Syntactic Model Reduction

H. Peng, Y. Mokhtari, S. Tahar

April 2001

Abstract

Model checking is a fully automatic approach to verify a finite state machine against its temporal specifications. However, its application is limited by the size of the system to be verified. Model reduction is one of the methods to tackle this problem. In this paper, we proposed a model reduction approach which is based on syntactic analysis of programs. The value domains of the state variables are extracted and analyzed based on the control flow diagram. The approach is sound based on the fact that the reduced system simulates the concrete system with respect to the properties. Throughout the analysis, we can find that the proposed approach is efficient on data-path heavy systems. We also made a simple case study in the paper to show its performance.

Contents

1 Introduction

Model checking [5] is a fully automatic approach to verify a finite state machine against its temporal specifications. However, its application is limited by the size of the system to be verified. Current model checking tools [2, 18, 3] are limited to several hundred boolean state variables due to state space explosion. There are two main methods to tackle this problem: *compositional verification* and *model reduction*. Compositional verification decomposes the system under verification such that we can verify the partitions separately. However, in today's multi-million-gate VLSI designs, the size of one single module usually is beyond the capability of a model checking tool. In this case, for now, the only way to make model checking is to reduce the size of the module while preserve the correctness of the properties. This is the so called model reduction approaches.

Beyond compositional approaches, model reduction is the most important technique for relieving the state explosion problem. Model reduction is a general approach [8, 6] which allows to reduce a concrete system (M) under verification to a more abstract and smaller one (M'). Both systems M and M' are connected by an abstraction relation which is *safe* with respect to a given property φ, namely it preserves the property. This means if the property holds for the abstract system, it holds for the concrete one as well. More formally, the property φ is either *weakly* preserved if $M' \models \varphi \Rightarrow M \models \varphi$, or *strongly* preserved if $M' \models \varphi \equiv M \models \varphi$. It should be intuitively clear that the more weakly the property is preserved, the more reduction can be achieved.

A typical abstraction techniques is the *cone of influence reduction* (COI) method [17]. This method decreases the size of the concrete system by focusing on the variables of the concrete system that are referred to in the property and eliminating variables that do not influence the variables of interest against the properties. In this way, the property satisfaction is preserved, but the size of the model that needs to be verified is smaller. However, sometimes, there are still lots of redundant information in the COI reduced model. We can easily find a case in practice where a variable A depends on a variable B, but the *value* of variable B does not affect the *value* of variable A. For example, a two-input AND gate, if one of the inputs is set to zero, then no matter what value the other input takes, the output of the gate is always at zero.

Based on the above observation, we will give a refined dependency definition by examining the values of the variables that influence the truth of the property.

In the proposed approach, a system under verification is considered as a program for which abstract syntax and semantics are defined. The approach analyzes its syntactic structure, i.e., the *control flow diagram* of the programs [12]. Throughout the analysis, the dependencies of the state variable in the program are extracted. Then, using a SAT solver, SATO, [23], we can partition the values of state variables in the program into *active values*, and *deactive values* according to their dependency to the properties. The deactive values then can be replaced by a typical *abstract* value, and thus the value domains of the variables are much smaller than the original ones. Accordingly, we can have a reduced program with respect to the abstracted variables. As we will see, after the above procedures are done, the state space of the reduced program is smaller than that of the original one, while the correctness of the properties are preserved. The proposed approach has the same flavor as the static analysis approaches such as symbolic data abstraction [6] or abstract interpretation [8]. However, we do not have to define the abstract domains. Throughout the analysis, we show that the approach is efficient in reducing the size of the datapath of a system with finite value domains. Because the reduction is based on the analysis of the control flow diagram, which is a canonical form of the program, it can be automatic.

In order to achieve our goal, i.e., determining the active and deactive values, we will use two semantic functions adapted from [12]. Namely, the reachability condition RC_τ , associated with every path τ of the *Control Flow Diagram* (CFD), is a boolean condition under which this path is traversed. The state transformation function ST_τ computes the values of the program variables at the end of the path, provided that this path is traversed. These functions are obtained by backward induction over paths of the control flow diagram of the program. A problem to deploy this approach in the practice is on how to solve RC_τ and get the true value assignment. Computing all the combinations of variables and their domains would be exponential. However, SAT [13] solvers are good at finding such variable assignments. Namely, we can use a SAT solver to compute RC_τ.

The rest of the paper is structured as follows. In Section 2, we overview some prelimnaries. In Section 3, we define our system model including the abstract syntax and semantics of a program, i.e., the control flow diagram, and the Kripke structure of the program. We propose the reduction procedure in Section 4. In Section 5, we apply our model reduction approaches on a forward table lookup processor. In Section 6, we apply our approaches on some other benchmarks. Section 7 is the related work and the contributions. Finally, Section 8 concludes the paper.

2 Prelimnaries

2.1 Specifications

We consider the temporal logic specification language **ACTL** [6]. **ACTL** is a subset of **CTL** [4], where only *universal* path quantifier, namely only **A** is allowed. The set of well-formed universal computation tree logic (**ACTL**) are constructed from a set of atomic propositions AP which represent properties of individual states, the standard boolean operators, the temporal operators **X**, **U** and **V**, and the universal path quantifier **A**.

Definition 2.1 *Formulas in* **ACTL** *are inductively defined as follows:*

- *For every atomic proposition $p \in AP$, p and $\neg p$ are formulas.*
- *If φ and ψ are formulas then $\varphi \wedge \psi$, $\varphi \vee \psi$, $\mathbf{AX}\varphi$, $\mathbf{A}(\varphi\ \mathbf{U}\ \psi)$, and $\mathbf{A}(\varphi\ \mathbf{V}\ \psi)$ are formulas.*

Below, we give the semantics of **ACTL** formulas relative to a Kripke structure $\mathcal{K}$.

Definition 2.2 *The relation $s \models \varphi$, where s is a state, is inductively defined as follows:*

- *$s \models p$ iff $p \in \lambda(s)$*
- *$s \models \neg p$ iff $\neg p \in \lambda(s)$*
- *$s \models \varphi \wedge \psi$ iff $s \models \varphi$ and $s \models \psi$.*
- *$s \models \varphi \vee \psi$ iff $s \models \varphi$ or $s \models \psi$.*
- *$s \models \mathbf{AX}\varphi$ iff for every path π starting at the state s, $\pi^1 \models \varphi$.*
- *$s \models \mathbf{A}(\varphi\ \mathbf{U}\ \psi)$ iff for every path π starting at the state s, there exists i such that $\pi^i \models \psi$ and $\pi^j \models \varphi$ holds for $j < i$.*
- *$s \models \mathbf{A}(\varphi\ \mathbf{V}\ \psi)$ iff for every path π starting at the state s, and every $j \geq 0$, if for every $i < j, \pi^i \not\models \varphi$ then $\pi^j \models \psi$.*

We say that $\mathcal{K}$ satisfies a formula φ, written $\mathcal{K} \models \varphi$, if for every state s in the initial states, $s \models \varphi$.

2.2 The Satisfiability Problem

The SATisfiability (SAT) problem is a core problem of a large family of computationally intractable NP-complete problems. Such problems have been identified as central to a number of areas in computing theory and engineering. An instance of the SAT problem is a Boolean formula that has three components [13]:

- A set of n variables: $x_1, x_2, \ldots, x_n$.
- A set of literals. A literal is a variable ($Q = x$) or a negation of a variable ($Q = \neg x$).
- A set of m distinct clauses: $C_1, C_2, \ldots, C_m$. Each clause consists of only literals combined by just *or* ($\vee$) connectives.

The goal of the satisfiability problem is to determine whether there exists an assignment to truth values to variables that makes the following Conjunctive Normal Form (CNF) formula satisfiable:

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m,$$

where $\wedge$ is a logic *and* connective.

Since SAT is NP-complete, it is unlikely that any SAT algorithm has a fast worst-case time behavior, namely the worst-case time is exponential [13]. However, clever algorithms can rapidly solver many SAT formulas of practical interest. There has been great interest in designing efficient algorithm to solve most SAT formulas. In practice, SAT is fundamental in solving many problems in automated reasoning, computed-aided design, machine vision, integrated circuit design automation and computer network design. Therefore, methods to solve SAT formulas play an important role in the development of efficient computing systems [13].

SATO [24, 23] is one of the provers for the SAT problem of propositional logic based on Davis-Putnam method [9]. It uses a set of optimization methods in the analysis of the CNF propositional logic formulas. Compared with other SAT tools, it is the most efficient one for the moment [23].

SATO expects an CNF formula in DIMACS or Lisp format [24]. For example, an CNF formula $(\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$ can be expressed as $(-120, 1-20)$ in DIMACS format or $((-12), (1-2))$ in Lisp format. SATO accepts this formula and will give the models satisfying the formula, i.e., $x_1 = x_2 =$ **true** or $x_1 = x_2 =$ **false**. However, for formula $(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1))$, no matter what you try, there is no satisfying assignment since x_1 must be **false** (third clause), so x_2 must be **false** (second clause), but then the first clause is unsatisfiable.

3 System Models

In this section, we model the various designs under verification by an abstract program including the syntax and the semantics. Generally, what we want to model is a reactive system which reacts to the stimulus by changing the value of the variables in the program. The reaction of the system is continuous starting from an initial condition. For simplicity, we only define two types of statements of the program. One is assignment, the other is test. Using these two statements, we can simulate most of the cases in the computation.

The semantics of the program is defined by the Kripke structure [15], where each state is an execution result of the statements in the program. The transition between two states is the reflexive and transitive closure of two consecutive statements in the execution. Intuitively, assuming there is a clock signal in the program and all the statements are executed in each clock cycle, one can consider that a state is the valuations of all the variables in the program, which are sampled at one edge of the clock.

In the following, we define the syntax and the semantics respectively.

3.1 Abstract Program Syntax

Definition 3.1 *An* abstract program P *is specified by a tuple* (V, I, S)*, where*

- $V = \{v_1, \ldots, v_n\}$ *is a finite, non-empty set of variables. Each variable* v_i*,* $i \in [1 : n]$*, has an associated finite domain of values,* $dom(v_i)$*.*
- $I(V)$ *is the initial condition, specified as assignments on* V*.*
- S *is a set of statements which are partitioned into the assignment statements* S_a *and the test statements* S_t*.*

Given a program $P = (V, I, S)$, the *control flow diagram* (CFD) of this program is defined as follows.

Definition 3.2 *A Control flow diagram (CFD) of a program* $P = (V, I, S)$ *is a directed diagram* (N, E, L_e) *where*

- N *is a finite set of nodes labeled by the program counter locations.*
- $E \subseteq N \times N$ *is a finite set of edges.*
- L_e *is the edge labeling function* $L_e : (E \rightarrow S)$ *such that for every node* n *in the diagram either* n *is of out-degree 1 and the edge leaving* n *is labeled with assignment statement(s) or* n *is of out-degree 2 and the edges leaving* n *are labeled with test statement* S_t *and* $\neg S_t$*.*

There are three special nodes in a CFD, i.e., $Init$, ϵ, and ω. These nodes are part of the program syntax, where

- $Init \in N$ is the entry point of the program and marks the initialization block of the program, which has no predecessor.
- $\epsilon \in N$ is the beginning of the code. The edge to ϵ is marked by $I(V)$.
- $\omega \in N$, marks the end of the program text, where the code between ϵ and ω is in a forever loop.

