Syntax-Directed Model Checking of Sequential Programs

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

January 3, 2002

Abstract

This work presents a syntax-directed, modular approach to temporal logic model checking of sequential programs.

In contrast to hardware designs, the models of software systems might be too large to fit into memory even when they consist of a single sequential unit. Furthermore, even when the model can be held in memory, model checking might exceed the memory capacity of the computer. To avoid the high space requirements for software we therefore suggest to partition the text of a sequential program into *sequentially composed subprograms*. Based on this partition, we present a model checking algorithm for sequential programs 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.

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 step towards making temporal logic model checking useful for verification of sequential programs.

1 Introduction

This work presents a new syntax-directed modular approach for temporal logic model checking of non-deterministic sequential finite-state programs. Our work applies model checking to programs while exploiting their textual structure in order to reduce space requirements. We consider this work as a step towards making model checking applicable to realistic sequential programs.

Finite-state programs can be useful for describing, in some level of abstraction, many interesting systems. Real life programs, such as communication protocols and controllers (e.g. of telephone switchboards or railway tracks), can be made finite by applying abstraction. Moreover, the resulting program can still be quite large and complicated, and important properties of the system can be verified on the finite-state model.

A first step in verification is choosing a specification language. Temporal logics [29], 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 [15] can be checked by efficient algorithms, called model checking [10, 31, 25, 6]. *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 [27] were successful in finding subtle bugs in real-life designs [28, 11] and are currently in use by the hardware industry in the verification process of newly developed hardware designs [3, 24].

Unfortunately, similar applications of model checking to programs are more difficult. 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 [30, 23, 17]. Most 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 is suitable for 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. In Section 7 we discuss possible extension of our method to other language constructs.

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, 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 results section we summarize these results and explain them.

The paper is organized as follows. In section 2 we introduce the temporal logic CTL, define its semantics, and give a short description of CTL model checking. Section 3.1 describes our programming language and Section 3.2 defines partition graphs. Section 4 defines assumption functions and then Section 5 gives the modular model checking algorithm. Section 6 presents the results of using our algorithm on several example, and Section 7 discusses possible extensions. Finally, Section 8 surveys related work

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

and gives our conclusions.

2 Basic Definitions

2.1 Models of Systems

Most model checking algorithms use *Kripke structures* to represent the computations of a finite-state system. Kripke structures are finite graphs in which nodes represent system states and edges represent possible transitions between states.

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 *computation path* (or simply 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.

When we use a Kripke structure to represent the possible computations of a program, every state is a pair (l, σ) where l is a program location (a value for the program counter) and σ is a valuation to the program variables. We elaborate on this later on.

Kripke structures usually come with a set AP of *atomic propositions* and a labelling function $L : S \rightarrow 2^{AP}$ that associates each state in the structure with the set of atomic propositions that hold in that state. This is used as a basis for specifications, because it gives different attributes to different states in the structure. In this work we use boolean expressions over system variables as atomic propositions, and hence we do not need a specific labelling function. For each state (which includes values for system variables) we know whether the state satisfies a given expression or not.

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 Specifications

We consider specifications in CTL, which is one of several specification languages globally referred to as *propositional temporal logics*. The feature that makes temporal logics appealing for automatic verification is that they can describe intricate behavior over time (as opposed to input-output specifications, for example, that only compare the starting and ending points of programs). This makes them most suitable for specifying reactive systems such as operating systems and communication protocols.

Definition 2.3 gives the syntax of CTL formulas, and definition 2.4 gives the semantics.

Definition 2.3: We assume a set AP of atomic propositions, which are boolean expressions over program variables. The set of *CTL formulas* is the smallest set that includes:

- Atomic propositions $p \in AP$, $true$ and $false$,
- $\varphi_1 \vee \varphi_2$ and $\neg\varphi_1$,
- $\mathbf{AX}\varphi_1$ and $\mathbf{EX}\varphi_1$,
- $\mathbf{A}(\varphi_1 \mathbf{U} \varphi_2)$ and $\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$

where φ_1 and φ_2 are CTL formulas.

We follow a standard definition of CTL semantics (see e.g. [9]) and define it with respect to Kripke structures in which all maximal paths are infinite. Satisfaction of a CTL formula φ in a state s of a Kripke structure M is denoted $M, s \models \varphi$. The relation $\models$ is defined below, where M is omitted for brevity.

Definition 2.4:

- $s \models true$ and $s \not\models false$. For a boolean expression $p \in AP$, $s \models p$ is defined in the usual manner.
- $s \models \varphi_1 \vee \varphi_2$ iff ($s \models \varphi_1$ or $s \models \varphi_2$)
- $s \models \neg\varphi_1$ iff $s \not\models \varphi_1$
- $s \models \mathbf{AX}\varphi$ iff for every maximal path $\pi = s_0, s_1, \ldots$ such that $s = s_0$,it holds that $s_1 \models \varphi$
- $s \models \mathbf{EX}\varphi$ iff there exists a maximal path $\pi = s_0, s_1, \ldots$ such that $s = s_0$ and $s_1 \models \varphi$
- $s \models \mathbf{A}(\varphi_1 \mathbf{U} \varphi_2)$ iff for every maximal path $\pi = s_0, s_1, \ldots$ such that $s = s_0$ there exists a number $i \geq 0$ such that $s_i \models \varphi_2$ and for every $0 \leq j < i$ $s_j \models \varphi_1$.
- $s \models \mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$ iff there exist a maximal path $\pi = s_0, s_1, \ldots$ and a number $i \geq 0$ such that $s = s_0$, $s_i \models \varphi_2$, and for every $0 \leq j < i$ $s_j \models \varphi_1$.

We say that a Kripke structure M satisfies a formula φ, denoted $M \models \varphi$, if for every initial state $s \in I$, $M, s \models \varphi$. We can also define the following abbreviations:

- $\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$
- $\mathbf{AF}\varphi \equiv \mathbf{A}(true \mathbf{U} \varphi)$, $\mathbf{EF}\varphi \equiv \mathbf{E}(true \mathbf{U} \varphi)$
- $\mathbf{AG}\varphi \equiv \neg\mathbf{EF}\neg\varphi$, $\mathbf{EG}\varphi \equiv \neg\mathbf{AF}\neg\varphi$

The CTL language is sometimes defined without the **AX** and **EX** operators, especially in the context of distributed systems. We define these operators because they can be useful for programs, to describe the execution of a single step. For example, one might write a specification that states "$\mathbf{AG}(p \wedge (pc = l) \rightarrow \mathbf{AXAG}q)$, meaning that if p is true at point l, then after the command at l is executed q will always be true. The difference between **AX** and **EX** would only be noticed when the command is an input. In any case, adding these operators is extremely simple, and adds no complexity to our algorithm, so there is no need to exclude them.

Definition 2.5: The *closure* of a formula ψ, denoted by $cl(\psi)$, is the set of all sub-formulas of ψ (including itself).

2.3 Model Checking

We say that a Kripke structure is a model of (satisfies) a formula f if every state $s \in init(M)$ satisfies the formula. A *Model Checking* algorithm is an (automatic) algorithm that decides whether a given structure M is a model of a given formula f.

There are two types of model-checking algorithms. *Explicit-state* algorithms are those that use an explicit representation of the Kripke structure being examined. Usually, these algorithms use a *next-state function*, which is a function that given a state returns the set of successors of this state. Temporal logic model-checking can be performed using a DFS or BFS style traversal of the reachable state-space of the structure, using the next-state function. For such algorithms it is usually not necessary to build in advance the set of states or the transition relation of the structure being examined.

The second type of algorithms is *symbolic* model checking. In symbolic model checking the transition relation of the structure is represented using BDDs [5]. A BDD is a directed acyclic binary-decision graph in which each internal node is labelled with a BDD variable. A BDD is an efficient representation for boolean functions. Each state of the Kripke structure is encoded using BDD variables. A set of states is represented by the boolean function that gives true iff the input vector is the encoding of a state in the set. In a similar manner, transition relations are represented as sets of pairs of states.

The importance of BDDs is that in many cases (although not all) they give a polynomial size representation of the transition relation. Also, they allow the execution of operations on sets of states, instead of traversing the structure one state at a time. For these reasons BDDs and symbolic model-checking have proved to be extremely useful verification methods.

In this work we focus mainly on symbolic model checking. The relationship of our work to explicit state methods is examined in the conclusions (Section 8).

3 Sequential Decomposition

3.1 Control Structures

We notice that sequential imperative programming languages are based on a limited set of control structures. We utilize these structures to decompose the program sequentially. We use three basic control structures: sequential composition (";"), choice ("if"), and loops ("while"). These structures are used to connect *sequential blocks*, which are portions of the program that manipulate data, and in which the program counter advances by 1 in each step. In section 7 we discuss how our algorithm relates to other, more advanced, features of programming languages.

Definition 3.1: A program is defined by:

Prog $\rightarrow$

$Prog_1$; $Prog_2$ |
if B then $Prog_1$ else $Prog_2$ fi |
while B do $Prog_1$ od |
$Simp$

where B is a boolean expression over the program variables, and $Simp$ is any simple command.

Simple commands are assignments (either deterministic or non-deterministic), inputs, outputs, etc. These are commands that manipulate data and not control. We deliberately avoid defining the syntax of simple commands so as to emphasize the fact that our algorithm is not specific to a certain programming language.

Given a program P we use *Loc* to denote the set of program counter locations in P. Each statement (command) in the program is associated with a location, which represents the point in the program execution where the next thing to do is execute this statement. To denote the point of termination of the program, we add an extra location l_{end}, which is the location arrived at when the program terminates, after the last command has been executed. We artificially add a command "terminate" at the location l_{end} to help us identify the point of termination. The semantics of this command is a self loop - it does nothing to the program variables, and does not change the value of the program counter. This addition is done to ease the discussion and presentation of the algorithm, and presents no restriction on implementing the method on existing languages (that may not have a terminate command).

The semantics of programs is defined by associating each program P with a Kripke structure *struct*(P) that represents its behavior. The set of states of this structure is $S = Loc \times \Sigma$, where Σ is the set of all possible valuations to the program variables. Each state in the program is a pair (l, σ) such that l is the value of the program counter *pc*, and σ is a valuation from Σ. The set of initial states is $I = l_{start} \times \Sigma$, where l_{start} is the initial program counter location in the program. The transition relation R is defined according to the semantics of the program. We do not formally define it here, but it should be clear that this transition relation can easily be created by a simple compilation of the program text.

In the following we will often relate to the flow of control in the program. To assist us, we define the function $Next$ that given a program counter location l gives us the location of the next statement to be executed after the command at l. Here we use a slight abuse of notation. If the command at l is a simple command, then $Next(l)$ gives the next command after it. If l is an "if" or "while" command then $Next(l, T)$ gives the location arrived at if the boolean condition is true, which is the "then" of an "if" command, or the body of the "while" command. Finally, $Next(l, F)$ gives the location arrived at if the boolean condition is false, which is the "else" for an "if" or the next command after the "while". Notice that if l is the location of the last command in the body of a loop, then $Next(l)$ gives us the location of the head of the loop (which is the location of the while statement). Also, if l is the last statement in the "then" or "else" parts of an "if" command, then $Next(l)$ is the location of the first command after the "if".

Before we continue to decompose the structure of a program, we notice a few characteristics of the structures of programs.

- The structure of a program is not deterministic. The interpretation of an input command is a non-deterministic choice of value (in the next step) for the input variable. For example, given a state $s = (l, \sigma)$ where l is the location of the command "input(v)", the successor states of s are all the states (l', σ') where $l' = Next(l)$, and σ' is identical to σ except (perhaps) for the value of v. The control structures of the program - the sequential composition, "if", and "while" - are all deterministic. Later, in Section 7, we investigate the possibility of non-deterministic choice.

