Model Checking and Abstractions

A Seminar on Model Checking

by

Ramesh Kumar
rkumar@mpi-sb.mpg.de

Guide: Prof. Wolfgang J. Paul
Mark Hillebrand

May 5, 2003

1 Introduction

This report is based on the the paper *Model Checking and Abstraction* by E. M. Clarke, O. Grumberg and D. E. Long. Model checking is a hot segment of formal verification, widely applied in verifying the systems (both hardware and software).

1.1 Model Checking

Model checking is an automatic technique for verifying finite-state reactive systems such as sequential circuit designs and communication protocols. Specifications are expressed in a propositional temporal logic and the reactive system is modeled as a state-transition graph. The global state-transition graph of the concurrent system can be viewed as a *finite Kripke structure*. An efficient search procedure is used to determine automatically if the specifications are satisfied by the state-transition graph. Typically the user provides a model of the system and the specification to be checked. The model checker will either terminate with the answer true, indicating that the model satisfies the specification or give a counterexample execution that shows why the formula is not satisfied. The counterexamples are particularly important in finding subtle errors in complex reactive systems. The technical challenge in model checking is devising algorithms and data structures that allow us to handle large search space.

The first model checking algorithm for propositional branching time temporal logic CTL was presented at the 1983 POPL conference [2]. In the span of 20 years, several tools have been developed and different techniques have been devised to increase the performance of model checkers. There are basically two approaches used for model checking:

1. **Explicit state based approach**. This approach is of wide applicability, since a large class of concurrent programming problems have finite-state solutions and interesting properties of many such problems can be specified in propositional temporal logic. The algorithm, which is called *model checker* in [2] has complexity linear in both the size of the Kripke structure (model of the system) and the size of the specification. When the number of global states is not excessive (i.e. not more than a few thousand), this technique may provide useful approach in verifying the system. The approach suffers from the *state explosion problem* and hardware systems which are typically very large, can not be handled using this approach.

2. **BDD-based Symbolic Model-checking Approach**. The idea behind symbolic model checking is to use a more efficient "symbolic" representation for the sets of the states of the Kripke structure. In this approach the model checking algorithm uses Binary Decision Diagrams (A Binary Decision Diagram (BDD) is a directed acyclic graph (DAG) representation of a boolean function where equivalent boolean sub-expressions are uniquely represented) as its internal representation, in order to avoid explicitly enumerating states of the model (we call this symbolic model checking). Due to this property, a BDD can be exponentially more compact than its corresponding truth table representation. This approach can handle medium size designs effectively (Refinement of the BDD-based method techniques is able to verify up to 10^{100} states) but for larger systems, it also faces the *state explosion problem*.

1.1.1 Possible Solution

We need possible solutions to handle the state explosion problem. The idea is to replace the system model by a smaller one (less states and transitions) that still preserves properties of interest. Some possible solutions are:

1. Modular Verification.

2. Symmetry.

3. Abstraction.

These methods can also be applied to infinite state systems.

The report is organized as follows: In section 2 we define transition systems and notion of abstraction. In section 3 we write on constructing an approximated abstract transition system directly from the program. In Section 4, we relate abstraction techniques to the temporal logic ($\forall$CTL*) that is used for specifications and Section 5 describes the verification of several systems via a variety of abstractions.

2 Abstractions

Abstraction is considered as one of the most useful ways to fight the state explosion problem. In [1] it is shown that by combining model checking with abstraction, it is possible to verify larger systems. In one example, a pipelined ALU circuit with 64 registers, each 64 bits wide and having more than 10^{1300} reachable states has been verified using this technique. The goal of this technique is to check the truth value of a formula in the abstract system, and conclude that it has same truth value in the original system. We will discuss that the approach is *conservative* if we restrict formulas to a subset of the logic called $\forall$CTL* in which only $\forall$ path quantifier is allowed. Further, abstractions should be constructed directly from the program. In this report, we are going to discuss about a particular kind of abstraction, called *data abstraction*.

