

Compositional Abstraction for Concurrent Programs

Junyan Qian^{1,2}, Baowen Xu²

¹*Department of Computer Science and Technology,
Guilin University of Electronic Technology, Guilin 541004, China*

²*Department of Computer Science and Engineering,
Southeast University, Nanjing 210096, China*

E-mail: qjy2000@gliet.edu.cn, bwxu@seu.edu.cn

Abstract

We present a methodology for automatically constructing an abstraction of concurrent programs against safety specifications based on finite state machine. And then automatically extract an initial abstract model from source code using predicate abstraction and theorem proving. However, the process of extracting a finite model from a program using predicate abstraction can be exponential in the number of predicates used. In order to make model construction effective, our method is to partition the set of candidate predicates into subsets, and construct abstract model independently.

1. Introduction

In industry, ensuring the reliability of software systems is important but extremely difficult due to the test coverage problem. As the number of possible execution sequences for software program is too large or even infinite, exhaustive testing is usually infeasible. For formal verification, we mostly have two methods: theorem proving and model checking. However, theorem proving requires considerable expertise to guide and assist the verification process. In order to overcome the problems mentioned above, formal method to automatic verification, such as *model checking* [5], has been proposed.

Model checking could produce major enhancements in software reliability and robustness. However, software programs typically have huge, sometimes infinite, so state spaces that cannot be model checked directly using conventional model checking methods. By applying mathematical abstraction methods, a reduced model may be extracted from the program, and making model checking feasible.

Abstraction has been widely used to make model checking more efficient for large systems. Our method is based on Predicate Abstraction [10, 11] presented firstly by Graf and Saidi [1], which is a special form of Abstract Interpretation [2], where the abstract domain is constructed using a given set of predicates, i.e. a

potentially unbound data type is abstracted to a finite set of points.

Predicate abstraction has been popularly and widely applied in systematic abstraction of programs in recent years, such as Bandera [6], Java PathFinder [3], SLAM [9], MAGIC [7, 8] and BLAST [4], where the first two focus on Java while the last three all deal with C. S. Bensalem et al. [14, 15] present a compositional method, where the set of concrete variables can be partitioned into a number of small sets of variables, abstracted independently, and then they present a uniform verification method that combines a abstraction, model checking and deductive verification.

2. Preliminary

2.1. Kripke Structure

Model checking program is the automatic process of deciding whether a program \mathcal{C} satisfies a given specification or property ϕ , and should yield a “yes” or “no” answer. There are three basic ingredients for model checking programs.

- A model of the program consists of a description of all its possible behaviors like a transition system.
- A specification language is a formula in a logic interpreted over such structure.
- A methodology to establish whether the program model satisfies the specification.

We represent a program model as a Kripke structure, Simply stated, a Kripke structure is a finite-state automaton whose states are labeled with atomic, primitive “properties” that “hold true” at the states. More formally define as follow.

DEFINITION 1. (Kripke structure) A Kripke structure is a triple $(\Sigma, I, \rightarrow, L)$, where (i) Σ is a set of states, (ii) I is a set of initial state, (iii) $\rightarrow \subseteq \Sigma \times \Sigma$ is the transition relation. We write $s \rightarrow s'$ to denote $(s, s') \in \rightarrow$. (iv) $L: \Sigma \rightarrow 2^{AP}$ associates a set of atomic properties, $L(s) \subseteq AP$, to each $s \in \Sigma$, for some fixed and finite set AP.

DEFINITION 2. (Synchronous product) Given two binary relations $\rightarrow_i \subseteq \Sigma_i \times \Sigma'_i$, for $i=1,2$, we define their synchronous or conjunctive product $\rightarrow_1 \otimes \rightarrow_2 \subseteq (\Sigma_1 \times \Sigma_2) \cup (\Sigma'_1 \times \Sigma'_2)$, by $(s, s') \in \rightarrow_1 \otimes \rightarrow_2$ iff $(s|_{\Sigma_i}, s'|_{\Sigma'_i}) \in \rightarrow_i$, for $i=1,2$, where $s|_{\Sigma_i}$ denotes the restriction of the state s to Σ_i .