- A state (l, σ) represents a situation in which the execution of the program has reached l while σ is the value for the variables. At this point, the command at l has not yet been executed. The transitions exiting this state represent the execution of the command.

- When creating the structure of a sub-program P_1 of some program P, we create a structure that includes the *execution* of all the commands in P_1. That is, if l is the location of the last command in P_1, and $l \neq l_{end}$, then $struct(P_1)$ includes the set of all states with location $Next(l)$ which are the result of executing the command at l. Notice that the location $Next(l)$ does not appear in the text of P_1, since it is the location of a command that is not a part of P_1. However, the execution of l *is* part of P_1, and this is represented by transitions into states with location $Next(l)$.

- Given a program P, all the (maximal) computation paths in P are infinite. This is because we added the "terminate" command at the end. However, a sub-program P_1 of P, might not end with "terminate". In this case the structure $struct(P_1)$ will include states that have no outgoing transitions. The set $end(struct(P_1))$ is the set of states with the location of the next command to be executed after P_1.

Figure 1 gives an example of a program, and the Kripke structure that represents its semantics.

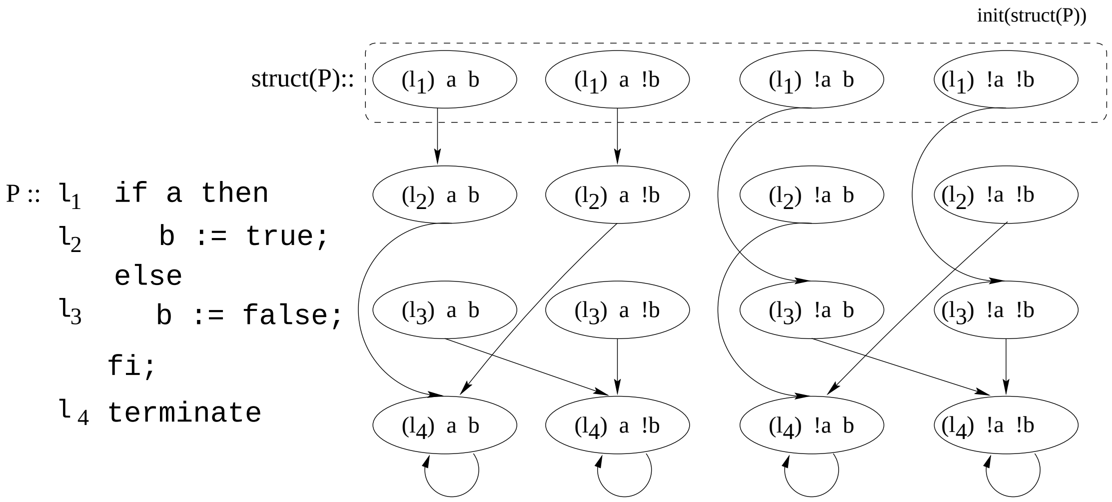


Figure 1: A sequential process and its Kripke structure.

3.2 Partition Graphs

A *Partition Graph* of a program P is a finite directed graph representing a decomposition of P into several sub-programs while maintaining the original flow of control. It is a natural extension of a *control flow graph*. The nodes of the graph are labelled with sub-programs of P or boolean conditions. A node labelled with a sub-program represents the execution of this program, and has one outgoing edge. A node labelled with a boolean condition represents the program counter location in which the condition is to be evaluated. We also add dummy nodes with no labelling which are used to maintain structure, but have no semantic meaning (they do not represent execution of commands). There are three types of edges: null-edges, true-edges, and false-edges, denoted $n_1 \rightarrow n_2$, $n_1 \xrightarrow{true} n_2$ and $n_1 \xrightarrow{false} n_2$ respectively. True-edges and false-edges, also called *step-edges*, represent the execution step in which a condition is evaluated and the program counter is updated accordingly. Each condition node has two outgoing edges - one true-edge and one false-edge. We choose to associate the execution step of the condition with the edges, rather than with the node from which they exit, because it gives a natural separation between the manipulation of states in which B is true and states in which B is false. This is particularly important in the while loop. A null-edge always exits a node labelled with a sub-program (or a dummy node). There is no execution associated with it.

Notice that in partition graphs a single node may be labelled with a complicated sub-program, with "if"s and "while"s in it. This is opposed to traditional control-flow graphs, which normally allow a single node to represent a sequential block with no internal structure.

We start by defining the set of all possible partition graphs for a program, and then define the semantics of partition graphs by associating a Kripke structure with every partition graph.

Every partition graph 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 leave it as a single node. In particular, we may decide to leave P itself as a single node, thus $P \in pg(P)$. 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. Every time a node n is partitioned into a graph G, all the edges that entered n will enter the initial node of G, and all the edges that exited n will exit from the exit node of G.

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 is in_1, and the exit node is 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 labelled with B and the other a dummy node with no labelling. The entry node is the B node, and the exit node is the dummy node. The edges connecting the different components are according to the semantics of the "if" command, i.e. a true-edge from the B node to G_1, a false-edge from the B node to G_2 and null-edges from G_1 and G_2 to the dummy node. The edges entering G_1 and G_2 are pointing 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 for B, which is the entry node, and a dummy node as the exit node. The edges represent the semantics of the "while" loop, i.e. a true-edge from the B node to G_1, a false-edge from the B node to the dummy node, and a null edge from G_1 to the B node. (Figure 2 C).

A partition graph is used to represent a decomposition of $struct(P)$ into several structures of sub-programs. We associate a Kripke structure with each element of the graph.

- For a node n labelled with a sub-program P' of P, the structure associated with n is the structure of the sub-program: $struct(n) = struct(P')$.

- A node n labelled with a boolean expression B represents the execution of an "if" or "while" command that evaluates this condition. The structure associated with this node is simply the set of states in which B is evaluated. Formally, let l be the program location of this statement. Then $struct(n) = \langle S, R, I\rangle$ such that $R = \emptyset$ and $S = I = \{(l, \sigma) \mid \sigma \in \Sigma\}$.

- The Kripke structure associated with the dummy node is empty because this node does not represent an actual execution step, but rather is added so that each sub-graph will have only one exit point. In the end there will be a null-edge from the dummy node to some real node n. Every edge that enters the dummy node is considered as if it entered n.

- A null edge $n_1 \to n_2$ does not reflect an execution step in itself, and therefore if $M_1 = struct(n_1)$ and $M_2 = struct(n_2)$ then $end(M_1) = init(M_2)$ (every ending state of M_1 is an initial state of M_2). For this reason, there is no structure associated with null-edges.

- Step-edges are the edges outgoing from a node labelled by a boolean expression B. Let l be the program location of the "if" or "while" statement that evaluates B. Execution from a state in a node labelled B continues through the true-edge or the false-edge, depending on whether the expression evaluates to $true$ or $false$ in that state. These edges represent an actual step in the execution of P, and the structures associated with them capture this step. For a true-edge $e = n_1 \xrightarrow{true} n_2$, let $l' = Next(l, T)$. The structure associated with e is $struct(e) = \langle S, R, I\rangle$ where $S = \{l, l'\} \times \{\sigma \mid \sigma \models B\}$, $R = \{((l, \sigma), (l', \sigma)) \mid \sigma \models B\}$ and $I = \{(l, \sigma) \mid \sigma \models B\}$. For $e = n_1 \xrightarrow{false} n_2$ the definition is similar, except that $l' = Next(l, F)$, and every $\sigma \models B$ becomes $\sigma \not\models B$.

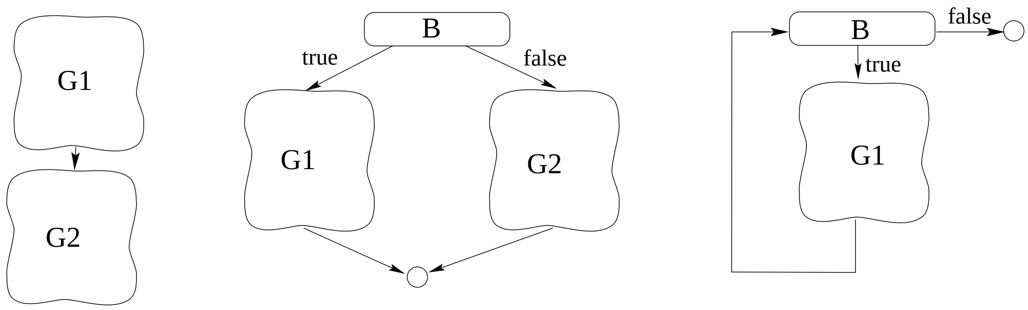
Given a partition graph $G \in pg(P)$, the structure that defines its semantics, denoted $struct(G) = \langle S, R, I\rangle$ is constructed as follows. S is the union of the sets of states over the structures associated with all its nodes and edges. Similarly, R is the union of the transition relations over all these structures. Finally, assume that the entry node of G is in then $I = init(in)$.

The resulting structure is exactly $struct(P)$, the Kripke structure representing the program. Notice that the connection between structures of sub-programs is through states that appear in more than one node. For example, given a partition graph of $P_1; P_2$ that has two nodes, one for P_1 and one for P_2, the structure for P_1 will include the set of states with the location of the beginning of P_2, because these are the ending states of $struct(P_1)$ (the execution of the last command in P_1 is represented by transitions that enter this location). The same states will also appear in the structure of P_2 because they are the initial states of this sub-program.

We define $init(G)$ to be the set of initial states I 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, including the Kripke structures associated with the nodes. The structures associated with the step-edges are omitted from the drawing, to make it readable.

4 Assumption Functions

In this section we define assumption functions and model-checking under assumptions.



(A) P = P1;P2 (B) P = if B then P1 else P2 fi (C) P = while B do P1 od

Figure 2: Creation of partition graphs

We recall that a sub-program does not necessarily end with the "terminate" command, and therefore the Kripke structure representing it may have a non-empty set of ending states. To create a modular model checking algorithm we need to be able to perform model checking on a sub-program, where some of its computations continue in another sub-program. We introduce assumption functions as a means to hold information about the ending states. Assumption functions tell us which formulas each ending state satisfies. We will later show how assumption functions are created and used.

Let ψ be a CTL formula, which we wish to check on a given program.

Definition 4.1: An *assumption function* for the Kripke structure $M = \langle S, R, I \rangle$ is a function $As : cl(\psi) \rightarrow (2^{S'} \cup \{\perp\})$ where S' is some subset of the set of states S. We require that $\forall \varphi \in cl(\psi)$, if $As(\varphi) \neq \perp$ then $\forall \varphi' \in cl(\varphi)$, $As(\varphi') \neq \perp$. When $As(\varphi) = \perp$ it means that we have no knowledge regarding the satisfaction of φ in S'. It is used when we want to represent knowledge relating to other sub-formulas and ignore φ at this stage. 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. The significance of the set S' is that it is the set of states which we examine. The assumption function gives no information regarding states outside of S'.

We say that As is an assumption function over a set of states A if A is the set S' about which As gives information, i.e. it is defined as $As : cl(\psi) \rightarrow (2^{A} \cup \{\perp\})$.

Satisfaction of a CTL formula φ in a state $s \in S$ under an assumption function As is denoted $M, s \models_{As} \varphi$ [2]. Satisfaction of formulas in M under an assumption As is defined only when the assumption As is defined over a set that includes $end(M)$. We define it so that it holds if either $M, s \models \varphi$ directly, or through the assumption function. For example, $M, s \models_{As} \mathbf{E}(f\mathbf{U}g)$ if there exists a 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'

[2]When no confusion may occur we omit M.