2.1 Data Abstraction

Data abstraction is an abstraction technique, which abstracts data information while still enabling to partially check properties referring to data. Given a program P with variables $x_1,\ldots,x_n$, each over domain D, the *concrete model* of P is defined over states $(d_1,\ldots,d_n) \in D \times \ldots \times D$. Abstractions will be formed by choosing a surjective abstraction mapping h : $D \rightarrow \hat{D}$, which maps D onto a set $\hat{D}$ of abstract values. The abstraction mapping then maps each program state to a corresponding abstract states $(\hat{d}_1,\ldots,\hat{d}_n) \in \hat{D} \times \ldots \times \hat{D}$. We refer it as an *abstract model* of the system.

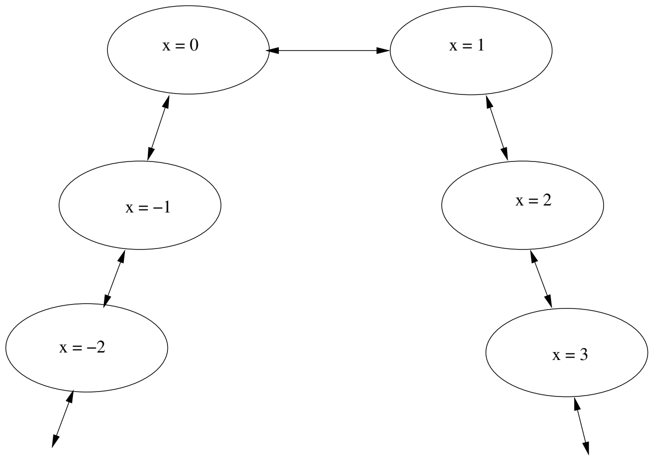
2.1.1 Labeling by abstract atomic propositions

We assume that the states of the concrete model M of P are labeled by *abstract atomic propositions* of the form $(x^A = \hat{d})$ for $\hat{d} \in \hat{D}$ (x^A means that we refer to the abstract value

of x) Then for a state s = $(d_1,\ldots,d_n)$, labeling function is defined as
L(s) = $\{(x_i^A = \hat{d}_i) |$ h(d_i) = $\hat{d}_i$ $1 \leq i \leq$ n$\}$. Given a model M, domain $\hat{D}$ and abstraction mapping h : $D \rightarrow \hat{D}$, h(($d_1,\ldots,d_n$)) = (h(d_1),...,h(d_n)), states s_1, s_2 in S are said to be equivalent ($s_1 \sim s_2$) iff h(s_1) = h(s_2). An abstract state $(\hat{d_1},\ldots,\hat{d_n})$ represents the *equivalence class* of states $(d_1,\ldots,d_n)$ such that h(($d_1,\ldots,d_n$)) = $(\hat{d_1},\ldots,\hat{d_n})$. We look an example which demonstrates the application of abstraction technique to handle infinte state system.

Example

Given a program P with variables x over the integer $\mathbb{Z}$; Initially x may be either 0 or 1. At any step x may non-deterministically either decrease or increase by 1. So, the concrete model can be drawn as

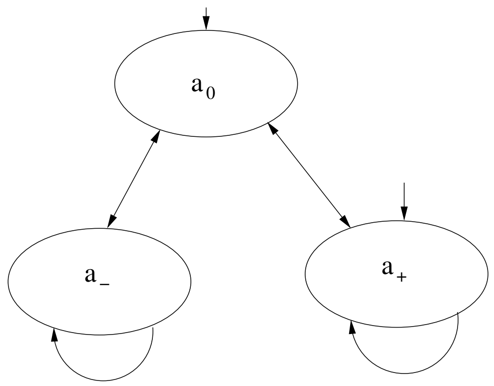


Define the abstraction mapping as

- $\hat{D}$ = { a_-, a_0, a_+ }
- h_1: $\mathbb{Z} \rightarrow \{a_+, a_0, a_-\}$ as

$$h_1(d) = \left\{ \begin{array}{l} a_+ \text{ if d} > 0, \\ a_0 \text{ if d} = 0, \\ a_- \text{ if d} < 0 \end{array} \right\}.$$