2.2. Weakest Preconditions

For a statement $s \in \text{Stmt}$ and a predicate p , let $\mathcal{WP}(s, p)$ denote the weakest precondition [11] of p with respect to a given statement s . $\mathcal{WP}(s, p)$ is defined as the weakest predicate whose truth before s entails the truth of p afterwards. Consider an assignment s of the form $x=e$, where x is a variable and e is an expression. Then the weakest precondition rule says that $\mathcal{WP}(x=e, p)$ is obtained from p by replacing all occurrences of x in p with e , denoted $p[e/x]$. For example, $\mathcal{WP}(x = x+1, x < 5) = (x+1) < 5 = x < 4$. Therefore, $(x < 4)$ is true before $x = x+1$ executes if and only if $(x < 5)$ is true afterwards. Consider an assert s of the form $\text{await}(e)$. Then the weakest precondition rule says that $\mathcal{WP}(\text{await}(e), p)$ is denoted $e \Rightarrow p$. \mathcal{WP} for assignments and asserts is defined as follows:

$$\mathcal{WP}(x=e, p) = p[e/x], \quad \mathcal{WP}(\text{await}(e), p) = e \Rightarrow p$$

Give a statement s , a set of predicates \mathcal{P} , and predicate $p \in \mathcal{P}$, it may be the case that $\mathcal{WP}(s, p)$ is not in \mathcal{P} . For example, suppose $\mathcal{P} = \{(x < 5), (x = 2)\}$. We have seen that $\mathcal{WP}(x = x+1, x < 5) = x < 4$, but the predicate $(x < 4)$ is not in \mathcal{P} . Therefore, we need to use theorem prover to strengthen the weakest precondition to an expression over the predicated in \mathcal{P} . In the example, we can show that $x=2 \Rightarrow x < 4$. Therefore if $(x=2)$ is true before $x = x+1$, then $(x < 5)$ is true afterwards.

2.3. Abstraction

Abstraction is a general proof technique where a system is first simplified, then the simplified system is analyzed, and the results are transferred back to the original system. Since a simplified system is analyzed, the proof is easier to do. Abstraction has been widely applied in program analysis, compilation and verification. For model checking, a general application of program abstractions is to reduce the complexity of a program in order to overcome the state-space explosion problem. Abstraction techniques reduce the program state space by

mapping the set of states of the actual system to an abstract set of states.

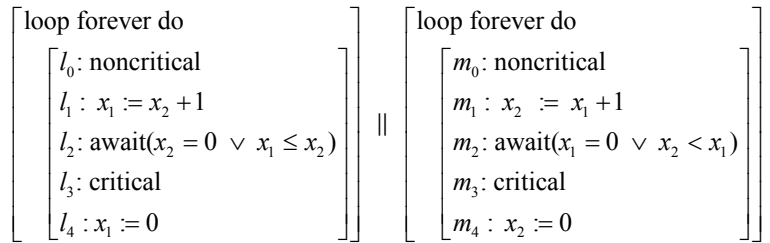
When model checking using abstraction, the main concern is that the abstractions must be property-preserving. There are two forms of property preservation: *Weak Preservation* and *Strong Preservation*. An abstraction is a weak property preserving if a set of properties true in the abstract system has corresponding properties in the concrete system that are also true, while an abstraction is a strong preserving abstraction if a set of properties with truth values either true or false in the abstract system has corresponding properties in the concrete system with the same truth values.

It is usually difficult and expensive to compute a precise abstraction directly. In order to reduce the complexity, approximation is often used. There are two main forms of approximation abstraction: over-approximation and under-approximation. In over-approximation, more behaviors are added in the abstract system than are present in the concrete system. This approach provides a very popular class of weakly preserving abstractions for universally quantified path properties. However, over-approximation often only works well for safety (or invariant) properties. In under-approximation, the behaviors are removed when going from the concrete to the abstract system. Under-approximation is also often found in the construction of an environment for a system to be checked. In this article, we focus on weak preservation, over-approximation method of abstraction.