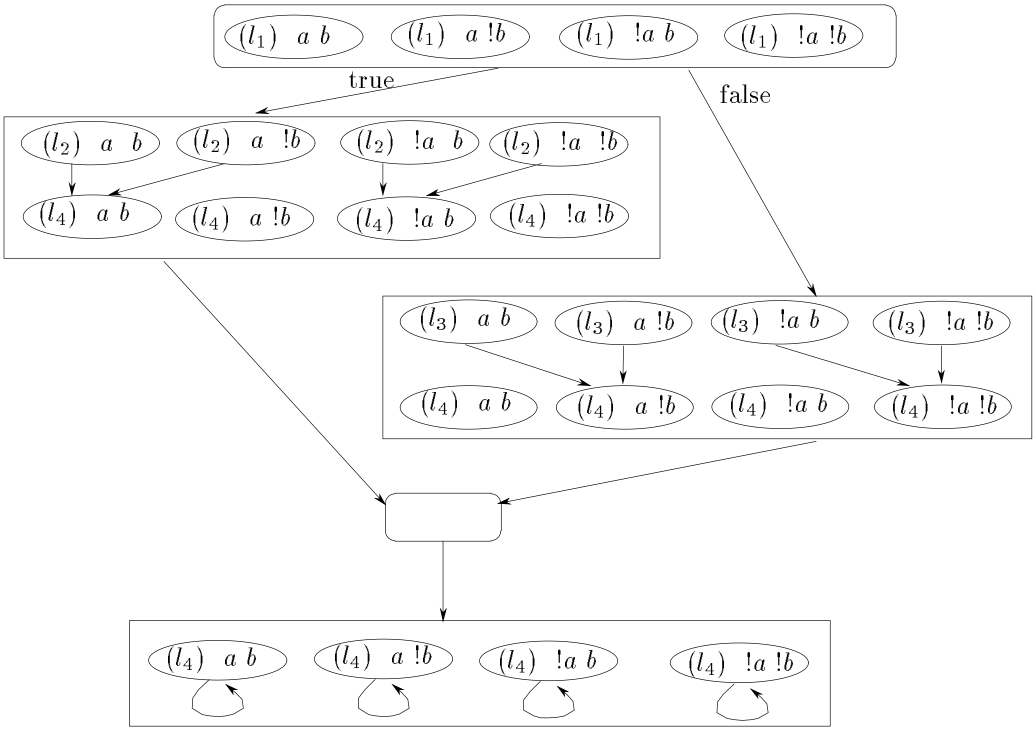


Figure 3: An example partition graph

This is a partition graph for the program P from figure 1. Instead of writing in each node the sub-program that it represents, we show the structure associated with that node. The only thing missing are the transitions associated with the step-edges.

satisfy f. Formally:

Definition 4.2: Let $M = \langle S, R, I \rangle$ be a Kripke structure and As an assumption function over a set S' such that $end(M) \subseteq S'$. 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 $s \notin end(M)$ then $s \models_{As} \varphi$ is defined as follows:

- For every $p \in AP$, $s \models_{As} p$ iff $s \models p$.
- $s \models_{As} \varphi_1 \vee \varphi_2$ iff ($s \models_{As} \varphi_1$ or $s \models_{As} \varphi_2$).
- $s \models_{As} \neg\varphi_1$ iff $s \not\models_{As} \varphi_1$ [3].
- $s \models_{As} \mathbf{AX}\varphi_1$ 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 $i < |\pi|$ such that:
either ($s_i \models_{As} \varphi_2$) or ($s_i \in end(M) \wedge s_i \in As(\mathbf{A}(\varphi_1 \mathbf{U} \varphi_2))$), and $\forall 0 \le j < i$ $[s_j \models_{As} \varphi_1]$.
- $s \models_{As} \mathbf{E}(\varphi_1 \mathbf{U} \varphi_2)$ iff there exists a maximal path $\pi = s_0, s_1, \ldots$ from s and a number $i < |\pi|$ such that:
either ($s_i \models_{As} \varphi_2$) or ($s_i \in end(M) \wedge s_i \in As(\mathbf{E}(\varphi_1 \mathbf{U} \varphi_2))$), and $\forall\, 0 \le j < i$ $[s_j \models_{As} \varphi_1]$.

Note that if the transition relation of M is total (i.e., all its maximal paths are infinite) 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. Since the assumption function is consulted only for states in $end(M)$ it seems pointless at this point to define an assumption function over a set S' that includes more than just the ending states. However, later on we use assumption functions as (intermediate) results of model checking, and then this possibility is used.

We write $M \models_{As} \psi$ iff $\forall s \in I, [M, s \models_{As} \psi]$. We can now define model-checking under assumptions.

Definition 4.3: Given a structure $M = \langle S, R, I \rangle$, and an assumption function As over a set that includes $end(M)$, we define a function $MC[M, As] : cl(\psi) \to (2^S \cup \{\bot\})$ so that for any $\varphi \in cl(\psi)$, if $As(\varphi) = \bot$ then $MC[M, As](\varphi) = \bot$. Otherwise, $MC[M, As](\varphi) = \{s \in S \mid M, s \models_{As} \varphi\}$.

One can view MC as an operator over assumption functions. The result of $MC[M, As]$ is an assumption function over S that given a formula φ produces the set

[3] Since $As(\neg\varphi) \neq \bot$, which is why $s \models_{As} \neg\varphi$ is defined, we conclude that $As(\varphi) \neq \bot$. This means that the set of states that satisfy φ is defined, and $s \not\models_{As} \varphi$ iff s is not in this set.

of all states in M that satisfy φ under the assumption As. Given M and As, this function can be created using any known model checking algorithm for CTL (e.g. [10, 6]), after adapting it to the semantics under assumptions. For example, $MC[M, As]$ can be calculated using BDDs since the modifications needed can be implemented using efficient BDD operations.

In the following, we use MC as a basis for our algorithm. We do not specify the exact algorithm used to perform model-checking under assumptions - our modular approach can theoretically be used with any algorithm. However, we do note that it will be most efficient using a symbolic algorithm.

4.1 Operations on Assumption Functions

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

The most basic operation is performing model checking under assumptions on a single partition-graph node n labelled by a sub-program P'. For this we use MC, which was defined earlier. Given an assumption function As over the ending states of $struct(P')$ we can calculate the assumption function $As' = MC[struct(P'), As]$. The function As' gives us full knowledge about which states of $struct(P')$ satisfy which formulas, under the assumption As.

Next we present a function called $CheckStepEdge$, which performs model checking under assumptions over a step edge. The input to this procedure is a step-edge $e = n_1 \xrightarrow{true} n_2$ or $e = n_1 \xrightarrow{false} n_2$ and an assumption function As over a set including the initial states of n_2 (which are the ending states of $struct(e)$) [4]. The output is the assumption function As' over $struct(e)$ calculated by: $As' = MC[struct(e), As]$. Notice that if e is a true-edge (false-edge) then As' is defined only on states that satisfy (do not satisfy) the condition labelling n_1. Actually, when calculating As' for a step-edge there is no need to use a full model-checking algorithm. The structure $struct(e)$ is defined so that each initial state has exactly one successor, and this successor is an ending state. Therefore, we can calculate As' more efficiently according to the following definition:

Definition 4.4: Define S_T to be the set of states in a condition node that satisfy the condition, and S_F the set of states that do not satisfy the condition. If l is the label of the "if" or "while" statement associated with the condition node, and B is the condition evaluated, then $S_T = \{(l, \sigma) \mid \sigma \models B\}$ and $S_F = \{(l, \sigma) \mid \sigma \not\models B\}$.

Definition 4.5: Let $e = n_1 \xrightarrow{true} n_2$ be a true-edge in a partition graph G, and let $As : cl(\psi) \to (2^{S_1} \cup \{\perp\})$ be an assumption function over S_1 such that $init(struct(n_2)) \subseteq S_1$. Define $As' = CheckStepEdge(e, As)$ s.t. $As' : cl(\psi) \to (2^{S_T} \cup \{\perp\})$. Let l be the program location of n_1 (an "if" or "while" statement), let l' be the program location of the beginning of n_2.

[4] The assumption function As must be defined over a set S' that includes the initial states of n_2, but it may include other states as well.

If $As(\varphi) = \bot$ then $As'(\varphi) = \bot$. Otherwise, $As'(\varphi)$ is defined as follows [5] :

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

For a false-edge $n_1 \xrightarrow{false} n_2$ the definition is the same, except that instead of S_T we use S_F.

Lemma 4.1: For any step edge e and assumption function As over S_1 such that $end(struct(e)) \subseteq S_1$, $CheckStepEdge(e, As) = MC[struct(e), As]$.

This statement is obvious from the definitions of model-checking under assumptions and $CheckStepEdge$.

5 Modular Model Checking

In this section we give an algorithm to check a formula ψ on a partition graph G of a 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 how the algorithm works. Variable names which are mentioned refer to variables in the algorithm.

The algorithm works on a partition graph G of a program P, and traverses it from the exit node upwards to the entry node. First the structure $struct(v)$ of a leaf node v of G is checked under an "empty" assumption for $cl(\psi)$, an assumption in which all values are $\emptyset$. Since v is a leaf it must end at the ending location l_{end}, with the "terminate" command that we have added, and therefore all paths in it are infinite. As we saw earlier, in this case the assumption function has no influence on the result. The result obtained for the leaf node is an assumption function As' over $init(struct(v))$ that associates with every sub-formula of ψ the set of all initial states of $struct(v)$ that satisfy that sub-formula. Once we have As' on v we can derive a similar function As over 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 in this way, each node in G can be checked in isolation, based on assumptions derived from its successor nodes. Naturally, we need to take special care when dealing with loops in the partition graph.

The algorithm is called CheckGraph. Given a procedure that properly computes $MC[M, As]$, CheckGraph takes a partition graph G and an assumption function As

[5] If $As(\varphi) \neq \bot$ then for all sub-formulas φ' of φ we are assured that $As(\varphi') \neq \bot$.

over $end(G)$ and performs model checking under assumptions resulting in an assumption function As' over $init(G)$. 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 by using the given procedure MC. 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, using a recursive call (see Figure 4). As_1 is the result of this call (As_1 is over the set $init(struct(G_2))$). It then uses As_1 as an assumption over the ending states of G_1 and checks G_1 w.r.t As_1 using another recursive call. The second call returns for all $\varphi \in cl(\psi)$ such that $As(\varphi) \neq \perp$ the set of all initial states of G_1 (which are the initial states of P) that satisfy φ, which is the desired result.

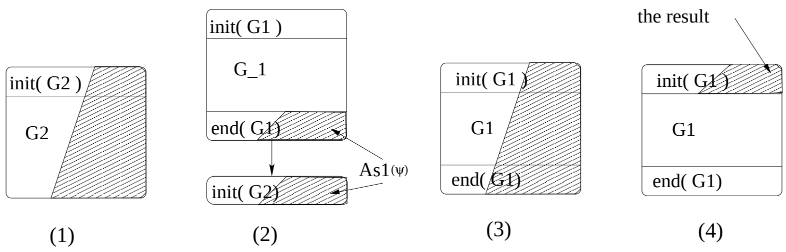


Figure 4: The operation of CheckGraph on sequential composition.
The gray area is the set of states that satisfy ψ.

Let G be a partition graph of $P =$ "if B then P_1 else P_2", as in figure 2 (B). To check G we first check G_1 and G_2, and then compute 'backwards' over the step-edges (using $CheckStepEdge$) 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 dummy node, with the assumption As over the set of states of the dummy node (the set of states with the program counter location of the next command after the while). Walking backwards on the false-edge we use $CheckStepEdge$ to get an assumption $As_{\neg B}$ over the initial states of G that satisfy $\neg B$. We then use recursive calls over the body of the while to compute the assumption function As'.

