Automatic Verification of Pointer Data-Structure Systems for All Numbers of Processes*

Farn Wang

Institute of Information Science, Academia Sinica
Taipei, Taiwan 115, Republic of China
+886-2-27883799 ext. 1717; FAX +886-2-7824814; farn@iis.sinica.edu.tw

ABSTRACT

Real-world softwares for concurrent systems may involve data-structures linked together with pointers. Even with such sophistication, they are usually supposed to work regardless of the number of processes. We propose a new automatic approximation method to safely verify algorithms used in such systems. The central idea is to construct a finite *collective image set (CIS)* which collapses reachable state representations for all implementations of all numbers of processes. Our collapsing scheme filters out unimportant information of system behaviors and results in CIS's with manageable space requirements which allow for efficient verification. Analysis shows our method can automatically generate a CIS of size 657 to verify that a version of Mellor-Crummy & Scott's algorithm preserves mutual exclusion for all numbers of processes.

1 Introduction

With the success of automatic verification technology for hardware systems in recent years[6, 8], people are now naturally looking forward to automating the tasks of software verification. However, with vast variety of abstract devices like unlimited concurrencies, pointers, dynamic data-structures, range-unbounded variables, unbounded buffers, ..., etc., software systems are far more sophisticate than hardware systems. Straightforward extension of the existing state-based technology[9, 10, 13] is at an inappropriate abstractness level and has generally bumped into steep complexities of the verification problems for software systems[1, 2, 14, 22, 23]. For example, iteratively model-checking on a concurrent algorithm implemented for a few different numbers of processes[4, 19, 24, 21] in no way proves the correctness of the algorithm for all numbers of processes. Besides, the algorithm implementations, which current model-checking technology can verify, are still too small as far as the numbers of processes are considered. In this paper, we propose a new automatic approximation verification method

* The work is partially supported by NSC, Taiwan, ROC under grant NSC 87-2213-E-001-007.

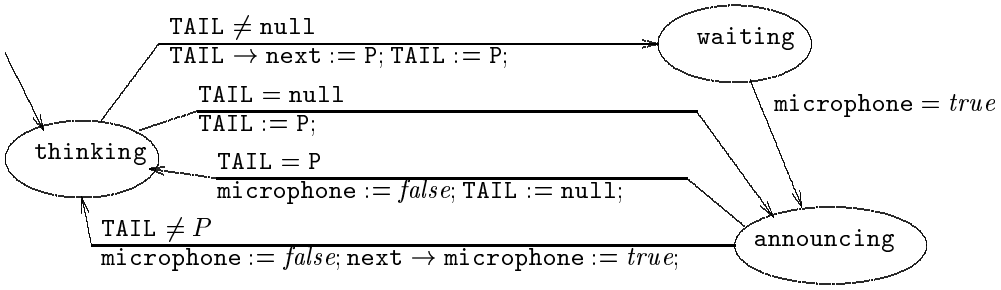

Fig. 1. Speaking philosophers

which reasons at an abstractness level similar to that of hamans by filtering out unimportant details of system behaviors for efficient veification. As will be shown in later sections, our method is able to automatically verify real-world algorithms like Mellor-Crummey & Scott's mutual exclusion algorithm (*MCS algorithm* for short) for concurrent systems[18] regardless of the numbers of processes. To our best knowledge, MCS algorithm has so far defied automatic verification.

We are dealing with concurrent systems of unspecified number of processes running different copies of a same algorithm. Conceptually, such an algorithm S is a tuple $(\mathrm{Enu}_S, \mathrm{Ptr}_S, \mathrm{Enu}_A, \mathrm{Ptr}_A, A(P))$ where Enu_S and Enu_A are respectively the sets of *global* and *local enumerate variables* (as in Pascal programming language), Ptr_S and Ptr_A are respectively the sets of *global* and *local pointers*, and $A(P)$ is the *process program template*, with *process identifier* symbol P. The global enumerate variables and pointers are accessible to all processes in an implememtnation of the algorithm. On the other hand, each process has its own local enumerate variables and pointers which no other processes can access.

Example 1. : Speaking philosophers There are many philosophers in a hall with a single microphone on the platform. Most of the time, they walk around pondering on the meaning of life. When a philosopher finds some truth worthy announcing, she/he will try to grasp the microphone and announce the finding on the platform. To avoid confusion, *mutual exclusion* to the microphone has to be enforced, that is, at any moment, at most one philosopher is allowed to use the microphone.