3. Constructing the Abstract model

We describe concurrent programs using simple language [12]. As a running example, we use the program shown in Figure 1. The program guarantees mutual exclusion, that is l_3 and m_3 are never reached at the same time. Synchronization is provided by the integer variable x_1 and x_2 , which can be thought of as numbers used in

local x_1, x_2 : integer where $x_1 = x_2 = 0$



waiting-lines at bakeries.

Fig. 1. Program bakery for mutual exclusion

The program P contains a finite set of variables $V = \{x_1, \dots, x_n\}$, where each variable x_i has a associated

finite domain D_{x_i} . The set of all possible states for program P is $D_{x_1} \dots D_{x_n}$ which we denote by D . Expressions are built from variables in V , constants in D_{x_i} , and function symbols in the usual way, e.g. x_1+1 . Atomic formulas are constructed from expressions and relation symbols, e.g. $x_1 \leq x_2$. Similarly, predicates are composed of atomic formulas using negation (\neg), conjunction (\wedge), and disjunction (\vee). Let p be a predicate containing variables from V , and $d=(d_1, \dots, d_n)$ be an element from D . Then we write $d \models p$ when the predicate obtained by replacing each occurrence of the variable x_i in p by the constant d_i evaluates to true.

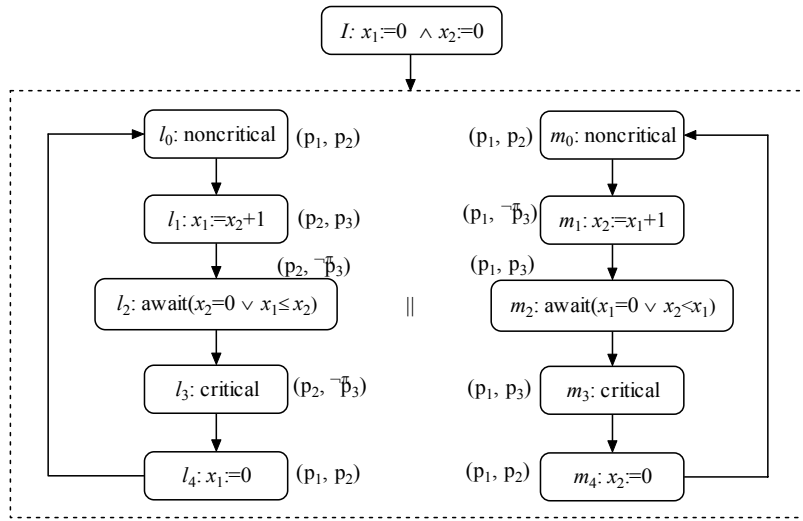


Fig. 2. The CFA of the program shown in Fig. 1. Each location is labeled by the corresponding statement label. The location s are also labeled with inferred predicates when $\mathcal{P} = \{p_1, p_2, p_3\}$ where $p_1 = (x_1==0)$, $p_2 = (x_2==0)$ and $p_3 = (x_2 < x_1)$.

The control flow graph CFG is a finite directed graph describing the flow of control in program. The nodes of the CFG correspond to the values of the program counter called control locations, and the edges denote transfer of control locations. Intuitively, the control flow automaton CFA can be obtained by viewing the CFG of a program \mathcal{C} as an automaton where the states of a CFA correspond to control locations and the transitions between states in the CFA correspond to the control flow between their associated control locations in the program. The CFA can be seen as a conservative abstraction of P 's control flow, i.e., it allows a superset of the possible traces of P . Formally, let $Stmt$ be the set of statements of P , a CFA of P is a Kripke structure $(\Sigma, I, \rightarrow, L)$, where Σ is a set of control locations, $I \in \Sigma$ is an initial location, $\rightarrow \subseteq \Sigma \times \Sigma$ is a set of transitions, $L: \Sigma \rightarrow Stmt$ is a labeling function. The

transitions between control locations reflect the flow of control between their labeling statements. The CFA of the program in Figure 1 are shown in Figure 2. Each location is labeled by the corresponding statement label, and each process is described respectively using CFA.