The abstracted model can be drawn as

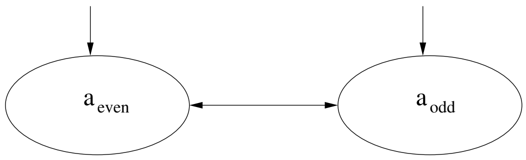


We can re-define the abstraction mapping as

- $\hat{D} = \{ a_{even}, a_{odd} \}$
- h_2: $\mathbb{Z} \rightarrow \{ a_{even}, a_{odd} \}$ as

$$h_2(d) = \left\{ \begin{array}{l} a_{even} \text{ if even(|d|)} \\ a_{odd} \text{ if odd(|d|)} \end{array} \right\}.$$

The corresponding abstracted model can be drawn as :



So, abstraction can be used to replace the system by a smaller one, which preserves properties of interest. Now, we formally define abstractions over the transition systems:

2.1.2 Transition Systems and Abstractions

We consider programs with a finite set of variables $v_1,\ldots,v_n$. If each variable ranges over a non-empty set D of possible values, then set of all possible program states is D^n. We represent the possible behaviors of the program with a set of transitions between states. This notion is formalized as:

Definition 1 (Kripke Structure). A transition system over D^n is a triple M = (S, I, R) where
(1) S = D^n is a set of states;
(2) I $\subseteq$ S is a set of initial states; and
(3) R $\subseteq$ S $\times$ S is a transition relation.

The abstract transition system($\hat{M}$) over the abstract domain($\hat{D}$) is defined as:

Definition 2. Let h be an abstraction mapping and $\hat{M}$ be a transition system over $\hat{D}^n$. We say that $\hat{M}$ approximates M (denoted M $\sqsubseteq_h$ $\hat{M}$) when for all $\hat{d}$, $\hat{d}_1$, $\hat{d}_2$ $\in$ $\hat{D}$ the following conditions hold:
(1) $\exists$d (h(d) = $\hat{d}$ $\wedge$ I(d)) implies $\hat{I}(\hat{d})$.
(2) $\exists d_1 \exists d_2$ (h(d_1) = $\hat{d}_1$ $\wedge$ h(d_2) = $\hat{d}_2$ $\wedge$ R(d_1, d_2)) implies $\hat{R}(\hat{d}_1, \hat{d}_2)$.

There is a natural abstract transition system having only those states and transitions required by the above definition. This transition system is called the *minimal transition system* ($\hat{M}_{min}$).

Definition 3. $\hat{M}_{min}$ is the transition system over $\hat{D}^n$ given by :
(1) $\hat{I}_{min}(\hat{d})$ if and only if $\exists$ d (h(d) = $\hat{d}$ $\wedge$ I(d)).
(2) $\hat{R}_{min}(\hat{d}_1, \hat{d}_2)$ if and only if $\exists$ d_1 $\exists$ d_2 (h(d_1) = $\hat{d}_1$ $\wedge$ h(d_2) = $\hat{d}_2$ $\wedge$ R(d_1, d_2)).

From the definitions 2 and 3, we can conclude that M $\sqsubseteq_h$ (approximated by) $\hat{M}_{min}$.

Theorem 1. For any other transition system $\hat{M}$ over $\hat{D}^n$, we see that M $\sqsubseteq_h$ $\hat{M}$ if and only if $\hat{I} \supseteq \hat{I}_{min}$ and $\hat{R} \supseteq \hat{R}_{min}$.

Proof. "$\Rightarrow$" Assume M $\sqsubseteq_h$ (approximated by) $\hat{M}$
claim: $\hat{I} \supseteq \hat{I}_{min}$ and $\hat{R} \supseteq \hat{R}_{min}$. It is sufficient to prove that

- $\forall \hat{d}\ \hat{d} \in \hat{I}_{min} \Rightarrow \hat{d} \in \hat{I}$
- $\forall \hat{d}_1, \forall \hat{d}_2\ (\hat{d}_1, \hat{d}_2) \in \hat{R}_{min} \Rightarrow (\hat{d}_1, \hat{d}_2) \in \hat{R}$

Consider any element $\hat{d} \in \hat{I}_{min}$. With the definition of $\hat{M}_{min}$, we have

$$\exists d(h(d) = \hat{d} \wedge I(d)) \Rightarrow \hat{I}(\hat{d}) \text{ (from Definition 1)}$$

Hence $\forall \hat{d}\ \hat{d} \in \hat{I}_{min} \Rightarrow \hat{d} \in \hat{I}$ proved.
A similar argument can be given for $\forall \hat{d}_1, \forall \hat{d}_2\ (\hat{d}_1, \hat{d}_2) \in \hat{R}_{min} \Rightarrow (\hat{d}_1, \hat{d}_2) \in \hat{R}$.