In figure 1, we have an algorithm for each philosopher to guarantee mutual exclusion. Each philosopher records its own Boolean variable `microphone` which is true if the the philosopher can exclusively use the microphone. Also, a global waiting queue is implemented with *local pointer* `next` of each philosopher and *global pointer* `TAIL` which points to the end of the queue. For each philosopher waiting in the queue, her/his `next` points to the next philosopher waiting in the queue. The purpose of the algorithm is that for any number of philosophers in the hall, mutual exclusion to the microphone has to be maintained. The algorithm

is presented in the form of finite-state automata. We have circles for *operation modes*: `thinking`, `waiting`, and `announcing`. Initially, each philosopher is mode `thinking`, all pointers are set to `null`, and all enumerate variable are set to zero (or *false* for Boolean variables). In between two operation modes, we have arcs for *transitions*. On each arc, we label the triggering condition and the actions to be taken on the happening of the transition. For example, at mode `thinking`, we may transit to mode `waiting` if `TAIL` is not equal to `null`, and assign the transiting philosopher's identifier P to the `next` pointer of the philosophoer recorded by `TAIL`, and then assign P to global pointer `TAIL`. ||

Given algorithm $S = (\mathrm{Enu}_S, \mathrm{Ptr}_S, \mathrm{Enu}_A, \mathrm{Ptr}_A, A(P))$ and a finite set of processes $\{p_1, \ldots, p_m\}$, in our notation, we shall write $S\{p_1, \ldots, p_m\}$ for an *implementation* of S of concurrent processes $p_1, \ldots, p_m$ whose behaviors are all described by $A(P)$. We put our verification tasks in the framework of *safety bound problem*. Given an algorithm S, a process predicate η (defined later), and a count C, the safety bound problem asks if there is a finite set Π of processes such that in a computation of $S\Pi$, C or more processes can satisfy η simultaneously. Safety bound problem is general enough to describe many practical verification tasks, e.g. mutual exclusion, or process state reachability. In section 3, we shall prove that such a problem is undecidable, i.e. no computers with finite memory can answer such a queustion.

Since the verification problem is extremely difficult, we instead develop an automatic approximation method which can answer the safety of a large class of such algorithms regardless of the number of processes. Our intention is to construct a finite *collective image set (CIS)* whose elements are reachable state images describing the behaviors of all implementations with any number of processes. Engineers' intelligence and experiences in design and verification is encoded in the mapping from states to images in CIS's and seems to result in small CIS's even for complicate data-structures. For safety analysis, if we can construct a finite CIS which contains no images of states violating the safety specification, then it is good enough to conclude that the algorithm is safe for any number of processes. However if there is a state image violating the safety specification in the CIS, then no conclusion can be made because the image may be included due to insufficient approximation precision.

With the known high complexities of most verification problem models[1, 2, 14, 22, 23], it is clear that current technology cannot identify a large class of concurrent algorithms subject to efficient verification. On the other hand, we argue that our technology can serve to identify such a large class of "well-designed" concurrent algorithms which can be efficiently verified. In section 5, we shall establish the mighty lemma 2 which allows us to eliminate much combinatorial complexity in CIS without sacrifising approximation precision. In section 6, we shall analyze our method on a modified MCS algorithm in which local pointers are set to null whenever the current values of the pointers will not be used in the future. The modification is consistent with good programming practice of elimination of "stray" pointers. The interesting thing here is that our method can generate a small CIS of size 657 for the modified MCS algorithm while fails

to do so for the original one. This shows that our verification method is indeed more efficient for "good" designs.

Here is our presentation plan. Section 2 discusses some related work. Section 3 formally defines the type of algorithms we aim to analyze and shows how hard the verification problem is. Section 4 rigorously defines our state collapsing scheme. Section 5 describes how to construct CIS's, how to verify safety properties with our method, and analyze the complexity. Section 6 shows that our method works efficiently for a modified version of MCS algorithm. Section 7 is the conclusion.

We shall adopt the following notations. Given a set or sequence K, $|K|$ is the number of elements in K. For each element e in K, we also write $e \in K$. We let $\mathcal{N}$ be the set of nonnegative integers.

2 Related work

Apt and Kozen already showed that in general verification of systems with unknown number of concurrent processes is undecidable[3]. This means that such verification problems are extremely hard and we can only rely on semi-decision procedures or, as in this work, approximation algorithms to answer them. Otherwise, we can also investigate to find out decidable subclasses of the problem. In the following, we briefly describe some of the related work.

Browne, Clarke, and Grumberg [5] use bisimulation equivalence relation between global state graphs of systems of different sizes. The equivalence relation must be strong enough for the method to work. Thus the construction of the equivalence relation is difficult to mechanize.