3.1. Predicate Abstraction

Abstraction methodologies are concerned with the process of abstraction: given a concrete program and a property to be verified, how to get to suitable abstract program such that the satisfaction on the abstract program implies the satisfaction on the concrete program.

Abstraction theory focuses on formalizing the relation between the semantic models of concrete and abstract programs. The main concern is that the abstractions must be property-preserving [13]: A property checked to be true for the abstract system should also hold for the concrete system being modeled. The key elements for abstraction are:

- Defining a set of abstract states and a mapping from concrete to abstract states
- construct a abstract transition system by constructing the abstract initial states and an abstract transition relation, and
- finally reason on the initial system by examining the abstracted system which has less states or a simpler representation

Abstract interpretation is the general framework for defining abstractions using Galois connections. Predicate abstraction is a special form of abstract interpretation. The basic idea of predicate abstraction is to replace a concrete variable by a Boolean variable that relative to a given Boolean expression over the original variable. In predicate abstraction, we model abstractly the state of a system by a set of the logical predicates which shall be used to represent sets. However, the main challenge is to identify the predicates that are necessary for proving the given property. In our framework, Starting with an initial set predicates from a specification, the algorithm iteratively computes the predicates required for the abstraction relative to that specification. These predicates are represented by Boolean variables in the abstract program.

The domain of the abstraction function α consists of sets of concrete states, represented by predicates, and ordered by implication. The range of the abstraction consists of Boolean formulas constructed using the Boolean variables B_1, \dots, B_k , ordered by implication. If X ranges over sets of concrete states and Y ranges over Boolean formulas in B_1, \dots, B_k , then the abstraction and concretization function α and γ have the following properties:

$$\alpha(X) = \wedge \{Y \mid X \Rightarrow \gamma(Y)\}, \quad \gamma(Y) = \vee \{X \mid \alpha(X) \Rightarrow Y\}.$$

In order to verify concurrent programs, we use predicate abstraction technique to model the memory state that be expressed by a set $\mathcal{P}=\{p_1, \dots, p_k\}$ of the pure Boolean expressions called the predicates. Each predicate p_i is associated with a Boolean variable b_i that represents its truth value. Let a predicate p_i denotes the subset of states that satisfy the predicate, $\{s \in \Sigma \mid s \models p_i\}$, then an element s is a member of the set if and only if $p_i(s)$ is *true*. Given a concrete memory state s and a predicate p , we can say that s satisfy p iff p evaluates to true during the execution of procedure when the memory state is s .

A valuation for \mathcal{P} is a vector $\vec{v} = v_1 \dots v_k$ of Boolean values, such that v_i expresses the Boolean value of p_i . \mathcal{V} denotes the set of all valuations, i.e., the set of abstract memory states. Intuitively, a valuation \vec{v} typically models many concrete memory states. Given a valuation $\vec{v} = v_1 \dots v_k$, the concretization $\gamma(\vec{v})$ is defined as $\bigwedge_{i \in [1, k]} p_i(s) \equiv v_i$, where $p_i(s) \equiv v_i$ is equal to p_i if v_i is true, and equal to $\neg p_i$ if v_i is false. For example, \mathcal{P} contains a single predicate $(x==0)$ and there has two valuations 0 and 1, so $\gamma(0) = \neg(x==0)$ and $\gamma(1) = (x==0)$. That is, Boolean valuation 0 models all concrete states where the variable x is not equal to 0 while Boolean valuation 1 models all concrete states where the variable x is equal to 0.

Our method is sound, in that the abstract program is always guaranteed be a conservative approximation of the original, with respect to the set of specification predicates \mathcal{P} . In order to avoid to constructing the explicit transition graph, the reduced and approximated model of the abstract program can derive directly from a concrete program.