"$\Leftarrow$" Assume $\hat{I}_{min} \subseteq \hat{I}$ and $\hat{R}_{min} \subseteq \hat{R}$
claim: M $\sqsubseteq_h$ $\hat{M}$ or

- $\forall \hat{d}$ [$\exists$d (h(d) = $\hat{(d)}$ $\wedge$ I(d))] $\Rightarrow$ $\hat{I}(\hat{d})$
- $\forall \hat{d}_1, \hat{d}_2$ [$\exists d_1\ \exists d_2$ h(d_1) = $\hat{d}_1$ $\wedge$ h(d_2) = $\hat{d}_2$ $\wedge$ R(d_1, d_2)] $\Rightarrow$ $\hat{R}(\hat{d}_1, \hat{d}_2)$

Assume $\hat{d} \in \hat{D}$ and d $\in D$ such that h(d) = $\hat{d}$ $\wedge$ I(d). From the definition of $\hat{M}_{min}$

$$\begin{aligned} &\Rightarrow\ \hat{I}_{min}(\hat{d}) \text{ or } \hat{d} \in \hat{I}_{min} \\ &\Rightarrow\ \hat{d} \in \hat{I} \text{ (from assumption)} \end{aligned}$$

hence $\forall \hat{d}$ [$\exists$ d (h(d) = $\hat{(d)}$ $\wedge$ I(d))] $\Rightarrow$ $\hat{I}(\hat{d})$ proved.
Similar argument for transition relation also.

Thus $\hat{M}_{min}$ is the most accurate approximation to M that is consistent with h.

3 Producing Abstract Models

A program in a finite-state language can be transformed into *relational expressions* I and R that can be evaluated to obtain the initial states I and the transition relation R of the transition system M represented by the program. These relational expressions are simply formulas in first-order predicate logic that will built up from a set of *primitive relations* for the basic operators and constants in the language. It is described in detail in [1]. In the next section, we will show how to manipulate I and R to obtain approximations of M.

3.1 Computing Approximations

Building $\hat{M}_{min}$ requires evaluating two relational products, both involving existential quantification over the elements of D. For conciseness, this kind of existential abstraction is denoted by the operator [.].
If ϕ depends on the free variables $x_1,\ldots,x_m$. Then we define

$$[\phi](\hat{x}_1,\ldots,\hat{x}_m) = \exists\, x_1 \ldots \exists\, x_m(h(x_1) = \hat{x}_1 \;\wedge \ldots \wedge h(x_m) = \hat{x}_m \;\wedge \phi(x_1,\ldots x_m)).$$

From the definition of $\hat{M}_{min}$, if I and R are the quantified boolean formulas representing I and R, then

$$\hat{I}_{min} = [I] \text{ and } \hat{R}_{min} = [R].$$

Since actually evaluating $\hat{I}_{min}$ and $\hat{R}_{min}$ can be computationally complex, we now show how to obtain formuals $\hat{I}_{app}$ and $\hat{R}_{app}$ describing an approximation $\hat{M}_{app}$ to M.

3.1.1 Evaluating $[I]$ and $[R]$ directly

Applying the [.] operator to complex formulas can be computationally expensive. So, we define another transformation T on quantified boolean formulas ϕ. The idea of T is to simplify the formula to which [.] is applied.
Assumption: ϕ is given in negation normal form i.e. negations are applied only to primitive relations.

1. If P is primitive relation, then $T(\mathrm{P}(x_1,\ldots,x_m)) = [\mathrm{P}](\hat{x}_1,\ldots\hat{x}_m)$ and $T(\neg\,\mathrm{P}(x_1,\ldots,x_m)) = [\neg\,\mathrm{P}](\hat{x}_1,\ldots\hat{x}_m)$.
2. $T(\phi_1 \;\wedge\; \phi_2\;) = T(\phi_1) \wedge T(\phi_2)$
3. $T(\phi_1 \;\vee\; \phi_2\;) = T(\phi_1) \vee T(\phi_2)$
4. $T(\forall \mathrm{x}\; \phi) = \forall \hat{x}\;\; T(\phi)$.
5. $T(\exists \mathrm{x}\; \phi) = \exists \hat{x}\;\; T(\phi)$.