Clarke, Grumberg, and Jha[11] propose to use regular languages to specify properties in a linear network with unknown number of processes. Then state-equivalence relation is defined based on the regular languages and a mechanical method is defined to synthesize a network invariant $\mathcal{I}$ in the hope that $\mathcal{I}$ can be contained by the specification. But there is no guarrantee that $\mathcal{I}$ is a model of the specification even if the system indeed satisfies the specification. Moreover, it is not known whether using the specification regular languages to derive equivalence class properly perserves the reasonings behind the system design. Lesens, Halbwachs, and Raymond[17] furthered the approach by designing a language for the specification in systems with complex structures and by using fixed-point resolution with different heuristics to calculate many network invariants. Compared to our approach, we argue that the technique of CIS better captures the design reasoning that the relations between processes in different states are far more important than the actual numbers of processes in different states. We believe in verifying complex systems, without utilizing the reasoning behind the system designs, state-explosion problem cannot be properly dealt with.

Kurshan and McMillan[16] proposes to use network structural induction which is not guarranteed to terminate. Also inductive hypothesis is difficult to construct, although once it is ready, the whole approach is usually very efficient. Compared to our approach, we are using an approximation algorithm which

captures the engineers' view of linear list. Users only have to guess the value of bound B, used in CIS construction, which for many real-world concurrent algorithms, small value like 1 will do.

Emerson and Naamjoshi[12] specialized on static token ring networks. They prove that for certain properties, verification on small size networks can be used to guarrantee the verification of large size networks. In contrast, our method is applicable to all different configurations of *"dynamic"* networks of processes.

Boigelot and Godefroid[7] choose to use state-space exploration to handle the verification problems of systems with unbounded FIFO queues. Their state-space representation is constructed by collapsing FIFO queues. Their approach does not guarrantee termination.

Recently, the author also has researched on the technique of collective quotient structures on dynamic linear networks [20]. The idea is similar to that of CIS in that they both collapse state-spaces of all implementations into single structures. However my work here is more general for pointer data-structures which allows the development of lemma 2, in section 5, and can lead to significant reduction in time and space complexity.

3 Concurrent algorithms and safety bound problem

We are dealing with concurrent algorithms with a local data structure for each process. The address of a data structure can be viewd as the identity of the corresponding process. We shall have the convention that if a process is named p, then p is also the address of process p's data-structure.

Two types of variables can be declared. The first is the type of *enumerate variables* with predefined finite integer value ranges. For convenience, we can also give symbolic names to those integer values. As in example 1, local Boolean variable `microphone` with values in $\{false, true\}$ denotes if a philosopher is using the microphone. Traditionally, *false* is interpreted as 0 while *true* as 1. The second is the type of *pointers (address variables)* to processes (data-stuctures). As in example 1, `TAIL` is a pointer to the tail of a queue. Variables can be declared as *global* variables which all processes can access, or *local* variables of a process which only the declaring process can directly access. Test can be made to determine if an enumerate type variable's content equals to a constant, if a pointer is `null`, or if two pointers point to the same process. We can also assign a constant to an enumerate type variable or to assign a process address to a pointer. In the following, we shall first formally define the syntax and semantics of our systems, and then define the safety bound problem.

3.1 Syntax of algorithm descriptions

Given algorithm $S = (\mathrm{Enu}_S, \mathrm{Ptr}_S, \mathrm{Enu}_A, \mathrm{Ptr}_A, A(P))$, a *process predicate* η of S has the following syntax.

$$\begin{aligned}
\eta &::= \gamma_1 = \gamma_2 \mid \delta_1 = \delta_2 \mid \neg\eta \mid \eta_1 \vee \eta_2 \\
\gamma &::= c \mid x \mid z \mid y \to x \mid w \to x \\
\delta &::= \mathrm{null} \mid P \mid w \mid y \mid w \to y
\end{aligned}$$

where $c \in \mathcal{N}$, $x \in \mathrm{Enu}_A$, $z \in \mathrm{Enu}_S$, $w \in \mathrm{Ptr}_S$, and $y \in \mathrm{Ptr}_A$. Traditional shorthands are $\gamma_1 \neq \gamma_2 \equiv \neg(\gamma_1 = \gamma_2)$, $\delta_1 \neq \delta_2 \equiv \neg(\delta_1 = \delta_2)$, $\eta_1 \wedge \eta_2 \equiv \neg((\neg\eta_1) \vee (\neg\eta_2))$, and $\eta_1 \rightarrow \eta_2 \equiv (\neg\eta_1) \vee \eta_2$, Thus a process may operate on conditions of the global and local variables, and also on the local variables of the processes pointed to by global pointers. We let $\mathrm{PPredicate}_S$ be the set of all process predicates of S.