In this part it is important that when calculating the set of states that satisfy a formula we have already finished all calculations for all of its sub-formulas. For this reason we order the formulas in $cl(\varphi)$ according to their length and operate on them one at a time. We start with an assumption function As' that has a $\perp$ value for all formulas except the shortest, and use a recursive call to evaluate this formula. The recursive call will disregard all formulas for which $As' = \perp$. When this is done, we move on to the next formula, changing the value of As' from $\perp$ to a set of states. We continue this process for each sub-formula in $cl(\varphi)$.

The value $As'(\varphi_k)$ is computed for each $\varphi_k \in cl(\varphi)$, according to the structure of the formula φ_k. The most complicated part here is the computation for the temporal operators **EU** and **AU**. We now demonstrate the computation of $\mathbf{E}(\varphi_1\mathbf{U}\varphi_2)$. The algorithm handles the formulas in $cl(\varphi)$ one at a time, so that when reaching $\mathbf{E}(\varphi_1\mathbf{U}\varphi_2)$ it has already dealt with φ_1 and φ_2. This means that the assumption functions $As'(\varphi_1)$ and $As'(\varphi_2)$ are already calculated (over $init(G)$). Since the algorithm is recursive, every time we apply it to the body of the loop (G_1) it calculates values for all formulas which are not $\perp$ in the input assumption function. Specifically, this means that it will calculate correctly the sets of states in $struct(G_1)$ that satisfy φ_1 and φ_2. The goal now 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 symbolic 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 predecessors in G of these states. Notice that $init(G) = end(G_1)$, because both are defined as $l \times \Sigma$ where l is the program location in which B is evaluated. This means that the predecessors we are looking for are inside G_1. Hence we continue from $end(G_1)$ backwards inside G_1 until we arrive at $init(G_1)$, and keep the result in Tmp, which is an assumption function over $init(G_1)$. We notice that at this point, only the marks on states of $init(G_1)$ are needed to proceed, the marks on all other states of G_1 are not preserved. If and when we pass through G_1 again, some calculations may have to be repeated. We will later discuss how to solve this problem. Notice also 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 until no new states are found. It is essential that the initial 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 begin calculating the proper result for $\mathbf{E}(\varphi_1\mathbf{U}\varphi_2)$.

This process stops when the assumption calculated reaches a fixpoint ($As^i = As^{i-1}$). 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. This is formally proved later.

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)**:**

For a single node n (labelled by a sub-program P'), return As' s.t. $\forall\varphi \in cl(\psi)$, if $As(\varphi) = \perp$ then $As'(\varphi) = \perp$, otherwise

$As'(\varphi) = MC[struct(P'), As](\varphi) \cap init(struct(P'))$.

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), n_B is the condition node (if present), and n_D is the dummy node (if present).

- 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. $As_B \leftarrow CheckStepEdge(n_B \xrightarrow{true} in_1, As_1)$
  4. $As_{\neg B} \leftarrow CheckStepEdge(n_B \xrightarrow{false} in_2, As_2)$
  5. For every formula $\varphi \in cl(\psi)$, if $As_B(\varphi) = \perp$ then define $As'(\varphi) = \perp$ [6]. Otherwise, $As'(\varphi) = As_B(\varphi) \cup As_{\neg B}(\varphi)$
  6. Return As'
- For a graph of $P =$ "while B do P_1 od" (Figure 2(C)) perform:
  1. $As_{\neg B} \leftarrow CheckStepEdge(n_B \xrightarrow{false} n_D, As)$
  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 $As'(\varphi_i) = \perp$ for all i. For $k = 1, \ldots, n$ perform step 3 to define $As'(\varphi_k)$ [7].
  3. To define $As'(\varphi_k)$ perform one of the following, according to the form of φ_k:
     $\underline{\varphi_k \in AP}$: $As'(\varphi_k) \leftarrow \{s \in init(G) \mid s \models \varphi_k\}$
     $\underline{\varphi_k = \neg\varphi_l}$: $As'(\varphi_k) \leftarrow \{s \in init(G) \mid s \notin As'(\varphi_l)\}$.
     $\underline{\varphi_k = \varphi_l \vee \varphi_m}$: $As'(\varphi_k) \leftarrow As'(\varphi_l) \cup As'(\varphi_m)$.
     $\underline{\varphi_k \in \{\mathbf{AX}\varphi_l, \mathbf{EX}\varphi_l\}}$:
     (a) $Tmp \leftarrow$CheckGraph(G_1, As')
     (b) Let l be the location of the "while" and $l' = Next(l, T)$.
     $As'(\varphi_k) \leftarrow As_{\neg B}(\varphi_k) \cup \{(l, \sigma) \mid \sigma \models B \wedge (l', \sigma) \in Tmp(\varphi_l)\}$ [8].

[6] Notice that since As_B and $As_{\neg B}$ both originate from the same assumption function As, it holds that $As_B(\varphi) = \perp$ iff $As_{\neg B}(\varphi) = \perp$. Also, the images of As_B and $As_{\neg B}$ are disjoint.

[7] Notice that when working on φ_k we have already calculated As' for all of its sub-formulas, so they are not $\perp$.

[8] The definition is the same for $\mathbf{AX}\varphi_l$ and $\mathbf{EX}\varphi_l$ because each state $(l, \sigma) \in init(G)$ has exactly one successor state $(l', \sigma) \in init(G_1)$.

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

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

(b) do:
   - $i \leftarrow i+1$
   - $Tmp =$CheckGraph(G_1, As^{i-1})
   - $TmpB \leftarrow CheckStepEdge(n_B \xrightarrow{true} in_1, Tmp)$
   - Define As^i by setting for all $j < k$, $As^i(\varphi_j) \leftarrow As^0(\varphi_j)$ and $As^i(\varphi_k) \leftarrow TmpB(\varphi_k) \cup As_{\neg B}(\varphi_k)$

   Until $As^i = As^{i-1}$.

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

4. Return As'.

Theorem 5.1: The above algorithm computes model checking under assumptions correctly for any partition graph G of any program P. Formally, for any assumption function As: $CheckGraph(G, As) = MC[struct(G), As]$.

The proof of this theorem is quite long, and it is differed to the appendix. The consequence of the above theorem is the following:

Theorem 5.2: For any program P, CTL formula ψ, partition graph $G \in pg(P)$ and empty assumption function $As : cl(\psi) \to \{\emptyset\}$, 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 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.

As promised, we now show how to make the algorithm more efficient by saving on recalculations done in recursive calls. When there is a recursive call to $CheckGraph$ on a smaller graph, all sub-formulas for which the input assumption function is not $\perp$ are calculated, even if they were already calculated in a previous call. To avoid this calculation, the results of calculations during recursive calls can be kept in files. This method is called memoization. Of course, the result for each node must be kept in a separate file to avoid BDD explosion. When there is a second call to the same sub-graph, because this sub-graph is part of the body of a while loop, the input assumption function for sub-formulas that have been calculated before will not change (because we are now working on a larger formula), and so the resulting assumption function for these sub-formulas is identical. When this case is identified, the previous results can be read from the appropriate file instead of making the same calculations again. This scheme may improve the performance of the algorithm substantially, especially in systems where accessing the file is not too big a problem.

Another way of making the algorithm for a while loop more efficient is to work on sub-formulas of the same size together, instead of one at time. Any combination that guarantees that when working on a formula, all of its sub-formulas have already been dealt with, will suffice. The simplest way is to choose to work on all formulas of length 1, then all formulas of length 2, etc.

The space requirements of our modular algorithm will usually be better than that of classical symbolic 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. Furthermore, since in many cases the explosion in space requirements comes from performing a relational product on the transition relation, working on smaller transition relations can in some cases prevent this explosion. Our algorithm needs also to keep 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.

We now analyze the time complexity of our algorithm. We claim that the number of fixpoint iterations performed by our algorithm is the same as the number of fixpoint iterations performed by classical symbolic model checking. Obviously, for formulas that do not involve fixpoint computation and for programs with no loops, the two algorithms work in a similar manner.

When a classical symbolic algorithm performs a least fixpoint computation starting with an initial set of states A, the number of iterations is the length of the longest path from any state to a state in A. This fixpoint computation can be viewed as a backward traversal of this path. In our algorithm, the same path is traversed backwards, only that the traversal is done in parts. The cost of each step in the fixpoint computation is relative to the size of the BDD of the transition relation used. Since at every step we use a part of the transition relation whose BDD is usually smaller, the cost of each step is smaller. On the other hand, when moving between parts we spend time on read and write operations. Obviously, there is a tradeoff between the gain in BDD sizes versus the cost of the read and write operations.

6 Results

6.1 SoftVer

We implemented the modular model checking algorithm in a prototype tool called *SoftVer*. The SoftVer system is a model-checker that implements *modular model checking* for the temporal logic CTL. Its features include:

- A simple, structured, programming language. The language allows boolean, bounded integer, and array types. The control structures are the basic sequential composition, "if", and "while" statements. Input statements are represented by non-deterministic assignments, and output statements by a "no-operation". All the basic boolean and integer operators are supported.

- Control over the partitioning of a program is done by use of a special compiler directive. The token "#MODULE#" marks the beginning of a sub-program, i.e. the top of a node in the partition graph. The directives inserted into a program must adhere to the definition of partition-graphs, or else the compiler issues an error message.
- Verification is done using a BDD based symbolic model checker, which serves as the basic model-checking algorithm. BDD operations are implemented using a BDD library by David Long [26]. This library supports writing BDDs to files, and reading them from files even when the ordering has changed. This library also enables to perform dynamic variable reordering.
- The tool supports a reduced form of reachability calculation, which we call *local reachability*. This option is explained below.
- For a single partition graph node, the model checking algorithm is the classical CTL algorithm, as defined in [6], adapted for model checking under assumptions.
- SoftVer is implemented in the C programming language, on a unix platform (Solaris).
- The implementation is a prototype, and includes no optimizations whatsoever, including the optimizations mentioned in the previous section.

Computing the set of reachable states before performing model checking can sometimes reduce the space requirements needed for model checking, and it is a common practice in symbolic model checking. However, this computation may be expensive in both running time and space requirement. The SoftVer model-checker offers the possibility to perform local reachability, which computes an over approximation of the set of reachable states in one pass over the program text (no fixpoint calculation).

Local reachability is computed while translating the program text into a transition relation. We compute the reachable state-space in the regular way as long as we are translating simple commands (assignments, inputs, outputs), and conditions. After creating the transition relation for a statement we compute the set A of ending states according to this transition relation. We then limit the transition relation of the next statement to include only transitions that start from A. For a program without loops this process will produce the same result as regular reachability. When there is a loop, however, we refrain from computing the fixpoint. The body of the loop is created using local reachability under the assumption that all initial states of the body are reachable (which, of course, might not be true). Using local reachability means that the state-space of our model will now include some unreachable states, but not as much as without reachability at all.

6.2 The examples

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 was 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 un-partitioned full program. In this way we can estimate the effect of partitioning without the influence of implementation details and optimizations.

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. This gives us some indication to the worst case.

Stop and Wait

This program runs 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 sender keeps a buffer ("window") of messages to be sent to the receiver, and the receiver keeps a buffer of acknowledges to be sent back. The communication lines back and forth are implemented by bounded lossy buffers. This is the largest program of the three, and it includes 5 integer variables, 8 boolean variables, 2 boolean arrays of length 3, and 71 statements.