Building $\hat{M}_{app}$: $\hat{M}_{app}$ can be produced by evaluating the formulas *T(I)* and *T(R)*. However, to be able to use $\hat{M}_{app}$ for verification purposes, we must ensure that we have not omitted any behavior of the abstract system. To do this we examine the relationship between $[\phi]$ and $T(\phi)$.

Theorem 2. $[\phi]$ implies $T(\phi)$. In particular, $[I]$ implies *T(I)* and $[R]$ implies *T(R)*.

Proof. We prove the theorem by induction on the structure of the formula ϕ.

1. If $\phi = \mathrm{P}(x_1,\ldots,x_m)$ or $\phi = \neg\,\mathrm{P}(x_1,\ldots,x_m)$ where P is a primitive relation, then $[\phi] = T(\phi)$ and the lemma holds.

2. Let $\phi(x_1,\ldots,x_m) = \phi_1 \ \wedge \ \phi_2$. ($\phi_1$ and ϕ_2 are assumed to have the same parameter lists as ϕ).
   Then, $\phi_1 \ \wedge \ \phi_2$ is identical to the formula
   $$\exists x_1, \ldots, \exists x_m (\bigwedge_i h(x_i) = \hat{x}_i \ \wedge \ \phi_1 \ \wedge \ \phi_2).$$
   This formula implies
   $$\exists x_1, \ldots, \exists x_m (\bigwedge_i h(x_i) = \hat{x}_i \ \wedge \ \phi_1) \wedge \exists x_1, \ldots, \exists x_m (\bigwedge_i h(x_i) = \hat{x}_i \ \wedge \ \phi_2),$$
   which is exactly $[\phi_1] \wedge [\phi_2]$.
   Now, $T(\phi_1 \ \wedge \ \phi_2) = T(\phi_1) \wedge T(\phi_2)$, and by the induction hypotheses, we have $[\phi_1]$ implies $T(\phi_1)$ and $[\phi_2]$ implies $T(\phi_2)$. Hence, $[\phi_1] \wedge [\phi_2]$ implies $T(\phi_1) \wedge T(\phi_2)$, and so $[\phi_1 \ \wedge \ \phi_2]$ implies $T(\phi_1 \ \wedge \ \phi_2)$.

3. $\phi(x_1,\ldots,x_m) = \phi_1 \ \vee \ \phi_2$; same as above.

4. Let $\phi(x_1,\ldots,x_m) = \forall$ x ϕ_1. Then [$\forall$ x ϕ_1] is
   $$\exists x_1, \ldots, \exists x_m (\bigwedge_i h(x_i) = \hat{x}_i \ \wedge \ \forall x \phi_1(x, x_1, \ldots, x_m)).$$
   We can assume w.l.o.g. that the bound variable x is different from the x_i and $\hat{x}_i$, so the above formula is equivalent to
   $$\exists x_1, \ldots, \exists x_m \forall x (\bigwedge_i h(x_i) = \hat{x}_i \ \wedge \ \phi_1(x, x_1, \ldots, x_m)).$$
   This implies
   $$\forall x \exists x_1, \ldots, \exists x_m (\bigwedge_i h(x_i) = \hat{x}_i \ \wedge \ \phi_1(x, x_1, \ldots, x_m)).$$
   Since h is surjection, for every abstract element in $\hat{D}$, there is some element in D that maps onto it.
   So, the above formula implies
   $$\forall \hat{x} \exists x \, [\exists x_1, \ldots, \exists x_m (h(x) = \hat{x} \ \wedge \textstyle\bigwedge_i h(x_i) = \hat{x}_i \ \wedge \ \phi_1(x, x_1, \ldots, x_m))].$$
   $$= \forall \hat{x} [\phi_1].$$
   Now, by induction hypotheses, $[\phi_1]$ implies $T(\phi_1)$ and so $\forall \hat{x}[\phi_1]$ implies $\forall \hat{x} \ T(\phi_1)$. This latter is equal to $T(\forall$x $\phi_1)$.