In the test statement, "*if B then* $proc_1$ *else* $proc_2$" or "*while B then* $proc_1$", the *true action domain* of the test is the set of statements in $proc_1$, which will be executed when the test is **true**, and denoted by $AD_{S_t^+}$. The *false action domain* of the test is the set of statements in $proc_2$, which will be executed when the test is **false**, and denoted by $AD_{S_t^-}$.

The CFD of program P is actually its graphic presentation. So, P can be translated into $CFD(P)$ according to the definitions or vice versa.

Example 3.1 *To illustrate our purpose, we have chosen a small program P. In Figure 1, we give P and its CFD. In program P, the value domains of the variables,* i, j, *and* k, *are* $\{0, 1, 2, 3\}$, *respectively. The initial condition is "*$i = j = k = 0$*", which is the starting point of program P. There are two kinds of statements in P, assignment and test. The statements between* ϵ *and* ω *are in a forever loop, namely when statement* L_2 *is finished, statement* ϵ *will start again automatically. In the CFD, nodes are marked by the program counter locations* $(\epsilon, L_0, L_1, L_2, \omega)$ *and edges are marked by the statements in the program. Since the code between* ϵ *and* ω *is in a forever loop, the edge from* ω *to* ϵ *is a "forever" transition.*

3.2 Abstract Program Semantics

Let $\overrightarrow{v} = \{v_1, \ldots, v_k\}$ be a vector of variables taking their values in a *universe* $U_{\overrightarrow{v}}=\{dom\{v_1\}, \ldots, dom\{v_k\}\}$. Semantically, we refer to the valuation of the variables $v_1 = e_1, \ldots, v_k = e_k$, where $e_i \in dom(v_i)$, as a *state* σ, and these variables v_i are often referred to as *state variables.* We use the notation $\sigma[\![v_i]\!]$ to denote the value of state variable v_i in state σ. Similarly, for an expression x (over state variables in the domain of state σ), we use $\sigma[\![x]\!]$ to denote the value of x in the state σ. The special brackets '$[\![$' and '$]\!]$' emphasize the fact that the entity enclosed between them (v_i or x in the above cases) is of a syntactic nature.

The *initial condition* $I(V)$ serves as the entry point of the program just like the reset value of a circuit, where the computation starts. In the initial condition, all the variables have to be initialized, however, this is not an essential restriction.

The *assignment* statement is denoted by S_a, where an assignment $\overrightarrow{v_1} = T(\overrightarrow{v_2})$ is represented as a function T from U_{v_2} into $U_{\overrightarrow{v_1}}$. In the simple case, $v_i = T(\overrightarrow{v_2})$. In the above

```
// Variable declaration
i, j, k : {0,1,2,3};
Init:    i = 0;
         j = 0;
         k = 0;
epsilon: i = (i + 1) mod 4;
L0:      j = (j + 1) mod 4;
L1:      if (i == 3)
L2:        k = 1;
omega:
```

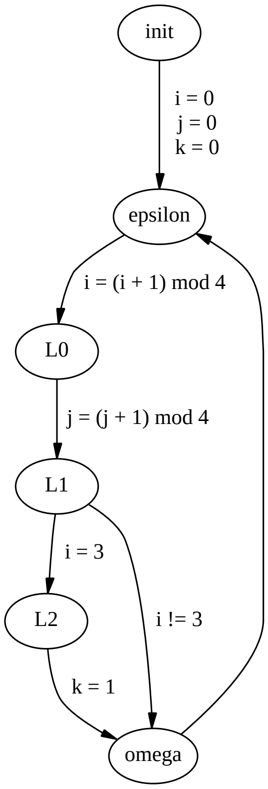

Figure 1: Program P and its CFD

assignment statement, $\overrightarrow{v_1} = T(\overrightarrow{v_2})$ can be multiple assignment, where the current value of the expression $T(\overrightarrow{v_2})$ can be assigned to the variable v_i simultaneously for all i. For example, $(x_1, x_2) = (x_1+1, x_1+x_2)$ assigns to x_1 the successor of its value and to x_2 the sum of the old value of x_1 and x_2. However, two assignment statements, "$\overrightarrow{v_1} = T(\overrightarrow{v_2}); \overrightarrow{v_3} = T'(\overrightarrow{v_4})$", are sequential, namely statement $\overrightarrow{v_3} = T'(\overrightarrow{v_4})$ can be executed only after statement $\overrightarrow{v_1} = T(\overrightarrow{v_2})$. For example, two assignment statements, "$x_1 = x_1 + 1; x_2 = x_1 + x_2$", x_1 is assigned the successor of its value and x_2 the sum of the *new* value of x_1 and x_2.

The *test* statement is denoted by S_t, where a test statement, "$if(B(\overrightarrow{v}))$" or "$while(B(\overrightarrow{v}))$", is represented as a function B from $U_{\overrightarrow{v}}$ into $\{\mathbf{true}, \mathbf{false}\}$. The test statement is a positive test in a state σ when $\sigma[\![B(\overrightarrow{v})]\!] = \mathbf{true}$, or a negative test when $\sigma[\![B(\overrightarrow{v})]\!] = \mathbf{false}$. The computation branches according to the value of the test function.

The semantics of program P is defined in terms of *configurations*. A configuration of a program is a pair $c = \langle \sigma, \lambda \rangle$. The state σ is a visible component and assigns values to program variables. The *counter location* λ is a hidden component and its value corresponds to a current statement to be executed.

Next, we define the particular transition relation "$\rightarrow$" for configurations.

Definition 3.3 *Let $c = \langle \sigma, \lambda \rangle$ and $c' = \langle \sigma', \lambda' \rangle$ be two configurations. We write $c \rightarrow c'$ if and only if the following hold.*

- *If λ is the $Init$ node, then $\lambda' = \epsilon$ and $\sigma' = \sigma$.*
- *If λ is the ω node, then $\lambda' = \epsilon$ and $\sigma' = \sigma$.*
- *If λ is an assignment $S_a : \overrightarrow{v_1} = T(\overrightarrow{v_2})$, where $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ are two vectors and T is a function from $U_{\overrightarrow{v_2}}$ into $U_{\overrightarrow{v_1}}$, then $\lambda' = suc(\lambda)$ and $\sigma'[\![\overrightarrow{v_1}]\!] = \sigma[\![T(\overrightarrow{v_2})]\!]$, where $suc(\lambda)$ means the successor of λ in CFD.*
- *If λ is a test statement $S_t : if(B(\overrightarrow{v}))$ or $while(B(\overrightarrow{v}))$, then $\sigma' = \sigma$, and either $\lambda' = suc^+(\lambda)$, in the case $\sigma[\![B(\overrightarrow{v})]\!]$ is true or $\lambda' = suc^-(\lambda)$ in the case $\sigma[\![\neg B(\overrightarrow{v})]\!]$ is true, where $suc^+(\lambda)$ and $suc^-(\lambda)$ means the true or false successor of λ in a CFD, respectively.*

We will use "$\rightarrow^*$" to denote the reflexive transitive closure of "$\rightarrow$".

Generally, "$\rightarrow$" is the transition from one statement to the next statement in the execution. However, "$\rightarrow^*$" is the transition after n continuous statements. During these n steps, the execution may either include arbitrary number of steps of $\rightarrow$, or do nothing ($n = 0$).

Definition 3.4 *The semantics of the program $P = (V, I, S)$ is a Kripke structure $\mathcal{K}(P) = (\Sigma, \langle \sigma_0, Init \rangle, \rightarrow, L)$, where*

- Σ *is a set of configurations, where* $\Sigma = \{\langle\sigma', \epsilon\rangle : \langle\sigma, \epsilon\rangle \to^* \langle\sigma', \epsilon\rangle\} \cup \{\langle\sigma_0, \epsilon\rangle : \langle\sigma_0, Init\rangle \to \langle\sigma_0, \epsilon\rangle\}$.
- $\langle\sigma_0, Init\rangle \in \Sigma$ *is an initial configuration such that* $\sigma[\![I(V)]\!]$ *is true.*
- $\to$ *is a transition relation defined in Definition 3.3.*
- *a labeling function* $L : (\Sigma, 2^{AP})$ *associates a non-empty set of atomic propositions (AP) with a configuration* $c \in \Sigma$.

Example 3.2 *The Kripke structure of the program in Example 3.1 is shown in Figure 2. The double circle is the initial state where the labels of the initial states are decided by the initial assignments. Here, since in the Kripke structure, the value of* λ *is always* ϵ, *we omit* ϵ *in the label of the configurations, namely* $< (0,0,0), \lambda >$ *becomes* $(0,0,0)$. *In this example, the initial condition is "*$i = j = k = 0$*", so the initial state is "*$i = j = k = 0$*". The labels of the other states are decided by the valuation of the variables in the program when program counter is* ϵ. *For example, if the current state is "*$i = j = 1, k = 0$*", according to the program, statement* L_2 *will not be executed because the test condition does not hold, so* k *keeps at* 0. *However, in this case, statement* ϵ *and* L_0 *are executed, the consequential results are "*$i = j = 2$*". When the execution gets to* ω, *the valuation of the variables are "*$i = j = 2, k = 0$*". So, the next state of "*$i = j = 1, k = 0$*" is "*$i = j = 2, k = 0$*".*

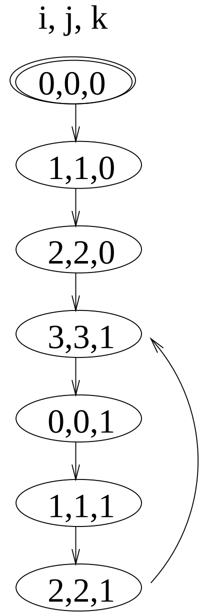

Figure 2: Kripke Structure of Example 3.1

4 Reduction Methods

4.1 Reduction Via Data Dependency

Data dependency defines the relationship between two state variables. If one variable affects the values of another variable during the computation, we call these two variables *dependent*. Generally, in the statements of a program, there are two types of dependency. One is the statement with direct data dependency; the other is that with indirect data dependency. Direct data dependency means that the value of one variable will directly affect the value of the other variable.

Definition 4.1 *Direct data dependency is the data dependency in the assignment statement* $Sa : \overrightarrow{v_1} = T(\overrightarrow{v_2})$, *where* $\overrightarrow{v_1}$ *directly depends on* $\overrightarrow{v_2}$.