The moderate partitioning separates the code of the receiver and the sender into two components. This requires breaking the loop into a head node (the boolean condition) and two nodes for the body, sequentially composed. The rational behind this is the assumption that since these two sub-programs execute two separate tasks, they refer to different variables. If a variable is not changed in a sub-program then after one variable reordering the BDD for this sub-program will contain a fixed small portion describing this fact. Thus, we expect to get BDDs that are considerably smaller than the BDD of the whole program. The extensive partitioning further separates different tasks in each of the components.

The specification used is a CTL formula that requires that the sender does not move to the next message before the current message has arrived at the receiver.

Learn Monom

This program is a learning algorithm that learns a single boolean term (a conjunction of boolean variables) by examples. Each example is an assignment to the term variables. The algorithm "guesses" whether the term evaluates to true or false on this example, and is then given the correct answer. As it gets more examples, its guesses get better, until it actually knows the term and will not make mistakes anymore. This program includes 1 integer variable, 2 boolean variables, 3 boolean arrays of length 7, and 36 statements.

The moderate partitioning separates the code into several components so that different tasks are in different components (getting an example, guessing, getting the correct answer and analyzing its database). The extensive partitioning breaks down the components even further, almost to the level of single commands. The specification used requires 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*.

Sort
This example is the shortest of the three (in terms of lines of code), 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). It includes 3 integer variables, 1 boolean variable, 1 integer array of length 5, and 25 statements.

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.

6.3 Running results

Table 1 (on page 41) gives the space and time used by each partitioning of each example. Times are given in hours, minutes, and seconds, memory consumption is given in Kilobytes. The "Min/Max module size" column gives the minimal and maximal sizes of the files that keep the structures of single partition graph nodes. These are also measured in Kilobytes. The examples were run on a machine with 400M RAM.

The main goal of our experiments was to examine the effect of partitioning on the performance of model checking. However, we also checked the effect of local reachability and dynamic reordering. The "reo" option in the table marks whether dynamic variable reordering was used. This is an option provided by the BDD package. When the BDD size goes over a certain threshold a variable reordering algorithm is performed. This algorithm attempts to find a different ordering of the BDD variables, so that the BDDs will be smaller. This option has proved to be very useful for symbolic model checking. The BDD library we used enabled us to use different variable orderings for different partition-graph nodes (which reside in separate files). The "lr" option in the table marks the use of local reachability, as explained above.

All three examples were verified with 8-bit integers. In the first two examples integers are used mainly as pointers into buffers or array indices, but in the Sort example the array which is sorted is an array of integers. For this reason, even though it is the shortest example (least number of program counter locations) it is not the smallest in terms of the size of its state-space. This example could not complete in the available 400M when run without variable reordering, so there are no entries in the table for this example without the reordering option.

To make the results more readable, we summarized them in two graphs. Figure 5(a) shows the effect of partitioning on memory consumption and figure 5(b) shows the effect of partitioning on running times. Each graph contains three sub-graphs, one for each example.

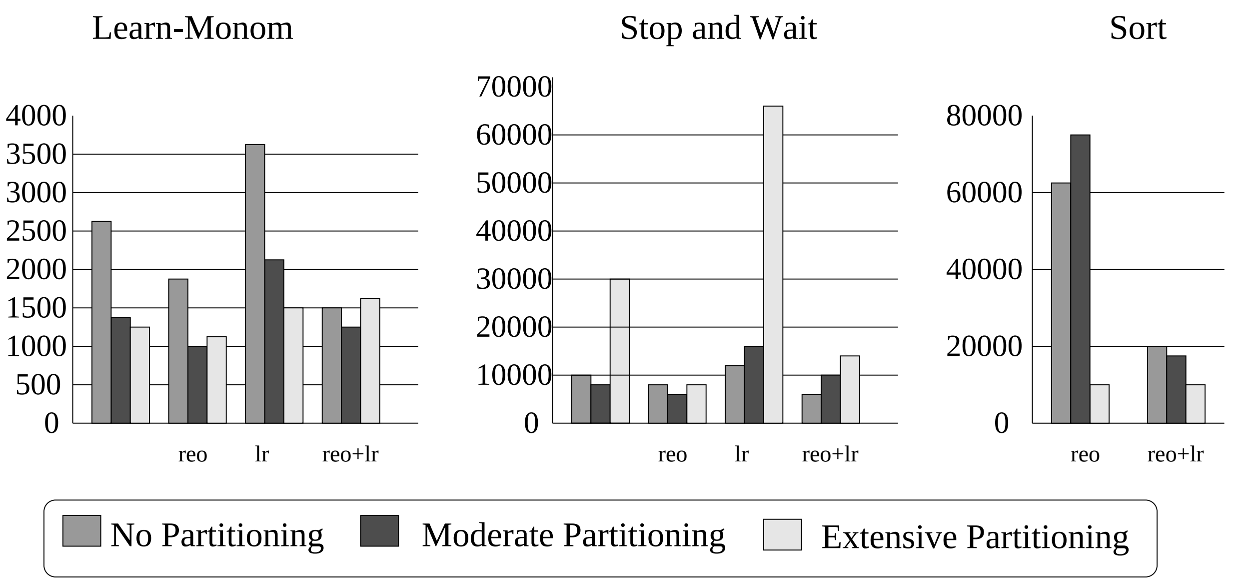
From figure 5(a) we can see that partitioning a program can in fact reduce space consumption considerably. The sort example is the most notable since it is a type of application which usually does not work well with BDDs. However in some cases partitioning the program can also enlarge the amount of space required. The moderate partitioning gives good results in 7 out of 10 cases, whereas the extensive partitioning gives good results only in 5 cases (out of which only in 4 cases the extensive partitioning gives better results than the moderate partitioning). On the other hand, the best result overall was achieved by the extensive partitioning in the Sort example. These results support our claim that the choice of partitioning is crucial. More research needs to be done in order to classify the types of programs in which partitioning is expected to improve performance, and the way in which such programs should be partitioned. When choosing a partition one must balance the overhead of partitioning with the reduction in the sizes of the modules.

By comparing figures 5(a) and 5(b) we see that there is no direct link between the space and time consumption. There are 2 cases in which although the extensive partitioning leads to a greater space consumption than the moderate partitioning, the extensive partitioning requires less run-time than the moderate one, and one case where the opposite happens. This can be explained by the fact that in the extensive partitioning there is one module for which the model checking at one point exploded and required more space, but since all other modules were small the model checking for all the other modules took much less time. The space consumption listed in the table is the *maximum* space used by the process, but in most cases this amount is used only for a short period of time.

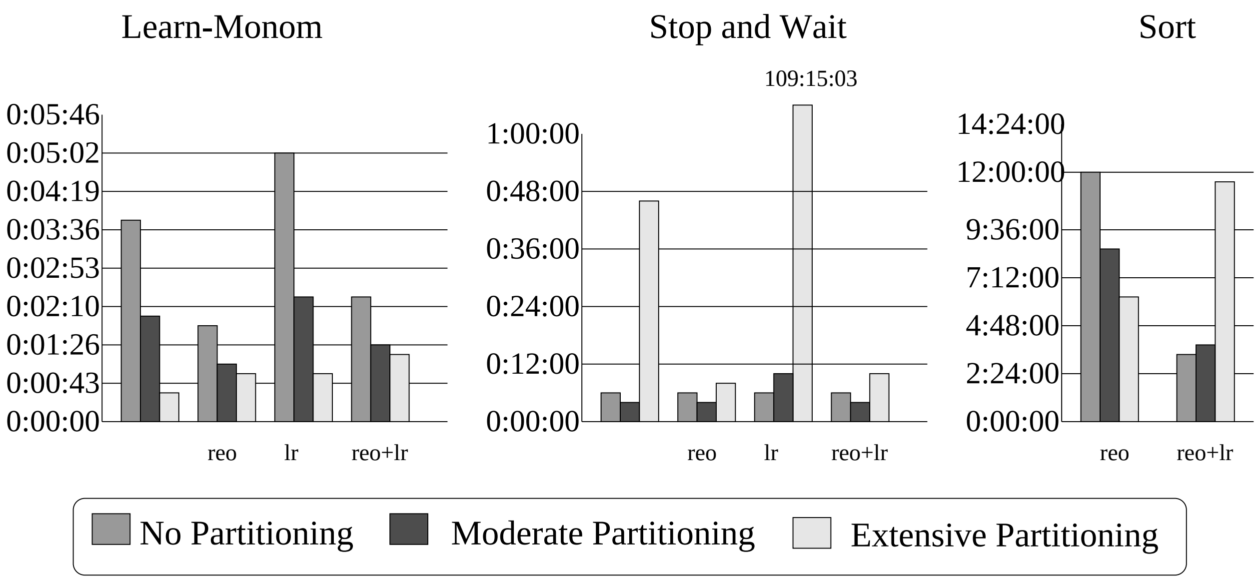
We notice the ill effect of local reachability. This is because local reachability introduces dependencies between variables, which usually enlarges the BDD sizes, but it does not reduce the state space sufficiently to compensate. We do not know whether this phenomenon will occur with larger examples, or different types of programs. We also notice the favorable effect of dynamic variable reordering. This supports the experience of others that dynamic variable reordering is helpful for model checking in most cases.

A question that should be asked is why the use of external memory is efficient, even though it is notoriously slow. The answer lies in the way we use the external memory. We do not use the external memory while performing model checking on a single structure, and we do not keep any kind of hash table in an external memory. Usually, use of external memory creates problems when it is used continuously and in small amounts. The main task of our algorithm, and the one that takes up most of the computation time, is the computation of model checking on a given structure. We use the external memory only when moving between nodes, and when we do - it is in one action - reading the BDD of the next node to be examined. This maintains a type of "locality", where we either work only on internal memory (while model checking) or perform a large continuous read or write operation.

To summarize our conclusions from the examples, we believe that the use of our



(a) Graph of memory consumption



(b) Graph of time consumption

Figure 5: Result graphs for Modular Model Checking

modular model checking algorithm can be instrumental in verifying some programs, but not in every case. We would advise the user to use it when it seems that regular model checking is too expensive, and to choose the partitioning so that the different tasks that the program performs reside in different partition graph nodes.

7 High-level programming language features

So far we have considered only very simple programming languages. We now discuss the possibilities for extending our approach to handle additional features of high-level programming languages.

7.1 Data types

There is no real restriction on the types allowed by a language to be used with our approach. The only limitations are those that come from the fact that we are, after all, performing model checking. This means that we need to create a finite Kripke structure from each program fragment. Obviously, unbounded integers cannot be handled. Extremely large integers will probably be impractical. Any other data type can be used - boolean, enumerated types, structures, multi-dimensional arrays, etc.

We do not allow for objects. The control flow in object oriented languages is different than the simple model of imperative languages. It may be possible to extend the ideas we present to the object oriented domain, but this is a subject for a different paper.

7.2 Non-deterministic choice

When presenting the structures of programs in Section 3.1 we noted that the only source of non-determinism is in simple commands (inputs). All of our control structures are deterministic. However, this limitation is not really necessary.

We can add to our control structures a non deterministic choice structure:

choose {
    $B_1 \rightarrow Prog_1$;
        $\vdots$
    $B_n \rightarrow Prog_n$;
    Default $\rightarrow Prog_{n+1}$
}

The semantics of this statement is a non-deterministic choice between the boolean conditions that evaluate to $true$, and the execution of the corresponding sub-program. If all of the conditions evaluate to $false$ then $Prog_{n+1}$ is executed.

The "choose" statement is an extension of the "if", and the modular algorithm can handle it in a similar manner. The partition graph for a "choice" will include a node for the head of the "choice" statement with edges to the partition graphs of the programs $Prog_1, \ldots, Prog_{n+1}$. Instead of having just true-edges and false-edges, we would now have edges labelled with the conditions $B_1, \ldots, B_n$ and an edge $\bigwedge_{i=1\ldots n} \neg B_i$ for the default. The structures associated with these edges would be defined accordingly. In the Kripke structure for a program, the states with the location of a "choose" statement may have several successors. In the partition graph, these states are in the structure of the head node, and each successor resides in a different sub-program, and hence in a different partition graph node.