5. The case $\phi(x_1,\ldots,x_m) = \exists$ x ϕ_1 is similar to the previous case.

Theorem 3. Let $\hat{M}_{app}$ be the transition system obtained by evaluating *T(I)* and *T(R)*. Then, M $\sqsubseteq_h \ \hat{M}_{app}$.

Proof. We know that M $\sqsubseteq_h$ $\hat{M_{min}}$. By the previous theorem $\hat{I}_{min} \subseteq \hat{I}_{app}$ and $\hat{R}_{min} \subseteq \hat{R}_{app}$. Applying theorem 1, we get M $\sqsubseteq_h$ $\hat{M}_{app}$.

Definition 4. A path in M is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that for every i $\in \mathbb{N}$, R(s_i, s_{i+1}).
If $\pi = s_0 s_1 s_2 \ldots$ is a sequence of states from D, we denote the sequence h(s_0)h(s_1)h(s_2)... by h(π).

Lemma 1. Assume M $\sqsubseteq_h$ $\hat{M}$. If π is a path in M, then h(π) is a path in $\hat{M}$.

4 Relating abstraction techniques to temporal logic

The logics that will be used for specifying properties will be subsets of the logic CTL*. CTL* is a powerful temporal logic that can express both branching time and linear time properties. The $\forall$CTL* is a restricted subset of CTL* in which the only path quantifiers allowed is the universal path quantifier ("$\forall$"). The syntax and semantics of these logics is detailed in [1] and [2].

4.1 Defining formulas on abstract transition system ($\hat{M}$):

We say that the atomic formula $\hat{v_i} = \hat{d}$ is true at state $(\hat{e_1}, \ldots \hat{e_n})$ if and only if $\hat{e_i} = \hat{d}$.

We now define a translation C between formulas describing the abstract transition system $\hat{M}$ and formulas describing M. The goal is to check a formula ψ on $\hat{M}$ and infer that the corresponding formula C(ψ) holds for M.

Definition 5.

1. C(true) = true. C(false) = false.
   $C(\hat{v_i} = \hat{d}) = \bigvee \{v_i = \mathrm{d} \mid \mathrm{h(d)} = \hat{d}\}$
   $C(\hat{v_i} \neq \hat{d}) = \neg\, C(\hat{v_i} = \hat{d})$.

2. If ϕ and ψ are state formulas, then $C(\phi \wedge \psi) = C(\phi) \wedge C(\psi)$ and $C(\phi \vee \psi) = C(\phi) \vee C(\psi)$.

3. If ψ is a path formula, then $C(\forall(\psi)) = \forall(C(\psi))$ and $C(\exists(\psi)) = \exists(C(\psi))$.

4. If ϕ is a path formula that is also a state formula, then C(ϕ) is given by the above rules.

5. If ϕ and ψ are path formulas, then $C(\phi \wedge \psi) = C(\phi) \wedge C(\psi)$ and $C(\phi \vee \psi) = C(\phi) \vee C(\psi)$.

6. If ϕ and ψ are path formulas, then

   (a) $C(X(\phi)) = XC(\phi)$.

   (b) $C(\phi \mathrm{~U~} \psi) = C(\phi) \mathrm{~U~} C(\psi)$.

   (c) $C(\phi \mathrm{~V~} \psi) = C(\phi) \mathrm{~V~} C(\psi)$.

Now we will prove the main preservation theorem: formulas that hold at the abstract level also hold for the concrete system.

Theorem 4. Assume M is simulated by $\hat{M}$(M $\sqsubseteq_h$ $\hat{M}$). Then:

1. for all ∀CTL* state formulas ϕ describing $\hat{M}$ and every state s of M, h(s) $\models$ ϕ implies s $\models$ C(ϕ); and

2. for all ∀CTL* path formulas ϕ describing $\hat{M}$ and every path π in M, h(π) $\models$ ϕ implies π $\models$ C(ϕ).

Proof. The proof proceeds by induction on the structure of the formula.
Let s = (e_1,...,e_n) and h(s) = ($\hat{e_1}$,...,$\hat{e_n}$).