For example, in the assignment statement "$x = y + 1$", the value of the RHS (Right Hand Side) variable y will affect the value of LHS (Left Hand Side) variable x directly.

Indirect data dependency means that the value of one variable will affect the value of the other variable via another variable.

Definition 4.2 *Indirect data dependency is the data dependency in the test statements,* "$if(B(\overrightarrow{v}))$" *or* "$while(B(\overrightarrow{v}))$". *Variables in* $\overrightarrow{v}$ *indirectly affects the LHS variables in* $AD_{S_t^+}$ *and* $AD_{S_t^-}$.

For example, in the "$if(z)$ *then* $w = y + 1$ *else* $w = y - 1$" statement, the value of z will affect the value of the LHS variable w indirectly via variable y. Another example from hardware, such as a D flip-flop as shown in Figure 3, the relation between D and Q is the direct dependency, and the relation between Clr and Q is the indirect dependency.

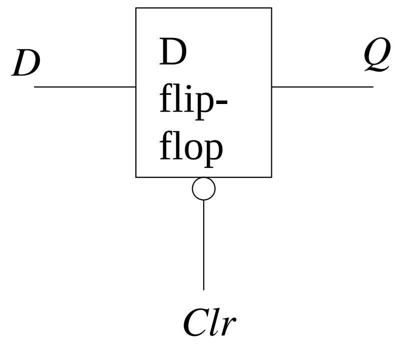

Figure 3: An D Flip-flop

A Data Dependency Diagram (DDD) of program P is a directed diagram which records the dependency between the variables in the program. The data dependency diagram is defined as follows:

Definition 4.3 *The data dependency diagram of a program (V, I, S) is represented as a directed graph (D, L_d, X), where*

- *D is the set of nodes.*
- *L_d is a one-to-one function $L_d : (D, V)$ which labels each node by a variable.*
- *$X \subseteq D \times D$ is the set of transitions between the nodes to mark the dependency via directions, $(d_i, d_j) \in X$ if and only if there is a direct dependency between v_i and v_j, i.e., $v_j = T(v_i)$, or there is an indirect dependency between v_i and v_j, i.e., v_i in $B(\overrightarrow{v}))$ and v_j in the LHS of $AD_{S_t^+}$ or $AD_{S_t^-}$. Namely v_i is either RHS of S_a or in $B(\overrightarrow{v})$; v_j is either LHS of S_a or LHS in $AD_{S_t^+}$ or $AD_{S_t^-}$ with respect to $B(\overrightarrow{v})$.*

Example 4.1 (DDD of the search program) *In example 3.1, the DDD of the program is shown in Figure 4, where only i affect k. Because i is in the test condition of statement L_1 and k is the LHS of L_2, this is an indirect dependency.*

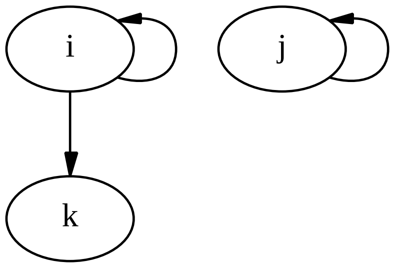

Figure 4: DDD of Example 3.1

Since in each verification run, a property is always a partial specification, and not all the state variables in the implementation are involved in the verification, we can prune those unnecessary state variables in each specific verification. Thus, the system to be verified will be smaller in terms of the number of state variables. This idea is the so-called *cone of influence* (COI) reduction [17, 7].

Generally, suppose we want to verify a specification φ, where $V' \subseteq V$ is the set of variables that are of interest with φ. We would like to reduce the original model P with respect to the specification φ, namely we will abstract away all the variables and transitions which are not of interest with the specification φ (property localization). However, because of the internal transitions in P, the values of variables in V' might depend on values of variables not in V'. Therefore, we define a cone of influence C of V' and use C in order to reduce the original model P.

The cone of influence $C = \{v_1, \dots, v_k\}$ of V' is the minimal set of variables such that

- $V' \subseteq C$
- if for some $v_l \in C$, and $v_j \in V$, v_l depends on v_j either directly or indirectly, then $v_j \in C$.

C is created by removing all the variables where the nodes marked by these variables cannot reach the nodes marked by the variables of interest in DDD. Since C is the subset of the variables V of model P and includes all the variables of interest against specification φ, it is an abstraction of the variables in P.

Theorem 4.1 *[6] Let f be a* **CTL*** *formula with atomic proposition in C, and P is the concrete model, $\widehat{P}$ is the COI reduced model. Then*

$$P \models f \Leftrightarrow \widehat{P} \models f$$

Example 4.2 (COI reduced DDD and CFD) *In Example 3.1, assuming we want to prove a property of variable k, the DDD of the COI reduced program is shown in Figure 5, where only i affect k and j is removed because the node marked by j cannot reach the node marked by k via a path in the diagram.*

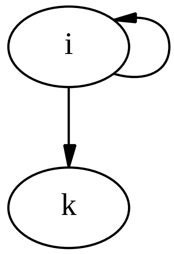

Figure 5: COI Reduced DDD

The reduced program is shown in the left of Figure 6 and The corresponding CFD is shown in the middle. The state space of the program after the COI reduction is shown in the right.

4.2 Reachability Condition and State Transform

In this section, we introduce two notations used in the following reduction approach, i.e., *reachability condition* and *state transformation.* These two notations are obtained from CFD.

For a sequence of nodes in CFD, $N_i \overset{E_{i+1}}{\to} N_{i+1} \overset{E_{i+2}}{\to} \ldots \overset{E_{i+k}}{\to} N_{i+k}$, where $k \geq 1$, we say this is a *finite path* in the diagram. We use the notation $expr[v \leftarrow e]$ for the expression that every occurrence of v in $expr$ is replaced by the value e. For example, "$x + y[x \leftarrow 1, y \leftarrow 1] = 2$" means that x and y are replaced by constant 1 respectively, and the result of the expression is 2. By convention, $\mathbf{true}[v \leftarrow e] = \mathbf{true}$ and $\mathbf{false}[v \leftarrow e] = \mathbf{false}$.

```
// Variable declaration
i, k : {0,1,2,3};
Init:    i = 0;
         k = 0;
epsilon: i = (i + 1) mod 4;
L1:      if (i == 3)
L2:        k = 1;
omega:
```

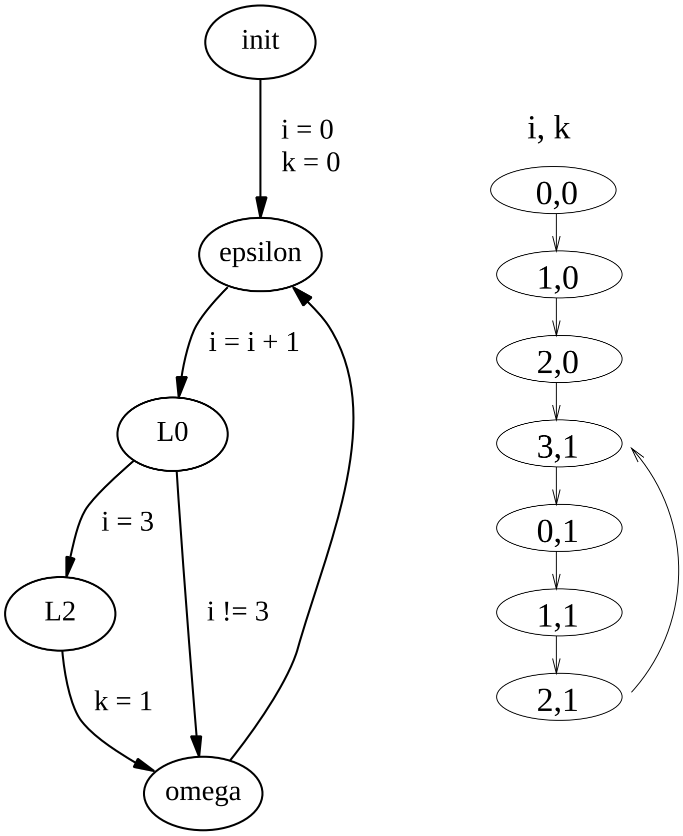

Figure 6: COI Reduced Program and its CFD and State Space

Definition 4.4 *[12] Let τ be a finite path in CFD.*

- Reachability condition *guarantees that the control will traverse τ. It is denoted by $RC_\tau(\overrightarrow{v})$, where $\overrightarrow{v}$ is the variable vector of interest.*
- State transformation *is the final state of $\overrightarrow{v}$ obtained if control indeed traverses τ with $\overrightarrow{v}$. It is denoted by $ST_\tau(\overrightarrow{v})$.*

$RC_\tau(\overrightarrow{v})$ and $ST_\tau(\overrightarrow{v})$ are obtained by backward induction over τ as follows.

Let $\tau = N_0 \overset{E_1}{\rightarrow} \ldots \overset{E_{k-1}}{\rightarrow} N_k$ be a finite path in CFD of length $k+1$, for some natural number k. We first define $RC^m_\tau(\overrightarrow{v})$ and $ST^m_\tau(\overrightarrow{v})$, the corresponding characteristics of the suffix $N_m \overset{E_{m+1}}{\rightarrow} \ldots \overset{E_{k-1}}{\rightarrow} N_k$ of τ, by induction on m going down from k to 0.

- *Induction basis*
  $RC^k_\tau(\overrightarrow{v}) = true$, $ST^k_\tau(\overrightarrow{v}) = \overrightarrow{v}$
  In other words, being at the end of τ to traverse τ implies an identically true reachability condition and an identity state transformation.
- *Induction assumptions*
  Suppose that $RC^{m+1}_\tau(\overrightarrow{v})$ and $ST^{m+1}_\tau(\overrightarrow{v})$ have already been defined for $0 \leq m < k$
- *Induction step*
  We define $RC^m_\tau(\overrightarrow{v})$ and $ST^m_\tau(\overrightarrow{v})$ according to the statements at edge E_m.
  - *Initialization*: $RC^m_\tau(\overrightarrow{v}) = (RC^{m+1}_\tau(\overrightarrow{v}))[\overrightarrow{v} \leftarrow I(V)]$, $ST^m_\tau(\overrightarrow{v}) = ST^{m+1}_\tau(\overrightarrow{v})[\overrightarrow{v} \leftarrow I(V)]$. Thus, the initial statement is captured by an assignment of new value to the variables.
  - *Assignment*: $RC^m_\tau(\overrightarrow{v}) = (RC^{m+1}_\tau(\overrightarrow{v}))[\overrightarrow{v} \leftarrow T(V)]$, $ST^m_\tau(\overrightarrow{v}) = ST^{m+1}_\tau(\overrightarrow{v})[\overrightarrow{v} \leftarrow T(V)]$. Thus, the effect of an assignment is captured by substitution.
  - *Positive test*: $RC^m_\tau(\overrightarrow{v}) = (RC^{m+1}_\tau(\overrightarrow{v})) \wedge B(\overrightarrow{v})$, $ST^m_\tau(\overrightarrow{v}) = ST^{m+1}_\tau(\overrightarrow{v})$. Thus a test does not change the state and remembers the condition.
  - *Negative test*: $RC^m_\tau(\overrightarrow{v}) = (RC^{m+1}_\tau(\overrightarrow{v})) \wedge \neg B(\overrightarrow{v})$, $ST^m_\tau(\overrightarrow{v}) = ST^{m+1}_\tau(\overrightarrow{v})$. This is similar to the positive test case.