A finite sequence κ of actions of algorithm S has the following syntax.

$$\begin{aligned}
\kappa &::= \ \mid \alpha\kappa \\
\alpha &::= \omega_1 := \omega_2; \mid \epsilon := \delta; \\
\omega &::= x \mid z \mid w \rightarrow x \mid y \rightarrow x \\
\epsilon &::= \ y \mid w \mid w \rightarrow y \mid y_1 \rightarrow y_2
\end{aligned}$$

Here δ is defined as in the syntax of process predicate. α defines what an *action* looks like. The set of all finite sequences of actions of S is named $\mathrm{Actions}_S$.

An algorithm S is described as $S = (\mathrm{Enu}_S, \mathrm{Ptr}_S, \mathrm{Enu}_A, \mathrm{Ptr}_A, A(P))$ where Enu_S is the set of global enumerate variables, Ptr_S is the set of global pointers, Enu_A is the set of local enumerate variables, Ptr_A is the set of local pointers, and $A(P)$ is the program template for each process with identifier symbol P. Program template $A(P)$ has the syntax similar to that of finite-state automata. $A(P)$ is conceptually a tuple (Q, q_0, E, τ, π) with the following restrictions.

- Q is a finite set of operation modes.
- $q_0 \in Q$ is the initial operation mode.
- $E \subseteq Q \times Q$ is the set of transitions among operation modes.
- $\tau : E \mapsto \mathrm{PPredicate}_S$ is a mapping which defines the triggering condition of each transition.
- $\pi : E \mapsto \mathrm{Actions}_S$ is a mapping which defines the action sequence performed at the happening of each transition.

We do require that there is a variable $\texttt{mode} \in \mathrm{Enu}_A$ which records the current operation mode of the corresponding process.

3.2 Computation of systems

Let $\mathit{\Pi}$ be the set of all processes (conceptually represented by either their identifiers, or their data-structure addresses) in an implementation. For each enumerate-type variable x, we let D_x be $\{0\}$ unioned with the set of all constants assigned to x in process program $A(P)$ labeled in all transition's assignment sequence. Especially, $D_{\texttt{mode}} = Q$ and $\texttt{mode} = 0$ means the process is in its initial operation mode.

A *process state* π is a mapping from $\mathrm{Enu}_A \cup \mathrm{Ptr}_A$ to $\mathcal{N} \cup \mathit{\Pi} \cup \{\mathrm{null}\}$ such that $\pi(x) \in D_x$ if $x \in \mathrm{Enu}_A$; and $\pi(x) \in \mathit{\Pi} \cup \{\mathrm{null}\}$ if $x \in \mathrm{Ptr}_A$. We shall let $\mathit{\Gamma}_A$ be the set of all process states.

A *global state* of $S\mathit{\Pi}$ is a pair (ψ, ϕ) with the following restrictions.

- ψ is a mapping from $\mathrm{Enu}_S \cup \mathrm{Ptr}_S$ to $\mathcal{N} \cup \mathit{\Pi} \cup \{\mathrm{null}\}$ such that $\psi(x) \in D_x$ if $x \in \mathrm{Enu}_S$; and $\psi(x) \in \mathit{\Pi} \cup \{\mathrm{null}\}$ if $x \in \mathrm{Ptr}_S$.
- ϕ is a mapping from $\mathit{\Pi}$ to $\mathit{\Gamma}_A$.

Given a global state $\nu = (\psi, \phi)$, a process $p \in \Pi$, and a process predicate $\eta \in \text{PPredicate}_S$, we define the relation of p *satisfies* η at ν, written $p, \nu \models \eta$, in the following inductive way. Assume that $x \in \text{Enu}_A$, $z \in \text{Enu}_S$, $y, y_1, y_2 \in \text{Ptr}_A$, and $w \in \text{Ptr}_S$.

- $p, \nu \models \gamma_1 = \gamma_2$ iff $\text{value}(p, \nu, \gamma_1) = \text{value}(p, \nu, \gamma_2)$
- $\text{value}(p, \nu, c) = c$
- $\text{value}(p, \nu, x) = \phi(p)(x)$
- $\text{value}(p, \nu, z) = \psi(z)$
- $\text{value}(p, \nu, y \to x) = \phi(\phi(p)(y))(x)$
- $\text{value}(p, \nu, w \to x) = \phi(\psi(w))(x)$
- $p, \nu \models \delta_1 = \delta_2$ iff $\text{value}(p, \nu, \delta_1) = \text{value}(p, \nu, \delta_2)$
- $\text{value}(p, \nu, \text{null}) = \text{null}$
- $\text{value}(p, \nu, P) = p$
- $\text{value}(p, \nu, w) = \psi(w)$
- $\text{value}(p, \nu, y) = \phi(p)(y)$
- $\text{value}(p, \nu, w \to y) = \phi(\psi(w))(y)$
- $\text{value}(p, \nu, y_1 \to y_2) = \phi(\phi(p)(y_1))(y_2)$
- $p, \nu \models \neg\eta$ iff it is not the case that $p, \nu \models \eta$
- $p, \nu \models \eta_1 \vee \eta_2$ iff $p, \nu \models \eta_1$ or $p, \nu \models \eta_2$

Given an action α of S, the new global state obtained by applying α to p at ν, written $\text{next\_state}(p, \nu, \alpha)$, is defined in the following way.