1. If ϕ = true or ψ = false, the result is trivial.
   Case: $\phi = (\hat{v_i} = \hat{d})$ then h(s) $\models$ ϕ if and only if $\hat{e}_i = \hat{d}$.
   So, s $\models$ $v_i = e_i$, since we have h(e_i) = $\hat{d}$.
   Therefore s satisfies the following
   $$s \models \bigvee\{v_i = d \mid h(d) = \hat{d}\}$$
   and this is C($\hat{v_i} = \hat{d}$) and so s $\models$ C($\hat{v_i} = \hat{d}$).
   Case: $\phi = (\hat{v_i} \neq \hat{d})$.
   We have h(s) $\models$ ϕ iff h(e_i) $\neq$ $\hat{d}$.
   The state s satisfies the concrete atomic propositions v_i $\neq$ d for all d different from e_i. With this we get the truth of the following conjunction:
   $$s \models \bigwedge\{v_i \neq d \mid d \neq e_i\}$$
   Because of h(e_i) $\neq$ $\hat{d}$, we have that every d with h(d) = $\hat{d}$ also satisfies d $\neq$ e_i. Thus,
   $$\{d \mid h(d) = \hat{d}\} \subseteq \{d \mid d \neq e_i\}.$$
   This infers that the conjunction also holds for
   $$\begin{aligned} s &\models \bigwedge\{v_i \neq d \mid h(d) = \hat{d}\} \\ &\models \neg\bigvee\{v_i = d \mid h(d) = \hat{d}\} \\ &\models \neg C(\hat{v_i} = \hat{d}) = C(\hat{v_i} \neq \hat{d}) \end{aligned}$$

2. h(s) $\models$ $\phi \wedge \psi$ implies h(s) $\models$ ϕ and h(s) $\models$ ψ.
   From the induction hypotheses, we have s $\models$ C(ϕ) and s $\models$ C(ψ),
   So, s $\models$ C(ϕ) $\wedge$ C(ψ) and so from definition, s $\models$ C($\phi \wedge \psi$).
   Similarly for $\phi \vee \psi$.

3. Assume h(s) $\models$ ∀(ϕ). Let π is any path in M from s. Then by the prev. lemma h(π) is a path in $\hat{M}$ from h(s).
   Since h(s) $\models$ ∀(ϕ) implies h(π) $\models$ ϕ.
   Then by induction hypotheses π $\models$ C(ϕ). Since it holds for any path π.
   So, s $\models$ ∀(C(ϕ)) and hence from definition, s $\models$ C(∀(ϕ)).

4. Assume ϕ is a state formula and $h(\pi) \models \phi$. If the initial state of π is s, then the initial state of $h(\pi)$ is h(s). This implies $h(s) \models \phi$, and then by the induction hypothesis, $s \models C(\phi)$. Hence $\pi \models C(\phi)$.

5. $h(\pi) \models \phi \wedge \psi$ implies $h(\pi) \models \phi$ and $h(\pi) \models \psi$. From induction hypotheses, we have $\pi \models C(\phi)$ and $\pi \models C(\psi)$, so $\pi \models C(\phi \wedge \psi)$.

6. (a) $h(\pi) \models X\phi$ implies $(h(\pi))^1 \models \phi$. Now $(h(\pi))^1 = h(\pi^1)$, and so the induction hypothesis implies $\pi^1 \models C(\phi)$. Thus, $\pi \models X(C(\phi))$, and so $\pi \models C(X\phi)$.

   (b) $h(\pi) \models \phi \, U \, \psi$, then there exists $n \in \mathbb{N}$ such that $(h(\pi))^n \models \psi$ and for all $i < n$, $h((\pi))^i \models \phi$. This implies $h(\pi^n) \models \psi$ and $h(\pi^i) \models \phi$ for all $i < n$. Using the induction hypothesis, we find $\pi \models C(\phi \, U \, \psi)$.

   (c) The case $h(\pi) \models \phi \, V \, \psi$ is similar to previous one.

Corollary 1. Assume $M \sqsubseteq_h \hat{M}$ and let ϕ be a $\forall$ CTL* formula describing $\hat{M}$. Then $\hat{M} \models \phi$ implies $M \models C(\phi)$.

This result talks about preserving the truth of formulas that describe behavior that should hold on *all paths* from a state.