  Finally, we define $RC_\tau(\overrightarrow{v}) = RC^0_\tau(\overrightarrow{v})$, and $ST_\tau(\overrightarrow{v}) = ST^0_\tau(\overrightarrow{v})$.

Example 4.3 (ST and RC of a program) *Consider the program in Example 3.1. If we set τ as a path $Init \rightarrow \epsilon \rightarrow L_0 \rightarrow L_1 \rightarrow \omega$, the reachability condition and the state transformation are as follows.*

- $RC^\omega_\tau(i,j,k) = \textbf{true}$, $ST^\omega_\tau(i,j,k) = (i,j,k)$.

- $RC_\tau^{L_1}(i,j,k) = \textbf{true} \wedge (i \neq 3) = (i \neq 3)$, $ST_\tau^{L_1}(i,j,k) = (i,j,k)$.
- $RC_\tau^{L_0}(i,j,k) = (i \neq 3)[j \leftarrow (j+1)] = (i \neq 3)$, $ST_\tau^{L_0}(i,j,k) = (i,j,k)[j \leftarrow (j+1)] = (i,j+1,k)$.
- $RC_\tau^{\epsilon}(i,j,k) = (i \neq 3)[i \leftarrow (i+1)] = (i \neq 3)$, $ST_\tau^{\epsilon}(i,j,k) = (i,j+1,k)[(i \leftarrow (i+1))] = (i+1,j+1,k)$.
- $RC_\tau^{Init}(i,j,k) = (i \neq 3)[i \leftarrow 0, j \leftarrow 0, k \leftarrow 0] = \textbf{true}$, $ST_\tau^{Init}(i,j,k) = (i,j+1,k)[i \leftarrow 0, j \leftarrow 0, k \leftarrow 0] = (1,1,0)$.

From the above induction, we know $RC_\tau(i,j,k) = \textbf{true}$ *and* $ST_\tau(i,j,k) = (1,1,0)$ *which means that node* ω *can be reached fro node* $Init$ *by path* $Init \rightarrow \epsilon \rightarrow L_0 \rightarrow L_1 \rightarrow \omega$ *and after the traverser the valuations of the variables are* $(1,1,0)$*, respectively.*

Example 4.4 (ST and RC of a program) *Consider the program in the left of Figure 7. The CFD of this program is shown in the right.*

```
// Variable declaration
int x;
Bool y,z;

// Initialization
Init:    x = 1;
         y = 1;
         z = 1;

epsilon: if (y <= x)
L0:        y = y + z;
         else
L1:        x = x + z;
        endif
omega:
```

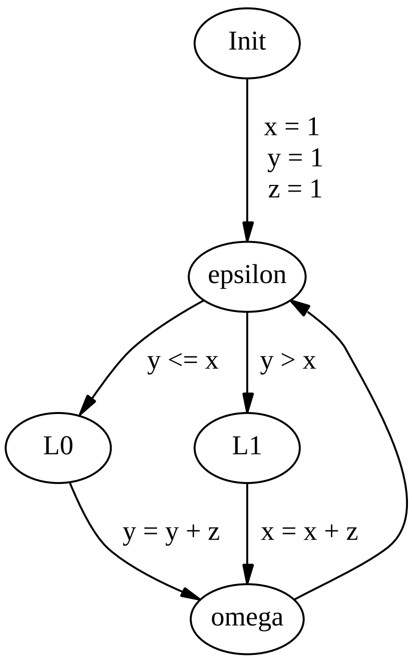

Figure 7: An RC/ST Example

If we set τ *as a path* $\epsilon \rightarrow L_0 \rightarrow \omega$*, the reachability condition and the state transformation are as follows.*

- $RC_\tau^{\omega}(x,y,z) = \textbf{true}$, $ST_\tau^{\omega}(x,y,z) = (x,y,z)$

- $RC_{\tau}^{l_0}(x, y, z) = \mathbf{true}[y \leftarrow (y + z)] = \mathbf{true}$, $ST_{\tau}^{l_0}(x, y, z) = (x, y, z)[y \leftarrow (y + z)] = (x, y + z, z)$

- $RC_{\tau}^{\epsilon}(x, y, z) = \mathbf{true} \wedge (y \leq x) = (y \leq x)$, $ST_{\tau}^{\epsilon}(x, y, z) = (x, y + z, z)$

From the above induction, we know $RC_{\tau}(x, y, z) = (y \leq x)$ and $ST_{\tau}(x, y, z) = (x, y+z, z)$. The actual values of $RC_{\tau}(x, y, z)$ and $ST_{\tau}(x, y, z)$ depend on the present values of $x, y,$ and z at node ϵ. In the computation right after the initialization $x = 1$, $y = 1$ and $z = 1$, $RC_{\tau}(x, y, z) = \mathbf{true}$ and $ST_{\tau}(x, y, z) = (1, 2, 1)$.

4.3 Deactive Variables Reduction

As we mentioned, even after COI reduction, the reduced model still has some redundant information which will create a redundant state space. In the context of model checking, this redundancy means much more CPU time and memory usage in the verification. In this section, we give a refined dependency definition of the COI approach. In the following, we consider whether or not the change of the value of a variable v_i will lead to a change of the value of another variable v_j. If so, then variable v_j is indeed depending on variable v_i. For example, v_j is the variable in a property. If the value change of v_i will alter the value of v_j, then variable v_i will definitely affect the truth of the property, else even v_i and v_j are dependent with respect to the COI dependency, v_i is a redundant variable according to the property.

The basic approach to deploy the above idea is to partition the value domain of the variables in the program. In the following, we will partition the value domain into *active values* and *deactive values*. The active values are those values which will affect the variables of interest, while the deactive values do not affect the variables. The partition is based on the analysis of the two notations of control flow diagram (CFD), i.e, reachability condition RC and state transform ST.

Now, we are able to define our partition of value domains. We consider first the *key nodes*, which are nodes of CFD that directly influence the specification. Next, we compute the reachability condition and state transformation function of the path starting from a node in CFD and ending at a key node.

Definition 4.5 *Let v be the variable of interest in the property, n be a node in CFD and e be the outgoing edge of n. We say that n is a* key node *if and only if $L_e(e) \triangleq v = T(\overrightarrow{v})$.*

For example, if i is the variable of interest, then a node n in CFD with the outgoing edge labeled by $i = T(\overrightarrow{v})$ is the key node. In the keynode, the value of i is to be changed, so the correctness of the properties of i will be affected. However, i is not the only variable in the program which can affect the correctness of the properties, because the variables may affect the correctness of the properties indirectly by affecting the value of i.

In the program, a variable j may affect variables i if there is a path such that the CFD node marked by the label of i is reachable from the node marked by the label of j. For example, there are two continues assignment statements in program P as shown in the left of Figure 8. The CFD of the program is on the right of the Figure. The value change of j will

```
L1:  j = j +1;
L2:  i = j +1;
...
```

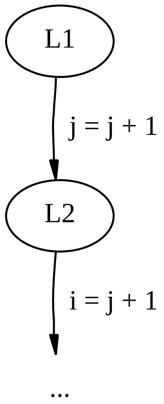

Figure 8: Dependency Example 1

affect the value of i since the CFD node L_2 is reachable from L_1 via the path "$L_1 \to L_2$". On the other hand, Consider the piece of program in the left of Figure 9, where the corresponding CFD is shown in the right of the figure. The CFD node L_3 is only reachable from L_1 when

```
L1:  j = j +1;
L2:  if (j > 10)
L3:      i = j +1;
...
```

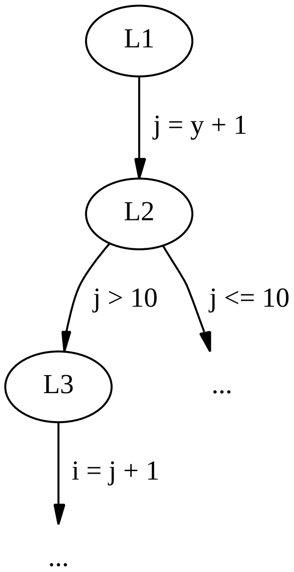

Figure 9: Dependency Example 2

$j \geq 10$, which means that all the values of j less than 10 cannot affect the value of i in this piece of code. Moreover, we can consider that only the values of j greater than 10 will affect the properties where i is the variable of interest, so we will keep this partition of value in the program, i.e., active values, and abstract the value of j less than 10, i.e., deactive values.

Definition 4.6 *Let v be a variable among the dependency list of interest variables, e the value of v and τ be a path leading to a key node.*
We say that the value e is active *if and only if $RC_\tau[v \leftarrow e]$ is true or $ST_\tau(v) = e$. Similarity, we say that the value e is* deactive *if and only if $RC_\tau[v \leftarrow e]$ is false.*

To use above definition in the practice, a problem is on how to find all the true value assignments for the reachability condition. Since SAT solvers are good at finding variable assignments which make the CNF propositional formula true, we can use a SAT solver, here SATO [23], to get all the values satisfying formula $RC_\tau[v \leftarrow e] = \textbf{true}$, which are the *active* values with respect to the RC. However, since SAT-solvers expect their input to be a propositional formula in CNF form, we still need to convert the reachability condition into CNF. Besides, when we do the conversion, since SATO does pure literal simplification [24], we have to keep a literal and its opposite in the CNF. Generally and during computations, a variable v may have both active and deactive values with respect to the key nodes. Hence, the domain values of v, i.e., $dom(v)$, is partitioned into disjoint subsets ACTIVE (v), eventually empty, of active values and DEACTIVE (v), eventually empty, of deactive values, where

- if ACTIVE $(v) = \emptyset$, then v is a dead variable and DEACTIVE $(v) = \{e\}$, where $e \in dom(v)$ and e is a typical value of $dom(v)$.
- if ACTIVE $(v) \neq \emptyset$ and DEACTIVE $(v) \neq \emptyset$, then v is a partial dead variable and DEACTIVE $(v) = \{e\}$, where e is a typical value of DEACTIVE (v).
- if DEACTIVE $(v) = \emptyset$, then v is a live variable and ACTIVE $(v) = dom(v)$.

From the above analysis, the variables of a program is partitioned into subsets with respect to the variable of interest, i.e., a set of live variables, that of dead variables, and that of partial dead variables. Every value of the active variables will affect the value of the variable of interest while that of the deactive variables cannot. In model checking, the variable of interest is the variable in the property. The value of dead variables cannot affect the correctness of the property. Consequently we can abstract these values and preserve the correctness of the property. This abstraction is shown in Figure 10, where all the values in a dead variable are mapped into a single value. This single value can be any value in the domain of the dead variable.