Given an assumption function As we can use a recursive call on the partition graphs of the programs $Prog_1, \ldots, Prog_{n+1}$ to create assumption functions $As_1, \ldots, As_{n+1}$. We can then collect the information from all these assumptions into one assumption function, and deduce an assumption function As' over the initial states (the location of the "choose" statement) by using MC on the combined structure of the head node and the step-edges.

7.3 Repeat-Until loops

The loop structure we use is the type in which the condition of the loop is evaluated at the top. It is of course possible to have loops in which the condition is evaluated at the end. This means that the body of the loop is executed at least once. The naive way to adapt our algorithm to such loops is to apply a simple transformation to the text of the program. Every "repeat $Prog_1$ until B" can be replaced by "$Prog_1$; while B do $Prog_1$ od". This does not change the complexity of our algorithm since we duplicate the body of the loop exactly twice.

We do not elaborate any further on this subject. However, we will note that it is possible to avoid this duplication and create an algorithm that specifically deals with "repeat-until" loops. It will be very similar to the "while" algorithm.

7.4 Goto

The "goto" command breaks the structure of the control flow in a program. It allows changing the program counter at any point in the program, and jumping to any other point. Our algorithm is based on the structure of the control-flow. For example, for a program $P_1; P_2$ we use the fact that from all locations in P_2 the control cannot go back to P_1 (without passing through the head of a containing loop). This assumption is what allows us to first model-check P_2 and then use the results on P_1. If we allow the use of goto commands our algorithm is no longer sound.

We could allow a restricted form of "goto" if we limit the possible partitionings of a program so that each goto changes the program counter to a location which resides in the same partition graph node. In this situation the presence of the "goto" does not influence the correctness of our modular algorithm.

7.5 Dynamic memory allocation

Naturally, we cannot handle a language that allows arbitrary dynamic memory allocation, since this creates programs with un-bounded state-space. As noted before, we must be able to create a finite Kripke structure that represents the set of all possible computations of the program.

However, we can take advantage of any known solution for this problem. Similarly to Bandera [12], PathFinder [4], and Demartini et al [22], a pre-processing can apply several techniques: static analysis to compute in advance the state space needed, abstraction to create an abstract finite representation of an infinite data space, or slicing to reduce the state-space. Once the program is translated into a finite structure, we can use our algorithm to boost the performance of CTL model checking.

7.6 Procedures and Recursion

It is obvious that any "real" programming language must allow the use of procedures. The existence of procedures without recursion does not present a problem in using our algorithm, since it is always possible to replace each procedure call with the procedure body, after replacing the formal parameters with the actual ones (this is called *inlining* in the Bandera tool [13]). However, this method does not utilize the procedural structure to gain efficiency of the model checking algorithm.

Recursion adds another layer of complexity, since it creates programs with unbounded state space. For this reason, we cannot handle languages that allow recursion. A similar restriction is taken also in the PathFinder tool [19].

7.7 Parallel composition and multi-threading

The algorithm as it is presented here is suitable for single sequential processes. It is based on the assumption that when visiting a partition graph node, we have already visited all of its successor nodes, and have accurate (although, perhaps, partial) information on the computations after this node. When several programs are run in parallel we need to take into account the points in which they communicate. If they use shared memory, then we must assume that the shared variables value can change at any given time. If they use message passing to communicate, then we must handle the communication commands. It seems possible to extend our ideas into an algorithm that will work for parallel composition and multi-threading, while taking advantage of the fact that each process is a sequential program. We leave this as a topic for future research.

A limited solution can be obtained by translating interleaving processes into one big non-deterministic loop that chooses at each step the process to execute and the action to be performed by this process (such a solution is taken by the Bandera tool [18]). However, this solution defeats the purpose of exploiting the modular structure of the program.

8 Conclusions

8.1 Related Work

A significant amount of research has been done lately in the area of software verification. Several works are targeted towards applying model checking to programs. The SPIN system [20] is an example of a model checking tool for high-level programs with finite-state models. It uses a language called PROMELA and is mainly used to verify communication protocols.

In this category we also find tools such as Bandera [13] and PathFinder [4], and the work of Demartini et al [22], which translate Java programs into finite-state models. In fact, they translate their programs into SPIN and count on its construction of finite models. Bandera also translates its programs to SMV, PathFinder and other tools.

To make model checking applicable, these tools restrict themselves to language constructs that produce finite-state models. In particular, they do not allow unbounded memory allocation and recursive procedure calls. They apply abstraction, slicing, and static analysis in order to bound and reduce the model size. A similar approach is taken by the AX tool of Holzmann [21], which translates distributed C programs to SPIN, based on user-defined abstraction.

As mentioned in Section 7, our modular model checking can be applied to many of the language constructs handled by these tools. Introducing our method into their framework should further enhance their capabilities.

Works such as Bebop [1] and Yahav et al [33] can handle interprocedural calls. They are both based on dataflow analysis developed by Reps, Horwitsz and Sagiv [32]. Bebop is a part of the SLAM [2] project in which sequential C programs are verified. Predicate abstraction and abstraction refinement are used in order to obtain boolean programs in which boolean variables represent predicates. The boolean programs are then verified using the Bebop tool.

Yahav et al [33] can verify programs that allow, in addition to interprocedural calls, also unbounded dynamic allocation of objects, including threads, and reference to objects. The programs are abstracted using 3-value logic and are verified with respect to LTL properties.

Another tool that can handle dynamic allocation is dSPIN [14] which extends SPIN with features such as dynamic object creation and deletion, multiple threads, and pointers. Programs in dSPIN are translated into SPIN. However, each state is represented by a number of vectors, allowing for dynamic enlarging and shrinking of the state representation.

Our method is based on the fact that each subprogram can be translated into a finite-state model and therefore it is not straightforward to extend it to programs with language constructs that are unbounded in nature.

The VeriSoft tool of Godfroid [16] presents a different approach for verifying C or C++ programs. The tool searches the state-space of a concurrent program by repetitively running each process until it reaches its next communication statement.

This method is not modular in nature, and it is not clear whether it can make use of our modular model checking algorithm.

Works that use process algebras to represent a program can be considered as handling software, although not written in a high-level language. One of the most relevant works, which bears some resemblance to our modular model checking algorithm, is the work of Burkart and Steffen [7, 8]. They present model checking algorithms for context-free processes, and for a generalization of context-free processes called pushdown processes. They consider the semantics of 'fragments', which are interpreted as 'incomplete portions' of the process. The model checking algorithms they propose are based on the calculation of the *property transformer* of each fragment, which is a function that represents the semantics of a fragment with respect to alternation-free μ-calculus formulas. A property transformer receives a set of μ-calculus formulas which are true at the exit point of a fragment, and returns the set of μ-calculus formulas true at the entry to the fragment. After calculating the property transformers of all fragments, the property transformer of the main fragment is given the set of μ-calculus properties that hold when a process halts, and the result computed by the property transformer is the result of the model-checking algorithm. The main draw-back of this algorithm, in our opinion, is that it is defined for Pushdown Processes Systems, which can hardly be considered as a high level programming language, and μ-calculus properties, which are not easy to use. This makes it an interesting theoretical result, but not useful for use by real designers of hardware or software systems. It should also be noted that the property transformers of all the fragments are computed together, one depending on the other, and it is not clear whether using secondary memory to store partial results would prove useful, or even possible.

8.2 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, in principle, 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 a next-state function). The latter is based on a BDD representation [5] of the system model. Each has its advantage for certain areas of applications and each can be made modular using our approach.

Theoretically, our algorithm can work with explicit-state model checkers that work on-the-fly. Even though our modular algorithm works in reverse on sequentially composed subprograms, the model checking of each node in isolation can be done using any method. In practice, however, our approach is most efficient when used in the context of symbolic model checking. The problem with using on-the-fly methods is not theoretical, but computational. Since we do not have exact reachability information, for every node we would need to apply the on-the-fly model checking algorithm starting with every possible initial state. This would prove extremely inefficient. In this case, local reachability might be of some help since it somewhat reduces the set of traversed states.

An important notion used in our 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. However, it should be useful to develop heuristics that will allow automatic creation of the partition graph.

References

[1] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, pages 103–122, 2001.

[3] 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.

[4] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *In IEEE International Conference on Automated Software Engineering (ASE)*, September 2000.

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

[6] 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.

[7] Olaf Burkart and Bernhard Steffen. Model checking for context-free processes. LNCS 630, pages 123–137. Springer, 1992.

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

[9] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, December 1999.

[10] 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.

[11] 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.

[12] J. Corbett, M. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The bandera specification language. Technical report, KSU CIS Technical Report 2001-04, June 2001. This paper is an extended version of the 2000 Spin Workshop paper.

[13] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[14] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *SPIN*, pages 261–276, 1999.

[15] 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.

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

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

[18] J. Hatcliff and M. Dwyer. Using the bandera tool set to model-check properties of concurrent java software. In *International Conference on Concurrency Theory (CONCUR)*, June 2001. Invited tutorial paper.

[19] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[20] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International Editors, 1991.

[21] G. Holzmann. Logic verification of ansi-c code with SPIN. In *Proceedings of the 7th international SPIN workshop*, volume 1885 of *LNCS*, pages 131–147, 2000.

[22] R. Iosif, C. Demartini, and R. Sisto. Modeling and validation of java multithreading applications using SPIN. In *In Proceedings of the 4th SPIN workshop*, Paris, France, November 1998.

[23] 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.

[24] 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.

[25] 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.

[26] D. E. Long. A BDD library, carnegie-mellon university. available at: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/modck/pub/www/bdd.html.

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

[28] 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.

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

[30] 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.

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

[32] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.

[33] Eran Yahav, Thomas Reps, and Mooly Sagiv. Ltl model checking for systems with unbounded number of dynamically created threads and objects.

A Proof of Main Theorem

This subsection is devoted to proving theorem 5.1 which stated that:
The modular model checking algorithm computes model checking under assumptions correctly for any partition graph G of any program P. Formally, for any assumption function As over $end(G)$: $CheckGraph(G, As) = MC[struct(G), As]$.

The proof of this theorem requires several stages. We start with the following lemma.

Lemma A.1: Let $M = \langle S, R, I \rangle$ be the structure of some program P. Let $M' = \langle S', R', I' \rangle$ be the structure of a sub-program P' of P, or of an edge e in the control-flow graph of P. Let As be an arbitrary assumption function over $end(M)$ and As' an assumption function over $end(M')$ such that for every $s \in end(M')$ and $\varphi \in cl(\psi)$ it holds that $s \in As'(\varphi) \Leftrightarrow M, s \models_{As} \varphi$ [9]. Then for every $s \in M'$ and every $\varphi \in cl(\psi)$: $M', s \models_{As'} \varphi \Leftrightarrow M, s \models_{As} \varphi$.

Proof: The proof is by induction on the top-most operator of φ. For the base case, $\varphi \in AP$, and the induction steps that do not involve temporal operators the proof is trivial since the definition of $M, s \models_{As'} \varphi$ does not depend on the assumption function As' (only on the induction hypothesis for sub-formulas of φ). The interesting cases are the ones that involve temporal operators. We show only the proofs for $\mathbf{EX}\varphi_1$ and $\mathbf{E}(\varphi_1\mathbf{U}\varphi_2)$, the proofs for $\mathbf{AX}\varphi_1$ and $\mathbf{A}(\varphi_1\mathbf{U}\varphi_2)$ are dual. The induction hypothesis is that for φ_1 and φ_2 the claim holds, i.e. for every $s \in S'$: $M', s \models_{As'} \varphi_i \Leftrightarrow M, s \models_{As} \varphi_i$ for $i = 1, 2$.