3.2. Constructing the Abstract Model

For the construction of abstract model $K_{\mathcal{A}}$, we combine the control flow graph and the predicate abstraction to obtain the state space $C \times \mathcal{V}$, where C is control location, \mathcal{V} denotes the set of all valuations of \mathcal{P} . A state of $K_{\mathcal{A}}$ is a pair $\langle c, \vec{v} \rangle$ where $c \in C$ and $\vec{v} \in \mathcal{V}$. By computing the weakest precondition \mathcal{WP} of a predicate p relative to a given statement s , we construct the \mathcal{P}_c associated with each control location c of the CFA a finite subset of \mathcal{P} .

The process of constructing \mathcal{P}_c is known as predicate inference and the algorithm is described in Figure 3. Figure 2 show the CFA with each location c labeled by \mathcal{P}_c .

input: the set of candidate predicate \mathcal{P}
output: \mathcal{P}_c for each CFA location
do
for each location $c \in C$ **do**

case c :
 case 1: $L(c)$ is an assignment statement
 and $L(c')$ is its successor
for each $p' \in \mathcal{P}_{c'}$ **do** add $\mathcal{WP}(L(c), p')$ to \mathcal{P}_c
 case 2: $L(c)$ is a condition statement
 and $L(c')$ is its successor
if $L(c) \in \mathcal{P}$ or $\neg L(c) \in \mathcal{P}$, **then** add $L(c)$ to \mathcal{P}_c
 $\mathcal{P}_c := \mathcal{P}_c \cup \mathcal{P}_{c'}$
 case 3: $L(c)$ is a await statement
 and $L(c')$ is its successor
 $\mathcal{P}_c := \mathcal{P}_{c'}$
until no \mathcal{P}_c was changed in the for loop

Fig. 3. The algorithm of predicate inference.

We have described a method for computing the CFA of program and a set of predicates associated with each location of the CFA. The states of the abstract model \mathcal{A} correspond to the various possible valuation of the predicate in each location. However, the generation of the abstract transition is done by calling a theorem prover for each potential assignment to the current and next state predicates. In order to obtain the most precise transition relation, this requires an exponential number of calls of the theorem prover.

DEFINITION 3. The abstract Kripke $K_{\mathcal{A}}$ of concrete program is a 4-tuple $(\Sigma_{\mathcal{A}}, I_{\mathcal{A}}, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ where: (i) $\Sigma_{\mathcal{A}} = C \times \mathcal{V}$ is the set of states, (ii) $I_{\mathcal{A}} = \text{INIT}$ is the initial state, (iii) $\rightarrow_{\mathcal{A}} \subseteq S_{\mathcal{A}} \times S_{\mathcal{A}}$ is the transition relation, (iv) $L_{\mathcal{A}}: \Sigma_{\mathcal{A}} \rightarrow \mathcal{V}$ is the label function.

It is obvious that each location c of the CFA gives rise to a set of states of \mathcal{A} , $\{c\} \times \mathcal{V}_c$, where \mathcal{V}_c is the set of all predicate valuations of \mathcal{P}_c . In the worst case, the size of $K_{\mathcal{A}}$ is exponential in the size of \mathcal{P} . In addition, $K_{\mathcal{A}}$ has a unique initial state *INIT*.

For computing the abstract transition relation it will often be necessary to determine whether two expression e and e' are mutually exclusive using theorem prover. If we can not prove that there is no transition between their corresponding concrete states, then a transition between two abstract states must be added. Therefore over-approximation occurs when not enough information is available for the theorem prover to calculate a deterministic next state. We can reduce to the problem of deciding whether $\neg(e \wedge e')$ is valid. The theorem prover return true on $\neg(e \wedge e')$, then e and e' are provable mutually exclusive. Otherwise the theorem prover return false or beyond the capabilities of the theorem prover, then the theorem prover could not prove that e and e' are mutually exclusive.