In the case of partial dead variables, the active values are kept while the deactive values are abstracted as shown in Figure 11, where the active values are kept while the deactive

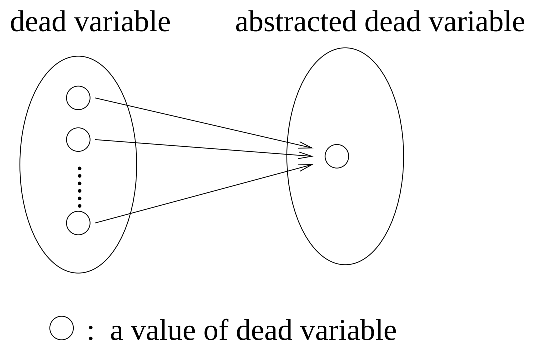

Figure 10: Abstraction of Dead Variables

values are mapped into a single value. This single value can be any value among the deactive values.

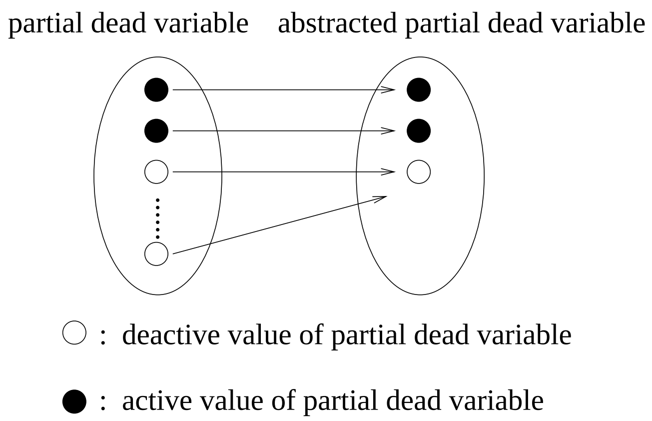

Figure 11: Abstraction of Partial Deactive Variables

In the case of live variables, all the values are kept as shown in Figure 12. In this case, no reduction is achieved.

Given the above reduction rules, we can obtain a reduced program from the original program as follows.

Definition 4.7 *$P = (V, I, S)$ is a program. The reduced program $\widehat{P} = (\widehat{V}, \widehat{I}, \widehat{S})$ is:*

- *$\widehat{V} = V$ is a finite, non-empty set of variables. For each $v_i \in \widehat{V}$, $dom(v_i) =$*ACTIVE *$(v_i) \cup$* DEACTIVE *(v_i).*
- *$\widehat{I}(\widehat{V})$ is equal to $I(V)[e_i \leftarrow$*DEACTIVE *$(v_i)]$ for each v_i such that $e_i \notin$*ACTIVE *(v_i).*
- *$\widehat{S}$ is obtained as follows: (1) if $B(\overrightarrow{v})$ or while $B(\overrightarrow{v})$ statements are obtained like the initial condition. (2) An abstract assignment statement $\widehat{S_a}$ is defined nondeterministically by $v_i =$*ACTIVE *$(v_i) \cup$* DEACTIVE *(v_i).*

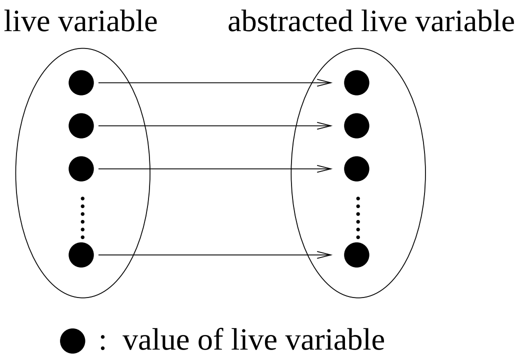

Figure 12: Abstraction of Active Variables

In the above reduced program, the variables in the program have new domains, i.e., ACTIVE $(v_i) \cup$ DEACTIVE (v_i). Since DEACTIVE (v_i) is actually the reduced set of deactive values, the abstract domain of v_i obviously is smaller than its original domain. Accordingly, because some values are missed from the program, the initial condition, assignment and test statement have to be revised as well. For the assignment statement, the new value of LHS variables is nondeterministically its active values and the typical deactive value. For test statement, if there is a deactive value in the test condition, then this value is replaced by the typical deactive value.

In the following, we will prove that the Kripke structure of the reduced program $\mathcal{K}(\widehat{P})$ simulates the original Kripke structure $\mathcal{K}(P)$ so that the reduction preserves **ACTL** properties.

Definition 4.8 *Let $B \subseteq \Sigma \times \widehat{\Sigma}$ be the relation defined as follows:*

$$((e_1, \dots, e_n), (\widehat{e}_1, \dots, \widehat{e}_n)) \in B$$
$$\Leftrightarrow$$
$$((\widehat{e}_i = e_i) \wedge (\widehat{e}_i \in \text{ACTIVE } (v_i))) \vee ((\widehat{e}_i \neq e_i) \wedge (\widehat{e}_i \in \text{DEACTIVE } (v_i)))$$

Theorem 4.2 *B is a simulation relation between the models $\mathcal{K}(P)$ and $\mathcal{K}(\widehat{P})$, namely $\mathcal{K}(P) \preceq \mathcal{K}(\widehat{P})$.*

PROOF:
1. For the initial configuration $c_0 \in \Sigma$ there exist an initial $\widehat{c}_0 \in \widehat{\Sigma}$. Since the abstract initial condition may differ only on deactive values, it follows trivially that $(c_0, \widehat{c}_0) \in B$

2. Let $c_1 \to c_2$ be a transition in $\mathcal{K}(P)$. We show that there is a transition $\widehat{c}_1 \to \widehat{c}_2$ such that $(c_2, \widehat{c}_2) \in B$, where $(c_1, \widehat{c}_1) \in B$. If the corresponding statement of λ_1 is an assignment, since the abstract assignment statement is nondeterministic, there is always a transition from

$\widehat{c}_1 \rightarrow \widehat{c}_2$ and $(c_2, \widehat{c}_2) \in B$ as required. For the *if* and *while* abstract statements, we update only inside their body the deactive values. Hence there is also a transition from $\widehat{c}_1 \rightarrow \widehat{c}_2$ and $(c_2, \widehat{c}_2) \in B$ as required.

3. By using steps 1 and 2, we can conclude that B is a simulation relation between $\mathcal{K}(P)$ and $\mathcal{K}(\widehat{P})$. □

Finally, since the reduced program simulates the original program, according to theorem 4.3, the **ACTL** properties are preserved.

Theorem 4.3 *[6] Suppose $M \preceq M'$, where M and M' are two models. Then for every* **ACTL** *formula φ, $M' \models \varphi \Rightarrow M \models \varphi$*

Example 4.5 (How the reduction works)

Considering the program P of Example 3.1, we want to verify a property about variable k. We can first apply COI reduction to find that variable j is irrelevant and thus can be removed. We will apply our approach based on the COI reduced program.

In the CFD, node L_2 is the key node since the statement $k = 1$ changes the value of k. Consider the value 0 of i. i is 0 at node ϵ. Through the backward reachability analysis from key node L_2 to ϵ, we find that $RC_{\epsilon \rightarrow L_2} =$ "$i = 3$". According to Definition 3 and using a SAT solver SATO [23], $RC_{\epsilon \rightarrow L_2}[i \leftarrow 0] =$ " $0 = 3$" which is false. So, the value 0 of i is a deactive value. In a similar way, we find that value 1 and 2 do not satisfy RC, so 1 and 2 are deactive values [1]. The value 3 of i is active since L_2 is reachable in path ϵ, L_1, L_2 ($RC_{\epsilon \rightarrow L_2}[i \leftarrow 3] = true$) while $ST_{\epsilon \rightarrow L_2}(i) = 3$. Hence, we can use abstract value domain $\{2, 3\}$ instead of its concrete value domain $\{0, 1, 2, 3\}$. Using the same approach, the abstract value domain of variable k is $\{0, 1\}$.

```
// Variable declaration
i : {2,3};
k : {0,1};
// Initialization
Init:    i = 2;
         k = 0;
epsilon: i = ND(2,3);
L1:      if (i == 3)
L2:        k = 1;
omega:
```

[1] In practice, we input the corresponding two bits propositional formula of $i = 3$ to SATO, SATO outputs that in order to satisfy the formula, both bits of i have to be 1.

In Figure 13, the left part is the state space of the program. It has 7 states. On the right side, there is the reduced state space. We can find that the concrete state space is divided into three blocks according to the deactive and active values, and each block is mapped to one abstract state, respectively.

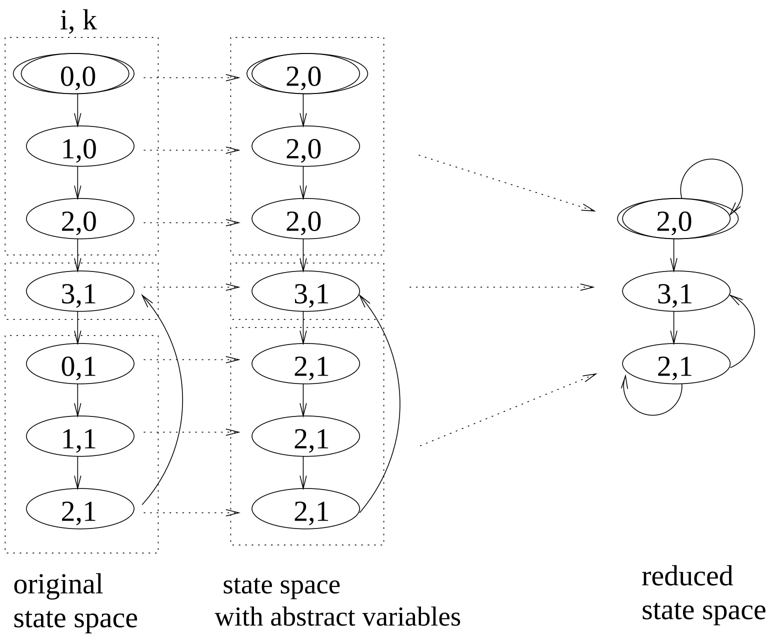

Figure 13: State Space Reduction

Example 4.6 (Reduction of the counter) *Consider the program in the left of Figure 14. The property to verify is $y \geq 0$. The CFD of the program is shown in the right of the Figure.*

In the CFD, the key node is L_1 because the outgoing edge is labeled by $j = i$. Since the RC to the key node must contain a condition "$i \geq 1000$", the key node is unreachable from $\{\epsilon, L_0\}$ with respect to the value $\{1, \ldots, 999\}$ of i, So, $\{1, \ldots, 999\}$ are the deactive values of i, yet the value 1000 of i is an active value since from path $\{\epsilon, L_1, \omega\}$, the key node is reachable with respect to the value 1000 of i. Hence, the abstract domain of i is $\{0, 1000\}$. However, in this example, the domain of variable j cannot be reduced since L_1 is reachable with respect to any value of j.