5 Application

There are several different abstractions that are useful for reasoning about programs. These abstractions include

1. congruence modulo an integer for dealing with arithmetic operations.

2. single bit abstractions, for dealing with bitwise logical operators

3. product abstractions, for combining abstractions as above.

4. symbolic abstractions. This is a powerful type of abstractions that allows us to verify an entire class of formulas simultaneously.

Now, we will illustrate the application of abstraction on a 32 bit by 32 bit unsigned multiplier (given below). The program has inputs req, in1 and in2. The program uses school method of multiplication and output gives either 16 bit of result of multiplication, or overflow is one if the multiplication overflowed. The specification we use for the multiplier is a series of the formulas of the following form

$$\begin{aligned} &AG(waiting \wedge req \wedge (in1modm = i) \wedge (in2modm = j) \\ &\quad \rightarrow A[\neg ackUack \wedge (ovf \vee (outputmodm = ijmodm))]). \end{aligned}$$

Here, i and j ranges from 0 through m - 1, hence we have to check $O(m^2)$ formulas.

A 32-bit multiplier(Program):

```
input in1, in2 : 32
input req : 1
output factor1, factor2, output : 32 := 0
output ovf, ack : 1 := 0
procedure waitfor(e)
   while ¬e
      wait
   endwhile
endproc
loop
  1: waitfor(req)
   factor1 := in1
   factor2 := in2
   output := 0
   ovf := 0
   wait
   loop
      if (factor1 = 0) ∨ (ovf = 1)
         break
      endif
      if lsb(factor1) = 1
         (ovf, output) := (output : 33) + factor2
      endif
      factor1 := factor1 >> 1.
      wait
      if (factor1 = 0) ∨ (ovf = 1)
         break
      endif
      (ovf, factor2) := (factor2 : 33) << 1
      wait
   endloop
   ack := 1
   wait
   waitfor(¬req)
   ack := 0
endloop
```

Useful Abstractions:

1. *Congruence modulo an Integer:* For verifying programs involving arithmetic operations. Defined as:

$$h(i) = i \bmod m.$$

The abstraction is motivated by following properties:

$$((i \bmod m) + (j \bmod m)) \bmod m = i + j (\bmod m)$$
$$((i \bmod m) - (j \bmod m)) \bmod m = i - j (\bmod m)$$
$$((i \bmod m)(j \bmod m)) \bmod m = ij (\bmod m).$$

Chinese-remainder theorem: Let $m_1,\ldots,m_n$ be positive integers that are pairwise relative prime. Define $m = m_1 m_2 \ldots m_n$, and let b, $i_1,\ldots,i_n$ be integers. Then there is a unique integer i such that b, $i < b + m$ and $i = ij$ (mod mj) for $1 \leq j \leq n$.
These properties implies that it is enough to verify our multiplier by choosing $m = 5$, 7, 9, 11, 13, 16, 17, 19, 23. These numbers are relatively prime and their product is $5 \times 7 \times 9 \times 11 \times 13 \times 16 \times 17 \times 19 \times 23 > 2^{32}$ i.e. sufficient to cover all 2^{32} possible values of output.

2. *Representation by logarithm:* The abstraction function is defined as

$$h(i) = \lg i = \lceil log_2(i+1) \rceil$$

This function has property that $\lg i = 0$ if $i = 0$ and for $i > 0$, $\lg i$ is the smallest number of bits needed to write i in binary.
This abstraction can be used to check the overflow in the "multiplier" example by using the formulas as specification:

$$AG(waiting \wedge req \wedge (\lg in1 + \lg in2 \leq 32) \rightarrow \forall(\neg ack\ U\ ack \wedge \neg ovf))$$

$$AG(waiting \wedge req \wedge (\lg in1 + \lg in2 \geq 34) \rightarrow \forall(\neg ack\ U\ ack \wedge ovf))$$

The abstract system is obtained by representing all 32 bit variables in the program by their logarithms.

So with method 1 & 2 we have verified that the multiplier operates correctly for inputs with $\lg i + \lg j \neq 33$.
In this report, we described a simple but powerful method for using abstraction to simplify the problem of model checking.

References

[1] E. M. Clarke, O. Grumberg and D. Long. Model Checking and Abstraction. *TOPLAS* 1994.

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