- $(\psi', \phi') = \text{next\_state}(p, \nu, x := \omega;)$ is identical to ν except that $\phi'(p)(x) = \text{value}(p, \nu, \omega)$.
- $(\psi', \phi') = \text{next\_state}(p, \nu, z := \omega;)$ is identical to ν except that $\psi'(z) = \text{value}(p, \nu, \omega)$.
- $(\psi', \phi') = \text{next\_state}(p, \nu, w \to x := \omega;)$ is identical to ν except that $\phi'(\psi(w))(x) = \text{value}(p, \nu, \omega)$.
- $(\psi', \phi') = \text{next\_state}(p, \nu, y \to x := \omega;)$ is identical to ν except that $\phi'(\phi(p)(y))(x) = \text{value}(p, \nu, \omega)$.
- $(\psi', \phi') = \text{next\_state}(p, \nu, y := \delta;)$ is identical to ν except that $\phi'(p)(y) = \text{value}(p, \nu, \delta)$.
- $(\psi', \phi') = \text{next\_state}(p, \nu, w := \delta;)$ is identical to ν except that $\psi'(w) = \text{value}(p, \nu, \delta)$.
- $(\psi', \phi') = \text{next\_state}(p, \nu, w \to y := \delta;)$ is identical to ν except that $\phi'(\psi(w))(y) = \text{value}(p, \nu, \delta)$.
- $(\psi', \phi') = \text{next\_state}(p, \nu, y_1 \to y_2 := \delta;)$ is identical to ν except that $\phi'(\phi(p)(y_1))(y_2) = \text{value}(p, \nu, \delta)$.

Given an action sequence $\alpha_1 \ldots \alpha_n \in \text{Actions}_S$, we let $\text{next\_state}(p, \nu, \alpha_1\alpha_2 \ldots \alpha_n) = \text{next\_state}(p, \text{next\_state}(p, \nu, \alpha_1), \alpha_2 \ldots \alpha_n)$.

The *initial state* (ψ_0, ϕ_0) of an implementation $S\Pi$ must satisfies the following restrictions: (1) $\psi_0(z) = 0$ for all $z \in \text{Enu}_S$, (2) $\psi_0(w) = \text{null}$ for all $w \in \text{Ptr}_S$, (3) $\phi_0(p)(x) = 0$ for all $p \in \Pi$ and $x \in \text{Enu}_A$, and (4) $\phi_0(p)(y) = \text{null}$ for all $p \in \Pi$ and $y \in \text{Ptr}_A$. We assume that processes interact with *interleaving semantics*, that is at any moment, at most one process can execute a transition. Interleaving semantics is well-accepted in verification theory for its simplicity.

A *computation* of an implementation $S\Pi$ is a sequence $\rho = \nu_0\nu_1 \ldots \nu_k \ldots\ldots$ of global states with $\nu_k = (\psi_k, \phi_k)$ for all $k \geq 0$ such that

- ν_0 is the initial state of $S\Pi$; and
- for each $k \geq 0$, either
  - $\nu_k = \nu_{k+1}$ or
  - there is a $p \in \Pi$ and transition from q to q' such that $p, \nu_k \models \tau(q, q')$ and $\text{next\_state}(p, \nu_k, \pi(q, q')) = \nu_{k+1}$.

3.3 Safety bound problem and its undecidability

The computation definition of our algorithm implementations is independent of the real names used for each process in Π. Never the names of processes are used to affect the behaviors of our implementations. Instead, only the count of processes in Π is important. Thus it is better if we can present our safety analysis problem regardless of the actual names used for processes. Given a global state $\nu = (\psi, \phi)$ of an implementation $S\Pi$ and a process predicate η, $\mathbf{count}_\eta(\nu)$ is the number of processes satisfying η at ν, i.e. $|\{p \mid p \in \Pi; p, \nu \models \eta\}|$. A computation $\rho = \nu_0\nu_1 \ldots \nu_k \ldots\ldots$ of $S\Pi$ violates safety property η with bound $c \in \mathcal{N}$ iff there is a $k \geq 0$ such that $\mathbf{count}_\eta(\nu_k) > c$.