5 Application: A Forwarding Table Lookup Processor

With the expansion of communication networks, fast and intelligent routers are required. Usually, these routers perform multiple lookups per packet into potentially large tables.

```
// Variable declaration
{0..10000} i, j;

// Initialization
Init:    i = 0;
         j = 0;

epsilon: while (i < 1000) begin
L0:         i++;
         endwhile
L1:      j = i;
omega:
```

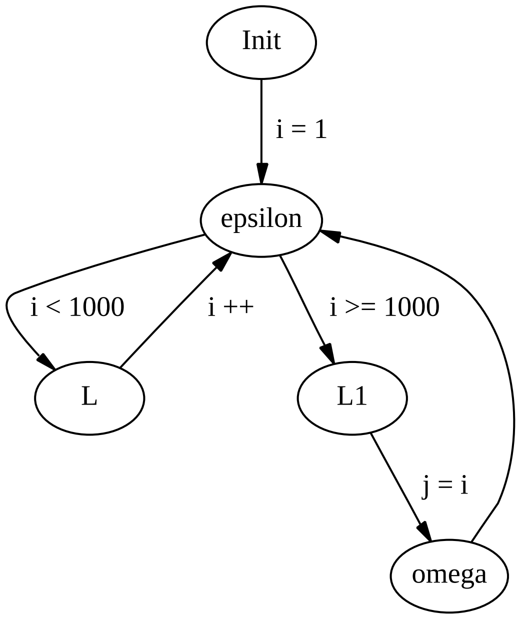

Figure 14: The Counter Program

For instance, the router of an ATM (Asynchronous Transfer Mode) switch fabric translates the incoming identifier (8-bit $Item\_In1$) into the destination identifier according to the routing table $Lookup\_Table$ [21]. Only those $Item\_In1$ inputs which can find a match ($Match\_Found = 1$) in the routing table can be switched. Here, we consider a simplified version of the original program and illustrate the application of our method.

```
//module name
module search;

 //Variables declaration
 reg [7 : 0] Lookup_Table [9 : 0];
 reg Match_Found;
 integer i, Error_Count;
 reg [7:0] Item_in1;

 //Initialization
Init:
 initialization  i=0; Match_Found = 0; Error_Count = 0;

epsilon: while (i <= 9) begin
L0:      if (Lookup_Table[i][7:0] == Item_In1) begin
L1:          Match_Found = 1;
```

```
L2:          i=9;
          endif
L3:       i ++;
          endwhile
L4:     if (!Match_Found)
L5:         Error_Count = Error_Count + 1;
        else
L6:         Error_Count = Error_Count - 1;
        endif
omega:
```

In the above program, if a match is found, the flag *Match_Found* is set. If after searching the whole table, no match can be found, the error counter (*Error_Count*) increments by one, else *Error_Count* decreases by one. The table totally has 10 items $\{00, 01, \ldots, 09\}$, and the length of each item is 16-bit. Each statement in the program is labeled by a program counter such as l_0, l_1 and so on, where $Init$ labels the initial statements, ϵ labels the beginning of the code, and ω labels the end of the code. The CFD of the search program is shown in Figure 15.

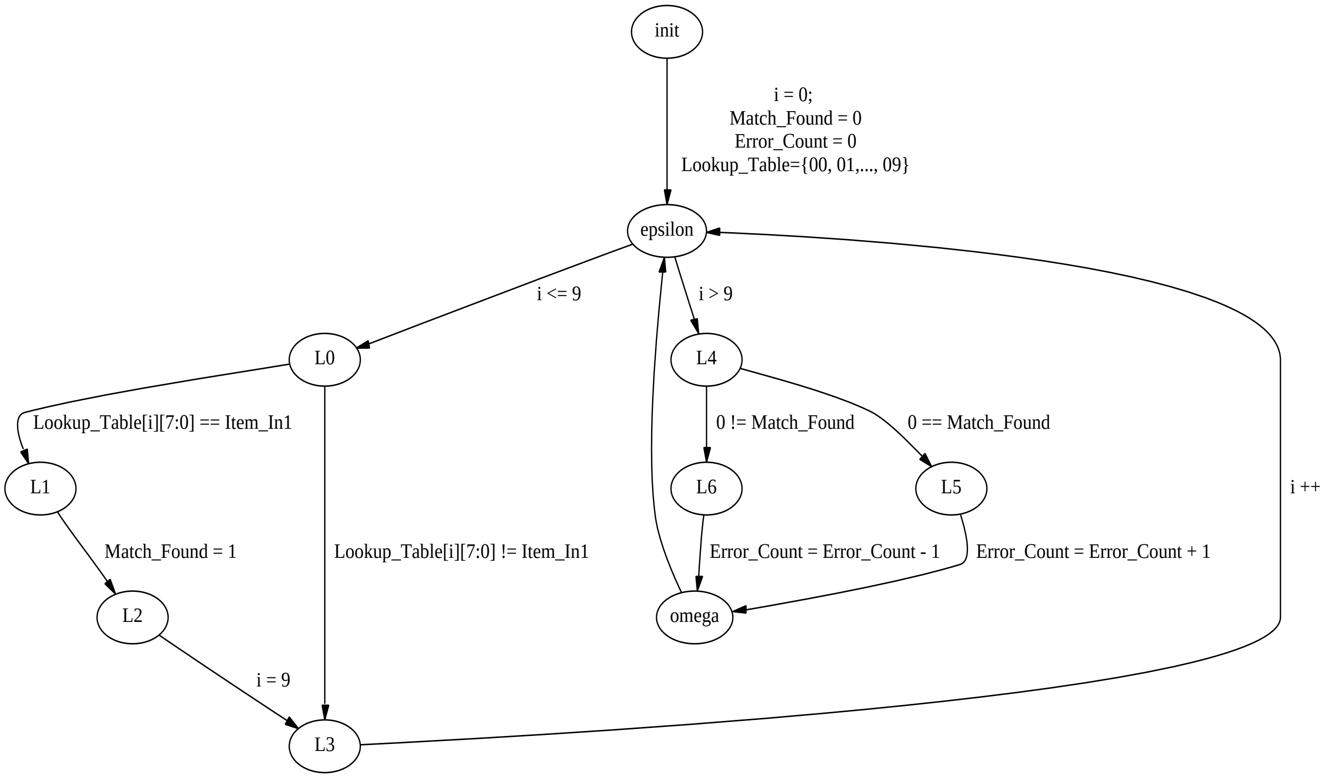

Figure 15: CFD of the Search Program

We want to prove that "the input $Item\_in = 02$ can find a match in the table such that $Match\_Found = 1$". The correspondent properties are φ_1 : $\mathbf{AF}((Item\_In1 = 2) \wedge (Match\_Found = 1))$ and φ_2 : $\mathbf{AG}((\neg(Item\_In1 = 2)) \rightarrow (Match\_Found = 0))$ meaning that $(Item\_In1 = 2)$ eventually can find the match and the others cannot. These properties could not be verified on the original model due to state space explosion during the verification in VIS [2] on a SUN Enterprise server with 6GB memory after more than 10 hours running. Hence, we have to reduce the model before we can make model checking. First, using COI, we want to remove any variables which are not affecting the variable of interest, i.e., $Match\_Found$. We do not need to consider $Item\_In1$ since it is an input.

The DDD of the search program is shown in Figure 16. Through the analysis of the data

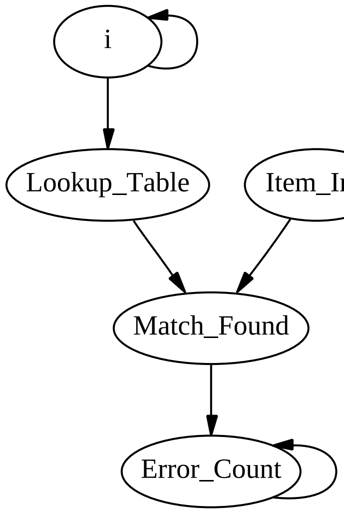

Figure 16: Dependency Diagram of the Search Program

dependency diagram of the search program, we can find that variable $Match\_Found$ depends on variables $Lookup\_Table$, $Item\_In1$, and i, but not on $Error\_Count$. Consequently, we can simply remove variables $Error\_Count$ in the verification of the property, and preserve the soundness and completeness of the property. The corresponding reduced data dependency diagram of the reduced program is shown in Figure 17. The control flow diagram of the search program after the COI reduction is shown in Figure 18.

Although we can remove $Error\_Count$ in the above, however, we cannot remove $Lookup\_Table$ which $Match\_Found$ depends on. In this case, we apply the reduction methods in Section 4.3.

The key node in the CFD is node L_2 because the variable of interest $Match\_Found$ is changed in the incoming edge of this node. The value $\{00, 01, 03, \ldots, 09\}$ of $Lookup\_Table[7:0]$ are deactive because we cannot find a path which leads to L_2 and $ST_{n \rightarrow L_2}(Lookup\_Table[7:0]) = 00, 01, 03, \ldots, 09$ for at least one node n in the path. The value $\{02\}$ of $Lookup\_Table[7:$