3.3. Parallel Composition

When dealing with complex programs obtained as the parallel composition of simpler programs, the application of this method requires the computation of the corresponding global transition relation from which an abstraction can be computed. The question then arises whether it is possible to compute abstractions of complex programs as the parallel composition of abstractions of their components in order to avoid building the transition relation associated with the complex program. This is guaranteed if the compositionality property.

$$\frac{(K^1 \prec K_A^1) \text{ and } (K^2 \prec K_A^2)}{(K^1 \parallel K^2) \prec (K_A^1 \parallel K_A^2)}$$

holds, where \parallel is a parallel composition operator.

DEFINITION 4. (Parallel composition) The synchronous composition of transition systems $K_A^i = (\Sigma_A^i, I_A^i, \rightarrow_A^i)$, $i=1,2$, denoted $K_A^1 \parallel K_A^2$, is the transition system $(\Sigma_A^\parallel, I_A^\parallel, \rightarrow_A^\parallel)$, where:

- $\Sigma_A^\parallel = \Sigma_A^1 \otimes \Sigma_A^2$ is the set of states. $s = \langle c^1, c^2, \vec{v}_c^1, \vec{v}_c^2 \rangle \in \Sigma_A^\parallel$ iff $s^1 = \langle c^1, \vec{v}_c^1 \rangle \in \Sigma_A^1$, $s^2 = \langle c^2, \vec{v}_c^2 \rangle \in \Sigma_A^2$.
- $I_A^\parallel = \text{INIT}$ is the initial state.
- $\rightarrow_A^\parallel = \rightarrow_A^1 \otimes \rightarrow_A^2$, $(s, s') \in T_A^\parallel$ iff $(s_{|\Sigma_A^i}, s'_{|\Sigma_A^i}) \in \rightarrow_A^i$, for $i=1,2$, where $s_{|\Sigma_A^i}$ denotes the restriction of the state s to \rightarrow_A^i .

3.4. Partitioning the Candidate Predicates

We write $\mathcal{A}(\mathcal{P})$ to denote the abstract model obtained via predicate abstraction from concrete program using the set of predicates \mathcal{P} . For the sake of simplicity, we indicate this explicitly by referring to \mathcal{A}^i as $\mathcal{A}(\mathcal{P}_i)$.

PROPOSITION 1. Assume that two transition systems $\mathcal{A} = (\Sigma_{\mathcal{A}}, I_{\mathcal{A}}, \rightarrow_{\mathcal{A}})$ and $\tilde{\mathcal{A}} = (\tilde{\Sigma}_{\mathcal{A}}, I_{\mathcal{A}}, \tilde{\rightarrow}_{\mathcal{A}})$ satisfy $\Sigma_{\mathcal{A}} \subseteq \tilde{\Sigma}_{\mathcal{A}}$, and $\rightarrow_{\mathcal{A}} \subseteq \tilde{\rightarrow}_{\mathcal{A}}$, then $\tilde{\mathcal{A}}$ simulates \mathcal{A} , i.e., $\mathcal{A} \preceq \tilde{\mathcal{A}}$.

It is usually computationally expensive to compute predicate abstraction directly with respect to \mathcal{P} . In the worst case, the size of the state space in abstract model \mathcal{A} is exponential in the size of \mathcal{P} . Computing the transitions $\rightarrow_{\mathcal{A}}$ between the states requires a theorem prover. Therefore, the worst time complexities for checking validities are exponential as well. Instead of building \mathcal{A} directly, Approximation is often used to reduce the complexity. If a transition systems $\tilde{\mathcal{A}} = (\Sigma_{\tilde{\mathcal{A}}}, I_{\tilde{\mathcal{A}}}, \tilde{\rightarrow}_{\tilde{\mathcal{A}}})$ satisfies $\rightarrow_{\tilde{\mathcal{A}}} \subseteq \rightarrow_{\mathcal{A}}$, then we say that $\tilde{\mathcal{A}}$ approximates \mathcal{A} , denoted $\mathcal{A} \preceq \tilde{\mathcal{A}}$. Intuitively, if $\tilde{\mathcal{A}}$ approximates \mathcal{A} , then

$\tilde{\mathcal{A}}$ is more abstract than \mathcal{A} , i.e., has more behaviors than \mathcal{A} .