The *safety bound problem* instance $\text{SBP}(S, \eta, c)$ is to determine if for all finite sets Π of processes and all computation ρ of $S\Pi$, ρ does not violate safety property η with bound c. Such a problem framework can be used to verify process state reachability problem[14] which is a special case of $\text{SBP}(S, \eta, c)$ with $c = 1$. Also mutual exclusion problem can be formulated with $c = 1$. Reader-Writer problem can be formulated with c set to the number of readers.

However, such a problem is extremely difficult to answer. In fact, we can show $\text{SBP}(S, \eta, 1)$ for a given S and η is undecidable, i.e. there is no computer with finite amount of memories capable of answering $\text{SBP}(S, \eta, 1)$. Lemma 1 proves this by reducing two-counter machine halting problem[15] to $\text{SBP}(S, \eta, 1)$. A two-counter machine M has a finite-state control and two counters which can hold any natural numbers. The finite-state control can increment a counter, decrement a counter, or transit between finitely many operation modes by testing whether a particular counter contains zero. It is known that two-counter machine can emulate Turing machine whose halting problem cannot be answered by any computers with finite amount of memories.

Lemma 1. *: Two-counter machine halting problem is reducible to SBP*$(S, \eta, 1)$.
Proof : Due to page-limit, we shall only give a sketch of the proof. Suppose we are given a two-counter machine M. We shall implement two stacks to emulate the two counters respectively with pointers linking together adjacent elements in the stacks. The halting state of two-counter machine is encoded in η. The first transiting process in the computation will be used to emulate the finite-state control. The second and third transiting processes in the computation will respectively be used to emulate the stack bottoms for the two counters. Then each increment operation of a counter will need one process to be pushed onto the corresponding stacks. If there is not enough number of processes for the

increment in the implementation, then the computation simply halts in a state without satisfying η. Each decrement operation of a counter will need the top process in the corresponding stack to be popped. Testing for zero value of a counter can be implemented by asking if the stack top process is equal to the stack bottom process for the corresponding counter. In this way, we can construct S and η such that $\mathrm{SBP}(S, \eta, 1)$ answers true iff M reaches its halting state. ||

4 Collective image set

With lemma 1 and many similar complexity results[1, 2, 14, 22, 23], it is clear that classic verification technology is not able to handle the complexity incurred by verification problems for concurrent algorithms with sophisticate data-structures. However, we have observed that classic verification theory does not distinguish "well-behaved" systems from "bad" systems. In many algorithms for concurrent systems, the number of processes is usually not a crucial factor in the correctness of systems. In the following, we shall formally define our global state images for all implementation. There are two crucial steps in our collapsing scheme.

- Process p *points to* process p' (and p' is called a *reference* of p) in state (ψ, ϕ) if there is a $y \in \mathrm{Ptr}_A$ such that $\phi(p)(y) = p'$. State of each process p is collapsed down to a *PDSI (process data-structure image)* which only records information that process p can read from the local variables of itself and its references.
- A global state image, called *GDSI (global data-structure image)*, is treated as a multiset of PDSI's of the participating processes. However, users have to choose a constant B. When more than B processes have the same PDSI in a global state, they are only recorded by a flag (∞ here) which denotes that the number of processes in that PDSI exceeds B.

With such techniques, we are able to map the infinitely many states of all implememtations down to finitely many global state images.

In the following subsections, we shall define rigorously the image mapping of global states of implementations. Then we shall define the *transitions*, among GDSI's, which corresponds to transition rules described in $A(P)$.

We need the following conventions regarding number systems respecting a bound B. Let $\mathcal{N}^\infty = \mathcal{N} \cup \{\infty\}$ where ∞ means any number greater than B. For any $c \in \mathcal{N}$, $c < \infty$. For any $c, d \in \mathcal{N}^\infty$, $c \leq \infty$ and $c + \infty = \infty + d = \infty$.

Given any two numbers c and B in $\mathcal{N}^\infty$, we let $c^{(B)} = c$ if $c \leq B$; $c^{(B)} = \infty$ if $c > B$. Finally, $[0, B]^{(\infty)} = \{0, 1, \ldots, B\} \cup \{\infty\}$.

