftarrow T(M_B \xrightarrow{yes} in_1, Tmp)$
 - Define As^i so that for all $j < k$, $As^i(\varphi_j) \leftarrow As^0(\varphi_j)$ and $As^i(\varphi_k) \leftarrow As_B^i(\varphi_k) \cup As_{\neg B}(\varphi_k)$
 - $i \leftarrow i + 1$
 as long as $As^i \neq As^{i-1}$.
 - (c) $As'(\varphi_k) \leftarrow As^i(\varphi_k)$
4. Return As' .

Theorem 1 For any full program P , CTL formula ψ , partition graph $G \in pg(P)$ and empty assumption function $As : cl(\psi) \rightarrow \{\emptyset\}$, if $As' = \text{CheckGraph}(G, As)$ then for every $\varphi \in cl(\psi)$ and $s \in init(G)$, $s \in As'(\varphi) \Leftrightarrow s \models \varphi$.

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

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. $(s, s') \in R_{M_B \xrightarrow{yes} in_1}$.

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*.

The space requirements of our modular algorithm will usually be 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 requirements depend on the model checking algorithm used for a single node and on the partition graph. However, experimental results show a save in time as well as space requirements, as described in Section 6.

6 Implementation and Future Research

We implemented the ideas described in this paper in a prototype tool called *SoftVer*. The tool is based on a BDD representation of models and on symbolic model checking [2].

In order to evaluate the effectiveness of partitioning on memory and time requirements, we applied the tool to a few small examples. Each example program has been checked with two different partitionings. The moderate partitioning divided the program into a few components, while the extensive partitioning further divided it into smaller components. For comparison we also checked the unpartitioned full program.

The largest overhead occurs when applying our algorithm to a program in which the body of a while loop is partitioned. Therefore, all our examples include while loops which are divided by both partitionings. Figure 5 gives the space and time used by each partitioning as a percentage of the space and time used by the unpartitioned example.

	SPACE			TIME		
	Ex 1	Ex 2	Ex 3	Ex 1	Ex 2	Ex 3
No Partitioning	100%	100%	100%	100%	100%	100%
Moderate Partitioning	47%	50%	64%	31%	59%	74%
Extensive Partitioning	33%	34%	47%	18%	26%	176%

Fig. 5. Space and time requirements for three examples.

Example 1 is a simulation of the “stop and wait” communication protocol. It consists of a large while loop whose body has two major parts, one for the sender

and one for the receiver. The moderate partitioning separates the receiver and the sender into two components. The extensive partitioning further separates different tasks in each of the components. The specification we checked required that the sender does not move to the next message before the current message has arrived at the receiver.

Example 2 is a learning algorithm that learns a single term by examples. As in the previous example, the code is partitioned so that different tasks are put in separate components. The specification required that the algorithm will never give a false negative result, i.e. the algorithm does not make errors on inputs for which the term evaluates to *true*.

Example 3 is the shortest of the three and it performs bubble sort on an array of 5 elements. This program consists of two nested while loops, with a single “if” statement in the body of the inner loop (that compares two adjacent elements). The moderate partitioning breaks the outer loop and the extensive partitioning breaks the inner loop as well. The specification states that the algorithm will terminate, and at that point the array will be sorted.

As can be seen in the table, in all examples space reduction was achieved by both partitionings where further save was obtained with finer ones. More surprisingly, save in time was also obtained in five out of six partitionings. This is explained by the fact that BDD based model checking is polynomial in the size of the BDD representation. Thus, applying the check once to a large model requires significantly more time than applying the check many times to smaller models. The save obtained this way compensates for the overhead of the modular algorithm. In the extensive partitioning of Example 3, however, the resulting components were only slightly smaller than those of the moderate partitioning. Thus, in this case the reduction in component size could not compensate for the additional overhead.

We consider a few directions for future research. One is to extend the language to include procedures and parallel composition. Another is to develop heuristics for optimal partitioning of the program text. Finally, we intend to apply *SoftVer* to large realistic examples.

Acknowledgment: We thank Sergio Campos for sharing with us his knowledge in compiling programs into Kripke structures. Arie Tal is also thanked for his help in the tool development.

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. 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.

3. Olaf Burkart and Bernhard Steffen. Pushdown processes: Parallel composition and model checking. LNCS 836, pages 98–113. Springer, 1994.
4. 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.
5. 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.
6. 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.
7. 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.
8. P. Godefroid. Model checking for programming languages using VeriSoft. In *Principle of Programming Languages*, January 1997.
9. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.
10. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International Editors, 1991.
11. 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.
12. 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.
13. Karen Laster and Orna Grumberg. Modular model checking of software. Technical report, Computer Science Dept., Technion. Can be found at: <http://www.cs.technion.ac.il/users/orna/>.
14. 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.
15. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
16. 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.
17. A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
18. A. Pnueli. In transition from 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.
19. J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.