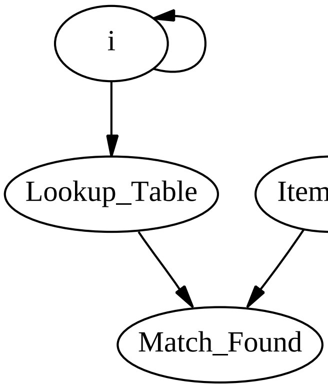

Figure 17: COI Reduced Dependency Diagram of the Search Program

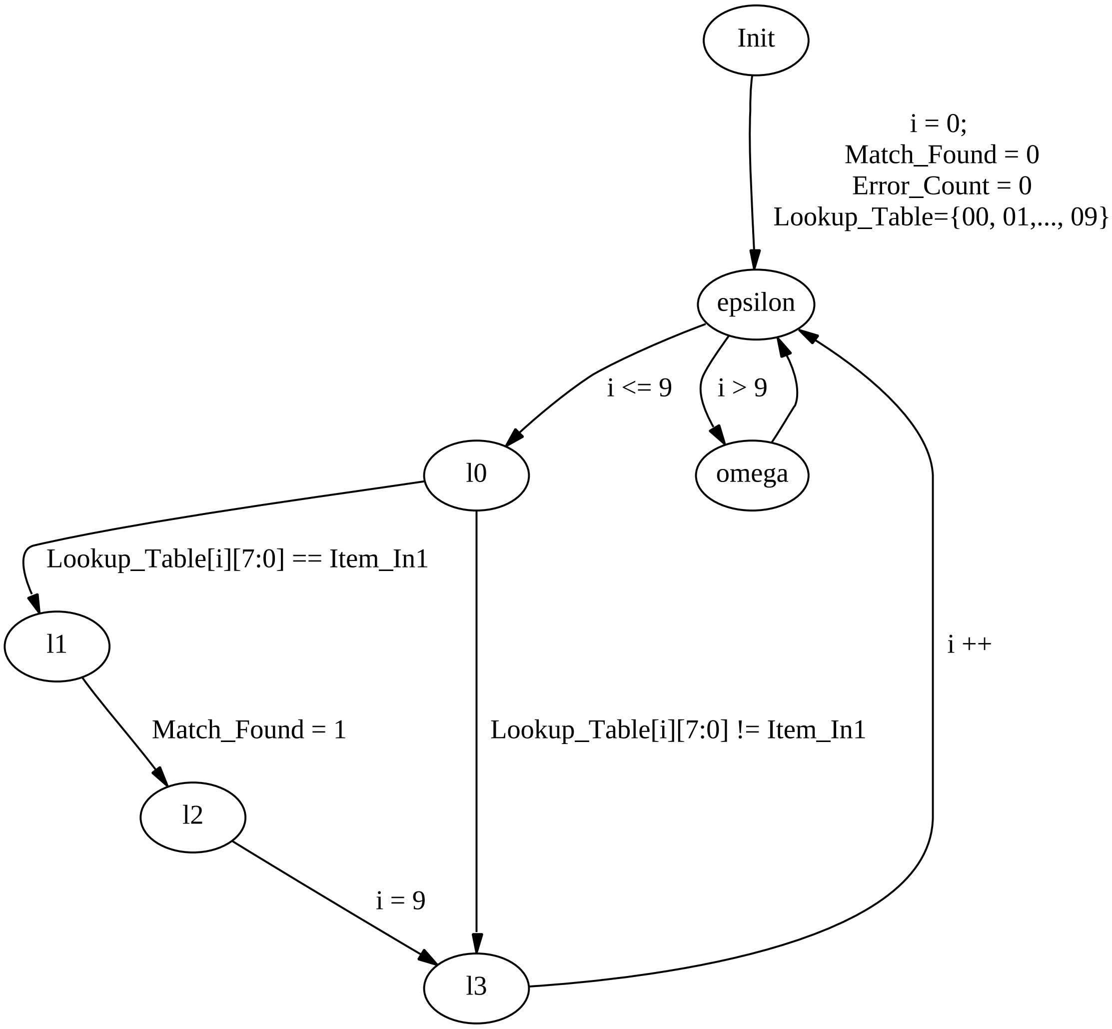

Figure 18: COI Reduced CFD of the Search Program

0] is active since L_2 is reachable in path ϵ, L_0, L_1, L_2 with $ST_{\epsilon \to L_2}(Lookup\_Table[7:0]) = 02$. Consequently, variable $Lookup\_Table[7:0]$ is an abstract variable, where the active value set ACTIVE $(Lookup\_Table[7:0]) = \{02\}$ and the typical deactive value DEACTIVE $(Lookup\_Table[7:0]) = 03$. So, we can use $Lookup\_Table[7:0] = \{02, 03\}$ instead of $\{00, 01, 03, \ldots, 09\}$. After such a reduction, we removed 8 items from the table, and get a reduced program. This reduced program can finish model checking in VIS on the same machine (SUN Enterprise with 6GB memory). The experimental results are shown in Table 1, where the verification could not be completed on the concrete model (actually, the concrete model has 1024 items), but succeed on the reduced model. The table shows the statistic results of CPU time (in seconds) and memory usage (in bytes) of VIS verification on the reduced model, and that of SAT solver on RC_τ, respectively.

	Reduced Model		RC_τ Computation	
Prop.	Verif. CPU	Verif. Mem	SAT CPU	SAT Mem
φ_1	11.3	8.54M	0.02	83.53K
φ_2	11.6	88.51M	0.02	83.53K

Table 1: Verification Result of Sample Properties in VIS

6 Benchmarks

In this section, we will test the proposed reduction methods based on the examples available with the VIS package [2]. Generally, we reduce the model with respect to the variables of interest. Through out the reduction, we will illustrate the advantages and disadvantages of the proposed approach.

6.1 The Bakery Controller

Following is the Verilog code of the bakery controller [2] [2].

```
module bakery_process ;
//Variable declaration
{ L1, L2, L3, L4, L5, L6, L7, L8, L9 } loc;
{0, 1, 2, 3} i, j;
```

[2]The program has been revised to fit our syntax defined in Section 3.1.

```
{0, 1, 2, 3} numberi, numberj;
{0, 1} choosingj;

//Initialization
Init:
      pc=L1; j=0;
      numberi = 0;
      numberj = 0;
      choosingj = 0;

epsilon:

    if (permit) begin
        case (pc)
            L1: begin pc=L2; end
            L2: begin pc=L3; end
            L3: begin pc=L4;end
            L4: begin j=0; pc=L5; end
            L5: begin if (j<2) pc=L6; else pc=L9; end
            L6: begin if (choosingj==1) pc=L6; else pc=L7; end
            L7: begin
                if ((numberj!=0) &&
                    (numberj<numberi || (numberj==numberi && j<i)))
                    pc=L7;
                else
                    pc=L8;
                end
            L8: begin j=j+1; pc=L5; end
            L9: begin pc = $ND(L1, L9); end//$
        endcase
    end
omega:
```

The reachable states of this program are as follows.

```
Reachability analysis results:
FSM depth =                 14
reachable states =         448
```

The property we want to verify is $\mathbf{AG}((pc = L1) \rightarrow (pc = L9))$. In this case, among the values of pc, only $L1$, and $L9$ are active. The other values are deactive, so we will use $L2$ as the typical value of the deactive values. Now, the abstract domain of pc becomes $\{L1, L2, L9\}$. Using the same approach, the abstract domain of j is $\{1, 2, 3\}$, where $\{1\}$ is the typical deactive value and $\{2, 3\}$ are the active values. Since there are no assignment statements of $numberi, numberj$, and $choosingj$, these variables can be treated as wired variables in order not to generate states. Because there are 6 *case* branches with deactive values in the condition, we use non-deterministic variable (*NDTran*) to make the branch, namely the branches from $L2$ to $L3$, $L4$, $L5$, $L6$, $L7$, and $L8$, respectively, are nondeterministic decided by *NDTran*. The new reduced program is as follows.

```
module bakery_process;
//Variable declaration
{ L1, L2, L9 } loc;
{ 1, 2, 3} j;
{0, 1, 2, 3} numberi, numberj;
{0, 1} choosingj;

//Initialization
Init:
      pc=L1; j=0;
      numberi = 0;
      numberj = 0;
      choosingj = 0;

   wire [1:0]   NDTran;
   assign  NDTran = $ND(0,1,2,3,4,5);//$

epsilon:
    if (permit) begin
        case (pc)
          L1: begin pc=L2; end
          L2: begin
              case (NDTran)
                0: pc = L2;
                1: j = 0; pc = L2;
                2: if (j<2) pc=L2; else pc=L9;
                   end
                3: if (choosingj==1) pc=L2;
```

```
                4: if ((numberj!=0) &&
                      (numberj<numberi || (numberj==numberi && j<i)))
                            pc=L2;
                5: j=$ND(1,2,3); pc=L2; //$
             endcase
         L9: begin pc = $ND(L1, L9); end//$
      endcase
  end
omega:
```

The reachable states of the reduced program are as follows, which is only half the size of the original one.

```
Reachability analysis results:
FSM depth =                    6
reachable states =           224
```

Through out the reduction, we can find that we have to examine the values of the variables one by one. Hence, it is required that the domain of the variables be finite. Moreover, to calculate whether a key node is reachable, we need to know exactly the truth or falsity value of the conditions in each branch in the program. Namely, assuming we are calculating whether or not the key node is reachable from the nodes with value e of variable j being active. Consider a test statement "$if(j < i)$" in the program. We need to know the truth value of this test when $j = e$. Hence, we have to examine n possible cases, where n is the size of the domain of variable i. Consequently, in order to check each value of variable j, at this test statement, we need to consider $Size(dom(j)) \times Size(dom(i))$ possible cases which is exponential. In this case, to solve this problem, either we put an environment constraint on i, for example, set i to be e_i, where e_i is an element in $dom(i)$, or we need a more efficient solution.

6.2 The Arbiter

Following is the Verilog code of the arbiter controller[2][3].

```
module arbiter-controller;

//Variable declaration
{A, B, C, X} reg sel, id;
reg ack, pass_token;
```

[3]The program has been revised to fit our syntax defined in Section 3.1.

```
{IDLE, READY, BUSY } reg  state;
reg is_selected;

//Initialization
Init:
state = IDLE;
ack = 0;
pass_token = 1;
id = A;

epsilon:
  is_selected = (sel == id);
  case(state)
    IDLE:
     if (is_selected)
        if (req)
          begin
          state = READY;
          pass_token = 0;
         end
        else
          pass_token = 1;
      else
        pass_token = 0;
    READY:
      begin
      state = BUSY;
      ack = 1;
      end
    BUSY:
      if (!req)
        begin
        state = IDLE;
        ack = 0;
        pass_token = 1;
        end
  endcase
omega:
```

The reachable states of the above model is shown as follows.

```
Reachability analysis results:
FSM depth =                   3
reachable states =           36
```

What we want to prove is that if arbiter controller A receives a request req, then it will eventually acknowledge this request by sending ack, i.e., $\mathbf{AG}((req = 1) \rightarrow \mathbf{AF}(ack = 1))$.

The latches in the program are

- $\{A, B, C, X\}$ reg sel, id;
- reg $ack, pass\_token$;
- $\{IDLE, READY, BUSY\}$ reg $state$;
- reg $is\_selected$;

The variables of interest are req and ack. From the program, we see that the property does not depend on variable "*pass_token*", so this variable can be removed. Among the values of "*sel*", "A" is the active value, and "B, C, X" are the deactive values. Consequently, we can use "$\{A, B\}$ reg *sel*" instead of "$\{A, B, C, X\}$ reg *sel*". The value of "*state*" are all active values, so we have to keep them. The reduced data becomes

- $\{A, B\}$ reg sel;
- reg ack;
- $\{IDLE, READY, BUSY\}$ reg $state$;
- reg $is\_selected$;

The reachable states of the corresponding reduced program is shown as follows.

```
Reachability analysis results:
FSM depth =                   3
reachable states =           10
```

Since the code between ϵ and ω are in a forever loop, the path from a node with value e to the key node may contain many cycles of the loop. In the above example, the value $READY$ of variable $state$ is set at statement $state = READY$. The path from this statement to the key node contains at least two cycles, i.e., "$state = READY$"$\rightarrow \ldots \rightarrow$ "ω"$\rightarrow$ "ϵ" $\rightarrow \ldots \rightarrow$ "*Keynode*". In the worst case, to check if key nodes are not reachable from another node in the CFD, we have to check all the paths between the node and the key nodes using the conditions in Definition 4.6.