4.1 Pointer data-structure images

The global-state images in our method is characterized by finite sets of propositional atoms. We shall first define *PSI (process state images)* as building blocks to construct PDSI. PSI represents the observation a process can make without going through the pointers. The *process state image (PSI)* of process p at

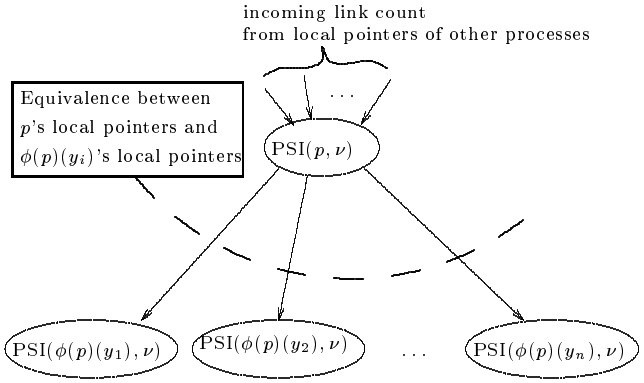

Fig. 2. Information of PDSI

state $\nu = (\psi, \phi)$, in symbols $\mathrm{PSI}(p, \nu)$, is a finite set of atoms constructed in the following way.

$$\begin{aligned}\mathrm{PSI}(p,\nu) = & \{x = c \mid x \in \mathrm{Enu}_A; c \in D_x; c = \phi(p)(x)\} \\ & \cup \{z = c \mid z \in \mathrm{Enu}_S; c \in D_x; c = \psi(z)\} \\ & \cup \{P = y \mid y \in \mathrm{Ptr}_A; p = \phi(p)(y)\} \\ & \cup \{P = w \mid w \in \mathrm{Ptr}_S; p = \psi(w)\} \\ & \cup \{y = \mathrm{null} \mid y \in \mathrm{Ptr}_A; \phi(p)(y) = \mathrm{null}\} \\ & \cup \{w = \mathrm{null} \mid w \in \mathrm{Ptr}_S; \psi(w) = \mathrm{null}\} \\ & \cup \{y = w \mid y \in \mathrm{Ptr}_A; w \in \mathrm{Ptr}_S; \phi(p)(y) = \psi(w)\} \\ & \cup \{y_1 = y_2 \mid y_1, y_2 \in \mathrm{Ptr}_A; \phi(p)(y_1) = \phi(p)(y_2)\}\end{aligned}$$

Here we use P to symbolically represent the address p of the corresponding process. Note that we conveniently define PSI's to also record information on global variables. Thus in our GDSI defined later, PSI's of all processes must all agree on the informations of those global variables.

Conveniently, we shall let L_A be the set of all PSI's.

Our process image, called *process data-structure image (PDSI)*, for a process p is graphically shown in figure 2 and only records (1) the PSI's of p and p's references; (2) the equality among p and p's references' references (i.e. if the references point back); and (3) the multiset of incoming local pointers from peer processes to p with bound B ($\mathrm{ILM}^{(B)}$ for *incoming link multiset* with bound B). As in example 1, the references of a philosopher is its predecessor and successor in the queue and ILM is of size 0, 1, or 2. In the following, we shall make the definition of PDSI more precise.

The *referenced image (RI)* from process p through $y \in \mathrm{Ptr}_A$ at state $\nu = (\psi, \phi)$, in symbols $\mathrm{RI}_y(p, \nu)$, is the set of basic true relations between p and $\phi(p)(y)$ observed from p at state ν and is constructed in the following way.

$$\begin{aligned}\mathrm{RI}_y(p,\nu) = \quad & \{P = y \to y_1 \mid y_1 \in \mathrm{Ptr}_A; p = \phi(\phi(p)(y))(y_1)\} \\ & \cup \{\mathrm{RV}(y,a_1) = \mathrm{RV}(y,a_2) \mid a_1 = a_2 \in \mathrm{PSI}(\phi(p)(y),\nu)\}\end{aligned}$$

where RV stands for *referenced variable* and $\mathrm{RV}(y,a) = y \to a$ if $a \in \mathrm{Enu}_A \cup \mathrm{Ptr}_A$; or $\mathrm{RV}(y,a) = a$ otherwise. Note that all atomic propositions true in p's references are reinterpreted with the corresponding local pointer y of p through function RV().

The *incoming link multiset (ILM)* of process p at state $\nu = (\psi, \phi)$, in symbols $\mathrm{ILM}(p,\nu)$, is the multiset of distinct local pointers from peer processes PSI pointing to p. A multiset is conceptually a set which allows an element to repeat many times. Mathematically, it is a mapping from a domain to $\mathcal{N}$. Formally speaking, for all $\lambda \in L_A$ and $y \in \mathrm{Ptr}_A$,

$$\mathrm{ILM}(p,\nu)(\lambda, y) = |\{(p', y) \mid p' \in \Pi; \mathrm{PSI}(p',\nu) = \lambda; \phi(p')(y) = p\}|$$

To respect bound B, we let $(\mathrm{ILM}(p,\nu))^{(B)}$ be a mapping from the domain to $[0,B]^{(\infty)}$ such that for all $\lambda \in L_A$ and $y \in \mathrm{Ptr}_A$, $(\mathrm{ILM}(p,\nu))^{(B)}(\lambda, y) = (\mathrm{ILM}(p,\nu)(\lambda, y))^{(B)}$.