In order to make our method effectively, the set of candidate predicates \mathcal{P} can be partitioned into a number of subsets $\mathcal{P}_1, \dots, \mathcal{P}_n$, abstracted independently. Therefore, we only need to consider the effect of the abstraction of concrete program \mathcal{C} on each subset \mathcal{P}_i separately, instead of the full abstract model on the set of predicates \mathcal{P} . If we have n candidate predicates and then partition them into two sets of n_1 and n_2 elements, when applying the partition method, we only check for $2^{2n_1} + 2^{2n_2}$ validities instead of for 2^{2n} validities.

We say that two predicates interfere with each other if the sets of variables appearing in them are not disjoint. Let \equiv be the equivalence relation over \mathcal{P} which is the reflexive, transitive closure of the interference relation. The equivalence class of a predicate $p \in \mathcal{P}$ is denoted by $[p]$. If two predicate p_1 and p_2 have non-disjoint set of variables, then $[p_1] = [p_2]$. That is, a variable cannot occur in the different equivalence class of predicates. By the equivalence relation \equiv , the set of predicates can be partitioned into some subsets.

Assume each $\rightarrow_{\mathcal{A}}^i$ defines the transition relation for a predicate subset \mathcal{P}_i . Then, we apply abstraction to each $\rightarrow_{\mathcal{A}}^i$ separately, i.e., $\tilde{\rightarrow}_{\mathcal{A}} = \rightarrow_{\mathcal{A}}^1 \otimes \dots \otimes \rightarrow_{\mathcal{A}}^n$. Finally, $\tilde{\mathcal{A}}$ is given by $\otimes_{i \in [1, n]} \mathcal{A}^i$. In general the abstract system $\tilde{\mathcal{A}}$ computed using a partitioning has more transitions than the system \mathcal{A} computed without using the partitioning.

PROPOSITION 2. Assume that $\mathcal{A} = (\Sigma_{\mathcal{A}}, I_{\mathcal{A}}, \rightarrow_{\mathcal{A}})$ denotes the abstract model obtained via predicate abstraction using the set of predicates \mathcal{P} , and $\mathcal{A}^i = (\Sigma_{\mathcal{A}}^i, I_{\mathcal{A}}^i, \rightarrow_{\mathcal{A}}^i)$ denotes the abstract model obtained using the set of predicates \mathcal{P}_i , such that the set of predicates \mathcal{P}_i is a partition of \mathcal{P} (satisfying the cover property $\mathcal{P} = \bigvee_{i \in [1, n]} \mathcal{P}_i$ and the disjoint property $\forall 1 \leq i, j \leq n, i \neq j \Rightarrow \mathcal{P}_i \cap \mathcal{P}_j = \emptyset$). Let $\tilde{\mathcal{A}} = \otimes_{i \in [1, n]} \mathcal{A}^i$, then $\mathcal{A} \subseteq \tilde{\mathcal{A}}$.

PROOF: $\tilde{\mathcal{A}} = \otimes_{i \in [1, n]} \mathcal{A}^i = (\Sigma_{\tilde{\mathcal{A}}}^\otimes, I_{\tilde{\mathcal{A}}}^\otimes, \rightarrow_{\tilde{\mathcal{A}}}^\otimes)$, where $\Sigma_{\tilde{\mathcal{A}}}^\otimes = \otimes_{i \in [1, n]} \Sigma_{\mathcal{A}}^i$, $I_{\tilde{\mathcal{A}}}^\otimes = \text{INIT}^1 \times \dots \times \text{INIT}^n$, $\rightarrow_{\tilde{\mathcal{A}}}^\otimes = \otimes_{i \in [1, n]} \rightarrow_{\mathcal{A}}^i$.