7 Related Work and Contributions

Abstract interpretation [8] is a classic static program analysis approach. It has been used intensively in formal verification and model reduction [6, 14, 20]. There are four steps to apply abstract interpretation in model reduction: (1) find abstract domains for the variables in the program with or without the aid of the property, (2) translate the property to an abstract property according to the abstract domains, (3) translate the operations in the program into abstract operations, and then change the program into an abstract program, (4) do formal verification based on the abstract program and the abstract property, then deduce whether or not the concrete program satisfies the concrete property.

The key step in the approach is to define the abstract domains. For example, consider a property containing a proposition "$x = 0$", where x is an integer in the implementation. Since an integer has infinite values, the state space of the implementation potentially is infinite. However, to prove this specific property, we actually do not need to know all the information about the integer x, rather, we only need to know if x is zero or non-zero. In this case, we can put the abstract domain of x as $\{neg, zero, pos\}$ instead of the infinite values, where neg means x is a negative number; $zero$ means x is 0; pos means x is a positive number. Since the value domain of x is changed, the corresponding operations on x has to be changed as well. For example, the abstract operation of '+', '$+_{abs}$', becomes $zero +_{abs} zero = zero$, $pos +_{abs} pos = pos$, $pos +_{abs} neg = \{neg, zero, pos\}$, etc.. We can find that non-determinism is introduced into the operation in the latter case. This type of non-determinism over-approximates the concrete model, and is the source of *false-negative* problem in the verification.

How to get the abstract domains is always a big problem in this approach. In the current existing abstract interpretation based approaches [7], the abstract domain is defined by the user. Namely, the user has to define, firstly the abstract values of one variable; secondly, the mapping between the concrete values and the abstract values. Once the abstract domain has been defined, the abstract operations can be generated [1]. However, in some approaches, the abstract operations have to be given by the user [6].

In practice, real systems are much more complicated than the above example. For different data types, and different operator types, we have to define different kinds of abstract interpretations, for example, for integers signs, for integer even/odd, for integer modulo-k, for sets, and so on [10]. Namely for one variable, there are many abstract interpretation choices. So, it is important that these choices do not conflict. For example, integer y is abstracted as $\{pos, zero, neg\}$, while integer z is abstracted as $\{even, odd\}$. If there is an assignment $y = z$ in the model, it would be a conflicting abstract type, since it is not clear how to covert a $\{pos, zero, neg\}$ type to a $\{even, odd\}$ type. Although tools can help users select different abstract interpretations during the verification [11], a lot of user interaction is unavoidablely involved. Once a correct abstract mapping has been established, translating a concrete program to an abstract program is not difficult. We can replace all the occurrences of the abstracted variable by its abstract type. For example, the definition "$int\ x$" can be replaced by "$\{pos, zero, neg\}\ x$", and "$x + y$" by "$plus\_abs(x, y)$", and so on.

In our approach, although we analyze the value as well, we do not need to define abstract domains. What we did is to simply find all the deactive values and use one typical value in the deactive value set instead of all the deactive values. Our approach is, in some cases, not as efficient as abstract interpretation, but we do not have the false-negative problem and the approach is automatic.

In [22], K. Yorav proposed ways to use the high level description (program text) of a system in order to improve the model checking process by reduction. The approaches are based on program static analysis, and analyze the control flow graph of a program to reveal runtime information of the program, without actually running it. The idea is to use static analysis to build a reduced Kripke structure for the program. The reduced Kripke is equivalent to the original program behavior with respect to **CTL*** without the next state operator. Two methods are presented and compared which use static analysis to create reduced models for programs. The first method, called "path reduction", reduces according to control, and the second, called "dead-variable reduction", reduces according to data. Both methods automatically create a reduced Kripke structure directly out of the syntax of the program (the control flow graph), thus avoiding the need to create the full Kripke structure. The path reduction method excludes possible interleavings between processes. In the reduction, all the paths between two states containing variables of interest are examined. All the states in the path that will not affect the variables of interest are compressed in a single step. The dead-variable reduction excludes some of the successors of a state in which the variables of interest are dead. Our proposed approach has advantages over this approach because: (1) our abstract model is less restrictive since in Yorav's approach the program is required to terminate while ours does not, (2) because we use reflexive and transitive closure as the transitions in the Kripke structure of the state space, the state space in our approach is inherently smaller, (3) the reduction in Yorav's approach is just to compress paths while we do not limit our reduction on paths, so our approach has a better performance in the reduction.

In [19], K. S. Namjoshi et al. proposed a reduction approach which translates a variable with large value domain, for example an integer, into a set of predicates. These predicates are determined by the automated syntactic analysis of the program under verification. The reduction result is a reduced program text, not an explicit-state graph, namely this reduced program can be reduced further more using other explicit-state reduction methods. Our reduction is different from this approach since we will not generate predicates but abstract domains.

Our reductions are also related to works like cone-of-influence reduction [17, 16]. However, our approach is more efficient because we analyze the dependency between the *values* of variables in addition to the dependency between variables, thus the dependency relation is more accurate.

Compared to the above approaches, our approach uses the syntactic and semantic analysis and generates a smaller program against the original one, which preserves the property to be verified, since the concrete model is simulated by the abstracted model with respect to the properties. This model reduction approach can be used together with the compositional verification to verify a large system. However, the approach concentrates on eliminating the redundant data in the data processing, which is a common case in current circuits, but for the moment does not work in some cases such as: (1) a state variable has infinite (or huge) value domain; (2) two parallel modules with shared variables. The compositional verification takes care of the second case. Regarding the first case, the deactive value domain has to be given by the user just like what abstract interpretation did [8], which is not the case in most of the circuits because, after all, the circuits are finite state machines.

8 Conclusion

In this section, we proposed an automatic model reduction approach. The approach uses syntactic analysis and generates a smaller program compared to the original one. It preserves the properties to be verified, since the concrete model is simulated by the abstracted model with respect to the properties. This model reduction approach can be used together with other state space explosion solution: compositional verification, where the size of single modules of a large system is beyond the capability of model checking.

In our approach, the solution is based on the syntactic analysis of the program source code. Compared to the on-the-fly explicit state space reduction during the generation of the state space, source code reduction can be combined with other model checking or reduction tools to improve its reduction efficiency even more. Moreover, since the size of the source code is usually much less than that of its state space or other intermediate forms, this approach uses less resources (CPU time and memory space). Another advantage is that the approach can work directly on hardware description languages such as Verilog via the syntactic and semantic analysis of Verilog code and this can be a feature of a Verilog compiler. However, in the worst case, the procedure examining the value domain of the variables is exponential with respect to the size of variables and the value domain of the variables in the reachability condition (RC) because it has to traverse all the possible values of the variables in RC. A SAT solver is used to find the true value assignment throughout the reduction. The complexity of the reduction depends on that of the CFD traverse and the SAT solver. One of our future research directions is to explore an efficient SAT decision procedure to improve the efficiency.

The main contributions of this work are summarized as follows.

- We proposed a new model reduction methodology, which is based on the syntactic and semantic analysis of the program. Compared to the on-the-fly COI during the generation of the state space, source code reduction can be combined with other model checking or reduction tools to improve its reduction efficiency even more. On the other hand, since normally the size of the source code is much less than that of its state space or other intermediate forms, this approach use less resources with respect to the CPU time and memory space through out the reduction.

- The approach can work on Verilog code directly via the syntactic and semantic analysis of Verilog code. The reduction techniques can be added as new Verilog compiler features. Together with the Verilog environment synthesis and the compositional verification, this approach thus can handle large scale industry designs.

References

[1] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of Computer Aided Verification*, LNCS 1427, pages 319–331, Vancouver, BC, Canada, June/July 1998. Springer Verlag.

[2] R. K. Brayton et al. VIS: A system for verification and synthesis. In *Proceedings of Computer Aided Verification*, volume 1102 of *LNCS*, pages 428–432. Springer Verlag, Rutgers University, NY, USA, July 1996.

[3] Cadence Design Systems. *Technical manual of FormalCheck*, v2.3 edition, 1987-1999.

[4] E. M. Clarke and E. A. Emerson. Design and synthesis of a synchronization skeletons using branching time temporal logic. In *Proceedings of the Logic of Programs Workshop*, Yorktown heights, NY, USA, May 1981.

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

[6] E. M. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, Vol.16(No. 5):1512–1542, Sept 1994.

[7] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, USA, 1977.

[9] M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Communication of the ACM*, 5(7):394–297, July 1962.

[10] M. Dwyer, J. Hatcliff, R. Joehanes, et al. Tool-supported program abstraction for finite-state verification. In *Proceedings of The 23rd International Conference on Software Engineering*, Toronto, Ontario, Canada, May 2001.

[11] M. B. Dwyer and C. S. Pasareanu. Filter-based model checking of partial systems. In *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundation of Software Engineering*, Lake Buena Vista, Florida, USA, November 1998.

[12] N. Francez. *Program Verification.* Addison-Wesley, 1992.

[13] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for satisfiability (SAT) problem: A survey. In *Discrete Mathematics and Theoretical Computer Science: Satisfiability (SAT) Problem*, volume 35, pages 19–152. American Mathematical Society, 1997.

[14] Y. Kesten and A. Pnueli. Modularization and abstraction: the key to practical formal verification. In *Proceedings of the 23rd Int. Symp. Mathematical Foundations of Computer Science*, Brno, Czech Republic, 1998.

[15] S. A. Kripke. Outline of a theory of truth. *Journal of Philosophy*, 72:690–716, 1975.

[16] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Verifying hardware in its software context. In *Proceedings of International Conference on Computer Aided Design*, pages 742–749, San Jose CA, USA, 1997.

[17] R. P. Kurshan. *Computer Aided verification of coordinating processes.* Princeton University Press, 1994.

[18] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[19] K. S. Namjoshi and R. P. Kurshan. Syntactic program transformations for automatic abstraction. In *Proceedings of Computer Aided Verification'00*, volume 1855 of *LNCS*, pages 433–449, Chicago, IL, USA, July 2000. Springer Verlag.

[20] D. A. Schmidt and B. Steffen. Data-flow analysis as model checking of abstract interpretations. In *Proceedings of the 5th Static Analysis Symposium*, Pisa, Italy, September 1998.

[21] Northern Telecom. *Specification of a 4*4 ATM switch*, November 1998.

[22] K. Yorav. *Exploiting syntactic structure for automatic verification.* PhD thesis, Israel institute of technology, June 2000.

[23] H. Zhang. SATO: An efficient propositional prover. In *Proceeding of International Conference on Automated Deduction 97*, volume 1249 of *LNAI*, pages 272–275, Berlin, German, July 1997.

[24] H. Zhang and M. E. Stickel. Implementing the Davis-Putnam algorithm by tries. Technical report, University of Iowa, Iowa City, 1994.