- Let $\varphi = \mathbf{EX}\varphi_1$. For every state $s \in S'$ we have:
  $M', s \models_{As'} \varphi \Leftrightarrow$
  $\exists s' \in S'.R'(s, s') \wedge M', s' \models_{As'} \varphi_1$ or $s \in end(M') \cap As'(\varphi)$
  $\Leftrightarrow$ (since $S' \subseteq S$, $R' \subseteq R$, and since $R(s, s') \wedge s \notin end(M')$ implies $s' \in S'$)
  $\exists s' \in S.R(s, s') \wedge M, s' \models_{As} \varphi_1$ or $s \in end(M') \cap As'(\varphi)$
  $\Leftrightarrow$ (from our assumption about $As'(\varphi)$)
  $M, s \models_{As} \varphi$.

- Let $\varphi = \mathbf{E}(\varphi_1\mathbf{U}\varphi_2)$. We first prove that for every state $s \in S'$: $M', s \models_{As'} \varphi \Rightarrow M, s \models_{As} \varphi$:
  $M', s \models_{As'} \varphi$
  $\Rightarrow$ (by definition)
  There exists a path $\pi = s_0, \ldots, s_i$ in M' ($s = s_0$) such that either $s_i \in end(M') \cap As'(\varphi)$ or $M', s_i \models_{As'} \varphi_2$, and for every $j < i$: $M', s_j \models_{As'} \varphi_1$
  $\Rightarrow$ (from the induction hypothesis about φ_1 and φ_2 and what we know of As')
  There exists a path $\pi = s_0, \ldots, s_i$ in M ($s = s_0$) such that for every s_i, either $M, s_i \models_{As} \varphi$ or $M, s_i \models_{As} \varphi_2$, and for every $j < i$: $M, s_j \models_{As} \varphi_1$
  $\Rightarrow$ (by definition)
  $M, s \models_{As} \mathbf{E}(\varphi_1\mathbf{U}\varphi_2 \vee \mathbf{E}(\varphi_1\mathbf{U}\varphi_2))$
  $\Rightarrow M, s \models_{As} \mathbf{E}(\varphi_1\mathbf{U}\varphi_2)$.

  Next, we prove that for every state $s \in S'$, $M, s \models_{As} \varphi \Rightarrow M', s \models_{As'} \varphi$. If $M, s \models_{As} \mathbf{E}(\varphi_1\mathbf{U}\varphi_2)$ then there exists a path $\pi = s_0, \ldots, s_i$ in M ($s = s_0$) such that $M, s_i \models \varphi_2$ and for every $j < i$: $M, s_j \models_{As} \varphi_1$. Here there are two possibilities:

  1. For every $k \leq i$, $s_k \in S' \setminus end(M')$. In this case, from the induction hypothesis about φ_1 and φ_2 we have:
     $M', s_i \models_{As'} \varphi_2$ and for every $j < i$: $M', s_j \models_{As'} \varphi_1$
     $\Rightarrow M', s \models_{As'} \varphi$

[9] The theorem will also hold if As is over a set that includes $end(M)$ instead of being defined exactly over $end(M)$ (and the same for As' and $end(M')$.

2. There exists a state from $end(M')$ along π. Let s_k be the first such state (so for every $j < k$ we have $s_j \in S' \setminus end(M')$). Then we have:
$M, s_k \models_{As} \varphi$ and for every $j < k$: $M, s_j \models_{As} \varphi_1$
$\Rightarrow$ (from our assumption about As' and the induction hypothesis)
$s_k \in As'(\varphi)$ and for every $j < k$ $M', s_j \models_{As'} \varphi_1$
$\Rightarrow$ (by definition)
$M', s \models_{As'} \varphi$

$\square$

The proof of theorem 5.1 is by induction on the size of the partition graph G. The most complicated part to prove is the operation of the algorithm on a while loop. This part is proven by induction on the sub-formulas φ_k. The most complicate part of this inner induction is, naturally, the proof for $\varphi_k = \mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$ or $\varphi_k = \mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$. In the following we prove several lemmas about this part, and then use them in the proof for the whole algorithm. These lemmas are going to be used in the inner-most induction, so the induction hypotheses may be needed as premises.

The first premise is the induction hypothesis of the outermost induction on the size of the partition graph.

Premise 1:
Every recursive call of CheckGraph on the partition graph of the body of the while (G_1) with an (arbitrary) assumption function As^{in} over $end(G_1)$, results in an assumption function As^{out} over $init(G_1)$ such that, for every $\varphi \in cl(\psi)$ s.t. $As^{in}(\varphi) \neq \perp$ and every state $s \in init(G_1)$ we have $s \in As^{out}(\varphi) \Leftrightarrow struct(G_1), s \models_{As^{in}} \varphi$.

The second premise is the induction hypothesis for the induction on the sub-formulas φ_k.

Premise 2:
After running the algorithm for the formulas $\varphi_1, \ldots, \varphi_{k-1}$ the algorithm built the sets $As'(\varphi_1), \ldots, As'(\varphi_{k-1})$ correctly, i.e. for every $s \in init(G)$ and $j < k$ we have $s \in As'(\varphi_j) \Leftrightarrow struct(G), s \models_{As} \varphi_j$.

To reason about the algorithm for the while loop we define the following three structures. The structure of the partition graph G of the whole while-loop is $M_G = \langle S_G, R_G, I_G \rangle$. The structure of the partition graph G_1 of the body of the while loop is $M_1 = \langle S_1, R_1, I_1 \rangle$. We define a third structure $M = \langle S, R, I \rangle$ which is the structure M_G without the transitions of the false edge $e_{\neg B} = n_B \overset{false}{\longrightarrow} n_D$. This means that $S = S_G \setminus end(G)$ ($end(G)$ is the set of states that have the program location following the while loop, the states you end up at when the loop is done), and $R = R_G \setminus R_{e_{\neg B}}$. The following observations are at the base of our proof:

- $end(M_1) = init(M_G) = init(M)$
  The set of ending states of the body of the loop is the set of initial state of the whole loop. These are the states in which the condition of the while is evaluated. Their successors are either in $init(M_1)$, if the condition is true, or in $end(M_G)$, if it is false.

- $end(M) \subseteq init(M)$
  The ending states of M are the states in which the condition B is evaluated and found not to hold. The transitions outgoing from these states are included in M_G, but not in M.

- For checking satisfaction under assumptions on M_G the assumption function must be defined over $end(M_G)$, which is the set of states with the location following the while loop. For checking satisfaction under assumptions on M the assumption function must be defined over $end(M)$. Any assumption function defined over $init(M)$ is also defined over $end(M)$ and can be used for satisfaction under assumption of states in M.

We start by connecting satisfaction of formulas in M_G under the assumption As with satisfaction in M under the assumption $As_{\neg B}$. Recall that As is over $end(M_G)$ and $As_{\neg B}$ is over $end(M)$.

Lemma A.2: For every $s \in init(G)$ and $\varphi \in cl(\psi)$: $M_G, s \models_{As} \varphi \Leftrightarrow M, s \models_{As_{\neg B}} \varphi$.

Proof: The assumption function $As_{\neg B}$ is created by a call to CheckStepEdge on the false edge, with As. Therefore, for every $s \in init(G)$ such that $s \not\models B$ and $\varphi \in cl(\psi)$ we have $s \in As_{\neg B}(\varphi) \Leftrightarrow M_G, s \models_{As} \varphi$. We have already noted that the set of states $s \in init(G)$ such that $s \not\models B$ is exactly $end(M)$, so we can use lemma A.1 to conclude that for every $s \in init(M)$: $M, s \models_{As_{\neg B}} \varphi \Leftrightarrow M_G, s \models_{As} \varphi$. □

From here on, we prove correctness with respect to M and $As_{\neg B}$, and this will imply correctness with respect to M_G and As.

The next lemma we prove makes two claims. One is that for every $s \in As^i(\varphi_k)$: $M, s \models_{As_{\neg B}} \varphi_k$. This means that every state that is marked as satisfying φ_k, actually does satisfy φ_k. The second claim is that the series of sets $As^i(\varphi_k)$ is a monotonically growing series of sets, i.e. each new set includes all the states of the previous set. This means that we do not "lose" states that have already been found to satisfy φ_k. Notice that when calculating the series of assumption functions As^i, the sets $As^i(\varphi_j)$ for $j < k$ are always the same, because they are copied from As^0. The goal of this part of the algorithm is only to compute the set $As'(\varphi_k)$. We also notice here that the assumption functions As^i are defined over $init(M)$, which includes $end(M)$. During the calculation of these functions we create the assumption function Tmp over $init(G_1)$ and $TmpB$ over the set of states in $init(M)$ that satisfy B.

Lemma A.3: Assume that premises 1 and 2 hold. Let As^i be one of the assumption functions calculated by the algorithm when working on a formula φ_k of the form $\mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$ or $\mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$ on a graph of a while loop. Then for every state $s \in As^i(\varphi_k)$, $M, s \models_{As_{\neg B}} \varphi_k$, and if $i > 0$ then $As^{i-1}(\varphi_k) \subseteq As^i(\varphi_k)$.

Proof: We prove both claims of the lemma together, by induction on i.

To show that $s \in As^0(\varphi_k)$ implies $M, s \models_{As_{\neg B}} \varphi_k$ we examine part (a) of this portion of the algorithm, where As^0 is created. If $s \in Init_B$ then s is an initial state that satisfies B and, according to the induction hypothesis for φ_m, also satisfies φ_m under the assumption $As_{\neg B}$. Therefore, $M, s \models_{As_{\neg B}} \varphi_k$. If, on the other hand,

$s \in As_{\neg B}(\varphi_k)$ then $s \in end(M)$ and by definition $M, s \models_{As_{\neg B}} \varphi_k$. To show that $As^0(\varphi_k) \subseteq As^1(\varphi_k)$ we move to examine part (b). We see that $As^1(\varphi_k) = TmpB(\varphi_k) \cup As_{\neg B}(\varphi_k)$. Let s be a state in $As^0(\varphi_k)$. If $s \in As_{\neg B}(\varphi_k)$ then automatically $s \in As^1(\varphi_k)$. Otherwise, $s \in Init_B$, which means that $s \models_{As} \varphi_m$, and from our induction hypothesis for φ_m (premise 2) we conclude that CheckStepEdge will create $TmpB$ correctly (states that satisfy φ_m also satisfy $\mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$ and $\mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$) and $s \in As^1(\varphi_k)$.

We now assume both claims hold for every $j < i$, and show them for As^i. Let $s \in init(M)$ be a state such that $s \in As^i(\varphi_k)$. We show that $M, s \models_{As_{\neg B}} \varphi_k$. If $s \in As_{\neg B}(\varphi_k)$ then as before $M, s \models_{As_{\neg B}} \varphi_k$. Assume that $s \in TmpB(\varphi_k)$. The state s has a single successor, which is a state $s' \in init(M_1)$. If CheckStepEdge inserted s into $TmpB(\varphi_k)$ then it must be that $s' \in Tmp(\varphi_k)$ (assuming that $s \not\models_{As_{\neg B}} \varphi_m$). From the induction hypothesis for the recursive call to CheckGraph we deduce that $M_1, s' \models_{As^{i-1}} \varphi_k$. If $\varphi_k = \mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$ then there must be a path $\pi = s_0, \ldots, s_n$ ($s' = s_0$) in M_1 such that either $s_n \in end(M_1) \cap As^{i-1}(\varphi_k)$ or $M_1, s_n \models_{As^{i-1}} \varphi_m$, and for every $j < n$: $M, s_j \models_{As^{i-1}} \varphi_l$. From premises 1 and 2 together we deduce that for every $j < n$: $M, s_j \models_{As_{\neg B}} \varphi_l$. If $s_n \in end(M_1) \cap As^{i-1}(\varphi_k)$ then $M, s_n \models_{As_{\neg B}} \varphi_k$, and if $M_1, s_n \models_{As^{i-1}} \varphi_m$ again we have $M, s_n \models_{As_{\neg B}} \varphi_k$. In both cases we conclude that $M, s' \models_{As_{\neg B}} \varphi_k$ which implies $M, s \models_{As_{\neg B}} \varphi_k$. The proof for $\mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$ is similar, except that instead of reasoning about one path π from s' we talk about all the paths from s'.