According to the definition of $\Sigma_{\tilde{\mathcal{A}}}^\otimes$, $s = \langle c, \vec{v}_c^1, \dots, \vec{v}_c^n \rangle \in \Sigma_{\tilde{\mathcal{A}}}^\otimes \Leftrightarrow s = \langle c, \vec{v}_c \rangle \in \Sigma_{\mathcal{A}}$ such that $\vec{v}_c = \langle \vec{v}_c^1, \dots, \vec{v}_c^n \rangle$, so $\Sigma_{\tilde{\mathcal{A}}} = \Sigma_{\mathcal{A}}$.

$$I_{\tilde{\mathcal{A}}}^\otimes = \text{INIT}^1 \times \dots \times \text{INIT}^n = \text{INIT} = I_{\mathcal{A}}.$$

Let $s = \langle c, \vec{v}_c^1, \dots, \vec{v}_c^n \rangle$ and $s' = \langle c', \vec{v}_c'^1, \dots, \vec{v}_c'^n \rangle$, for all $1 \leq i \leq n$, $(\langle c, \vec{v}_c^i \rangle, \langle c', \vec{v}_c'^i \rangle) \in \rightarrow_{\mathcal{A}}^i$, i.e., $(s_{|\Sigma_{\mathcal{A}}^i}, s'_{|\Sigma_{\mathcal{A}}^i}) \in \rightarrow_{\mathcal{A}}^i$, $(s, s') \in \rightarrow_{\mathcal{A}}^{\otimes} \Leftarrow (s, s') \in \rightarrow_{\mathcal{A}}$, so $\rightarrow_{\mathcal{A}} \subseteq \rightarrow_{\mathcal{A}}^{\otimes}$.

So $\mathcal{A} \subseteq \tilde{\mathcal{A}}$.

4. Conclusion

We present a methodology for automatically constructing an abstraction of concurrent programs using predicate abstraction and theorem proving. However, the process of extracting a finite model from a concrete program using predicate abstraction can be exponential in the number of predicates used. In order to make model construction effective, our method is to partition the set of candidate predicates into subsets, and construct abstract model independently.

5. Acknowledgment

This work is supported by National Outstanding Young Scientists Foundation of China (No.60425206), Natural Science Foundation of China (No.60373066), Guangxi Natural Science Foundation of China (No. 0542036).

6. References

[1] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In: Proceedings of Computer Aided Verification, LNCS 1254, pp.72–83, 1997.
 [2] 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 POPL, pp.238—252, 1977.
 [3] K. Havelund and T. Pressburger. Model Checking JAVA Programs using JAVA PathFinder. International

Journal on Software Tools for Technology Transfer, 2(4): 366–381, 2000.

[4] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In: Proceedings of POPL, pp. 58–70, 2002
 [5] E. M. Clarke, O. Grumberg, and D. A. Peled. Model Checking. MIT Press, 1999.
 [6] J. C. Corbett, M. B. Dwyer and J. Hatcliff et al. Bandera: Extracting Finite-state Models from Java Source Code. In: Proceedings of ICSE, pp. 439–448, 2000.
 [7] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. In: Proceedings of ICSE, pp. 385–395, 2003.
 [8] S. Chaki, E. M. Clarke, A. Groce et al. Efficient Verification of Sequential and Concurrent C Programs. Formal Methods in System Design, 25(2-3): 129-166, 2004.
 [9] T. Ball, R. Majumdar, T. D. Millstein et al. Automatic Predicate Abstraction of C Programs. In: SIGPLAN Conference on PLDI. pp. 203–213, 2001.
 [10] S. Das, D. L. Dill, and S. Park. Experience with Predicate Abstraction. In: Proceedings of Computer Aided Verification, LNCS 1633, pp. 160–171, 1999.
 [11] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In SPIN workshop, LNCS 2057, pp. 103-122, 2001.
 [12] Z. Manna, A. Pnueli. Temporal verification of reactive systems: safety. Springer, 1995.
 [13] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. Formal Methods in System Design, 6(1):11–44, 1995.
 [14] S. Bensalem, Y Lakhnech and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In: Proceedings of Computer Aided Verification, LNCS 1427, pp. 319–331, 1998.
 [15] S. Bensalem, S. Graf and Y. Lakhnech. Abstraction as the Key for Invariant Verification. Verification: Theory and Practice 2003: 67-99.