We are left with showing that $As^{i-1}(\varphi_k) \subseteq As^i(\varphi_k)$. Let s be a state in $As^{i-1}(\varphi_k)$. The set $As^{i-1}(\varphi_k)$ is created at the end of the "Until" loop, as the union of $TmpB(\varphi_k)$ and $As_{\neg B}(\varphi_k)$.

If $s \in As_{\neg B}(\varphi_k)$ then it will stay there (since $As_{\neg B}(\varphi_k)$ does not change) and so we are guaranteed that $s \in As^i(\varphi_k)$. If $s \in TmpB(\varphi_k)$ then we show that it will also be in $TmpB(\varphi_k)$ when $As^i(\varphi_k)$ is created. To prevent confusion, we use $TmpB^j$ and Tmp^j for the functions $TmpB$ and Tmp created in the jth iteration of the loop, which is the iteration in which $i = j$ and As^j is created.

From the induction hypothesis on the recursive call to CheckGraph, the correctness of CheckStepEdge, and lemma A.1, we deduce that for every state $t \in init(M)$, and every $j \leq i$, $t \in TmpB^j(\varphi_k) \Leftrightarrow M, t \models_{As^{j-1}} \varphi_k$. Obviously, if $s \in As^{i-1}$ because $M, s \models_{As_{\neg B}} \varphi_m$ then $s \in TmpB^i(\varphi_k)$ and so $s \in As^i(\varphi_k)$. Otherwise, if $s \in As^{i-1}(\varphi_k)$ it is because $M, s \models_{As^{i-2}} \varphi_k$. We know that for every $j < k$ we have $As^{i-2}(\varphi_j) = As^{i-1}(\varphi_j)$, and that $As^{i-2}(\varphi_k) \subseteq As^{i-1}(\varphi_k)$, so $M, s \models_{As^{i-2}} \varphi_k$ implies $M, s \models_{As^{i-1}} \varphi_k$. This last step is true because of the monotonicity of the **EU** and **AU** operators. In the definition of satisfaction under assumptions we see that if the sets of states assumed to satisfy $\mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$ (or $\mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$) grow larger, then the set of states that satisfy $\mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$ (or $\mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$) under this assumption cannot grow smaller. So now we know that $M, s \models_{As^{i-1}}$, and this implies $s \in TmpB(\varphi_k)$, i.e. $s \in As^i(\varphi_k)$. $\square$

Lemma A.4: Assume that premises 1 and 2 hold. For every state $s \in init(M)$ such that $M, s \models_{As_{\neg B}} \varphi_k$ ($\varphi_k = \mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$ or $\varphi_k = \mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$) there exists a number i such that $s \in As^i(\varphi_k)$.

Proof: We prove lemma A.4 for the case of $\varphi_k = \mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$, the proof for $\varphi_k = \mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$ follows a similar line of reasoning.

Let s be a state in $init(M)$ such that $M, s \models_{As_{\neg B}} \varphi_k$. Let $\pi = s_0, \ldots, s_n$ $(s = s_0)$ be the shortest path from s that proves this, i.e. it is the shortest path such that either $s_n \in end(M) \cap As_{\neg B}(\varphi_k)$ or $M, s_n \models_{As_{\neg B}} \varphi_m$, and for every $j < n$: $M, s_j \models_{As} \varphi_l$. We define n to be the *depth* of s.

Assume to the contrary that there are one or more states that satisfy φ_k but will not get into any set $As^i(\varphi_k)$. Obviously, these must be states that satisfy B. Let s be the state with minimal depth, and n its depth. Let $\pi = s_0, \ldots, s_n$ be the path that shows this. Then there are two possibilities:

1. The state s_0 is the only state from $init(M)$ along π. In this case we know that the states $s_1, \ldots, s_n$ are all inside M_1. From the premises of the lemma we know that when the first recursive call to CheckGraph on G_1 is done we have $s_1 \in Tmp(\varphi_k)$ because $M_1, s_1 \models_{As^0} \varphi_k$ can be proven on M_1 just by examining the sets of states that satisfy φ_l and φ_m in M'. Therefore we must conclude that CheckStepEdge will create $TmpB$ so that $s_0 \in TmpB(\varphi_k)$ which implies $s_0 \in As^1(\varphi_k)$ in contradiction to the assumption that $s = s_0$ does not get into any set $As^i(\varphi_k)$.

2. There exists a state other than s_0 along π which is in $init(M)$ (the path π goes more than once through the loop). Let s_j $(j > 0)$ be the first such state along π. Since π was chosen as the shortest path that proves $M, s_0 \models_{As_{\neg B}} \varphi_k$, we know that $s_j, \ldots, s_n$ is the shortest path that proves $M, s_j \models_{As_{\neg B}} \varphi_k$, and that the depth of s_j is strictly smaller than the depth of s_0. From the way we chose s we conclude that there is a function As^i such that $s_j \in As^i(\varphi_k)$. This means that $M, s_0 \models_{As^i} \varphi_k$. For similar reasoning as we have done before, using the two premises, we conclude that $s_0 \in As^{i+1}(\varphi_k)$ in contradiction to our assumption that $s = s_0$ never gets into any set $As^i(\varphi_k)$.

$\square$

We can now finally prove theorem 5.1.

Proof: As mentioned before, the proof of the algorithm is by induction on the size of the partition graph. The base case is the application of the given model checking under assumptions procedure on the structure of a single partition graph node. For this case we know that the given procedure is correct. The induction step is according to the topmost structure of the partition graph G. In all the cases below we use M_G for the structure of P and M_i for the structure of P_i $(i = 1, 2)$.

- For a graph of $P = P_1; P_2$ as in figure 2(A):
  From the induction hypothesis for the smaller graph G_2 we conclude that for

every state $t \in init(G_2)$ and every formula $\varphi \in cl(\psi)$ such that $As(\varphi) \neq \bot$, $t \in As_1(\varphi) \Leftrightarrow M_2, t \models_{As} \varphi$. Similar reasoning shows that for every $s \in init(G_1)$, $s \in As'(\varphi) \Leftrightarrow M_1, s \models_{As_1} \varphi$. Using lemma A.1, and since $init(G_1) = init(G)$ we have: for every state $s \in init(G)$ and every formula $\varphi \in cl(\psi)$ such that $As(\varphi) \neq \bot$, $s \in As'(\varphi) \Leftrightarrow M_G, s \models_{As} \varphi$.

- For a graph of $P =$ "if B then P_1 else P_2 fi" as in figure 2(B):
  We use the induction hypothesis for the recursive calls on G_1 and G_2 to conclude that for every $t \in init(G_i)$ and φ such that $As(\varphi) \neq \bot$, $t \in As_i(\varphi) \Leftrightarrow M_i, t \models_{As} \varphi$ $(i = 1, 2)$. From the correctness of CheckStepEdge, and using lemma A.1, we conclude that for every $s \in init(G)$ and every φ such that $As(\varphi) \neq \bot$, $s \in As'(\varphi) \Leftrightarrow M_G, s \models_{As} \varphi$.

- For a graph of $P =$ "while B do P_1 od" as in figure 2(C):
  For this part we use induction on k.

  For $\varphi_k \in AP$, $\varphi_k = \neg\varphi_l$ and $\varphi_k = \varphi_l \vee \varphi_m$ the proof is trivial, since we assume correctness for φ_l and φ_m (the induction hypothesis).

  For $\varphi_k = \mathbf{AX}\varphi_l$ or $\varphi_k = \mathbf{EX}\varphi_l$, we use the induction hypothesis for the recursive call on G_1 to conclude that for every $t \in init(G_1)$, $t \in Tmp(\varphi_l) \Leftrightarrow M_{G_1}, t \models_{As} \varphi_l$. For every state $s \in init(G)$ such that $s \not\models B$, we know that $s \in As_{\neg B}(\varphi_k) \Leftrightarrow s \models_{As} \varphi_k$ from lemma A.2. For every state $s \in init(G)$ such that $s \models B$ we know that s has only one successor, so $M_G, s \models \varphi_k$ iff its successor state t satisfies $t \in Tmp(\varphi_l)$. Therefore, for every state $s \in init(G)$ we have $s \in As'(\varphi_k) \Leftrightarrow M_G, s \models_{As} \varphi_k$.

  For $\varphi_k = \mathbf{A}(\varphi_l \mathbf{U} \varphi_m)$ or $\varphi_k = \mathbf{E}(\varphi_l \mathbf{U} \varphi_m)$ we have already proven, using both lemma A.2 and lemma A.3, that for every $s \in init(G)$, $s \in As'(\varphi_k) \Rightarrow M_G, s \models_{As} \varphi_k$. This proves that every state we mark actually satisfies the formula. We have also shown in lemma A.3 that with every iteration the set $As^i(\varphi_k)$ can only grow larger, and since there are finitely many states in $init(G)$ the process is guaranteed to stop. In lemma A.4 we have shown that any state which satisfies φ_k will eventually get into one of the sets $As^i(\varphi_k)$ and so we have $M_G, s \models_{As} \varphi_k \Rightarrow s \in As'(\varphi_k)$, which concludes the proof for the while loop.

$\square$

Example	Break-down	Options lr	Options reo	Time (h:m:s)	Memory (K)	Min/Max module size
Learn-Monom	None	-	-	0:03:46	2600	1450/1450
		-	+	0:01:56	1864	15058/15058
		+	-	0:05:00	3625	20860/20860
		+	+	0:02:11	1420	4787/8168
	Moderate	-	-	0:01:49	1352	3024/10661
		-	+	0:01:06	968	3150/10968
		+	-	0:02:25	2025	4495/15502
		+	+	0:01:22	1160	4648/15788
	Extensive	-	-	0:00:36	1192	2911/5357
		-	+	0:50:00	1128	3016/5450
		+	-	0:00:49	1450	4495/5165
		+	+	0:01:18	1545	4787/8168
Stop and Wait	None	-	-	6:46:18	478787	83440/83440
		-	+	7:04:06	6560	36145/36145
		+	-	6:21:51	10975	198592/198592
		+	+	6:35:50	4475	75249/75249
	Moderate	-	-	5:51:22	8130	4908/45129
		-	+	5:58:28	6050	4986/22226
		+	-	10:17:00	13650	4078/120447
		+	+	4:09:43	9225	4727/37508
	Extensive	-	-	47:12:55	29200	695/19890
		-	+	8:40:44	7800	695/8430
		+	-	109:15:03	65000	695/52559
		+	+	8:24:52	12050	695/13023
Sort	None	-	+	12:05:04	61500	109285/109285
		+	+	3:33:31	19100	205829/205829
	Moderate	-	+	8:32:54	73500	839/103324
		+	+	4:13:22	18300	875/205197
	Extensive	-	+	6:24:38	8150	842/49058
		+	+	11:45:27	9500	830/19718

Table 1: Results for Modular Model Checking