With the definition of PSI, RI, and ILM, we now can define the PDSI of a process p in a state $\nu = (\psi, \phi)$ with bound B, in symbols $\mathbf{PDSI}^{(B)}(p,\nu)$, as the following pair

$$\mathbf{PDSI}^{(B)}(p,\nu) = (\mathrm{PSI}(p,\nu) \cup \textstyle\bigcup_{y \in \mathrm{Ptr}_A} \mathrm{RI}_y(p,\nu), (\mathrm{ILM}(p,\nu))^{(B)})$$

Given an algorithm, it is clear that the number of different possible PDSI at all states is finite. We let ${U_A}^{(B)}$ be the set of all possible PDSI's with bound B.

The *global data-structure image (GDSI)* χ of a state $\nu = (\psi, \phi)$ with bound B, in symbols $\mathbf{GDSI}^{(B)}(\nu)$, is a mapping from ${U_A}^{(B)}$ to $[0,B]^{(\infty)}$ such that for all $\mu \in {U_A}^{(B)}$, if $\mathbf{GDSI}^{(B)}(\nu)(\mu) \neq \infty$, it means that there are exactly $\mathbf{GDSI}^{(B)}(\nu)(\mu)$ processes in ν whose PDSI's are μ; otherwise, it means that there are more than B processes in ν whose PDSI's are μ. Since GDSI's are constructed with finite set of atomic propositions and constant B, it is clear that the number of GDSI's is finite.

Notationally, we let ${X_A}^{(B)}$ be the set of all distinct GDSI's with B.

4.2 transitions among GDSI's

We define the transitions among GDSI's by visualizing transition taking place in three steps as shown in figure 3. In each transition, global pointers, local pointers of the transiting process and its references, and the local pointers of processes pointed to by global pointers may change their contents. The changes can also affect backward the PDSI's of those processes which have the above-mentioned processes as references. The first step is to identify the PDSI's, which corresponds to those processes mentioned in the last two sentences, and label them with an auxiliary process symbol T to transform the current GDSI into a *transiting GDSI (TGDSI)*. The second step is to change the variables in labeled PDSI's to calculate the new TGDSI after the transition. The third step is to

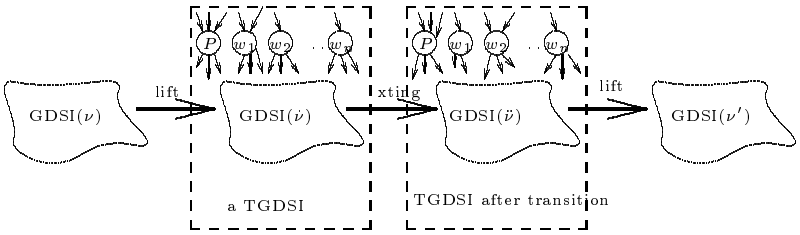

Fig. 3. transition taking place between GDSI's

discard the auxiliary symbol T and backward transforms the new TGDSI down to a new GDSI which corresponds to the global state after the transition.

We shall use T to symbolically denote the current transiting process. Let *transiting atom set* be

$$\begin{aligned}
\text{TAS} = & \{T = P\} \\
& \cup \{T \to y = P \mid y \in \text{Ptr}_A\} \\
& \cup \{T = P \to y \mid y \in \text{Ptr}_A\} \\
& \cup \{T \to y_1 = P \to y_2 \mid y_1, y_2 \in \text{Ptr}_A\} \\
& \cup \{T = w \mid w \in \text{Ptr}_S\} \\
& \cup \{T = w \to y \mid y \in \text{Ptr}_A; w \in \text{Ptr}_S\} \\
& \cup \{T \to y = w \mid y \in \text{Ptr}_A; w \in \text{Ptr}_S\} \\
& \cup \{T \to y_1 = w \to y_2 \,|y_1, y_2 \in \text{Ptr}_A; w \in \text{Ptr}_S\}
\end{aligned}$$

A *transiting process data-structure image (TPDSI)* of a process p at state ν is a tuple $(\tau \cup \theta, \beta)$ such that (θ, β) is a PDSI while τ is a subset of TAS denoting the true TAS atoms for process p at state ν right before the next transition. Intuitively, a TPDSI represents the PDSI of a process whose PDSI will be changed by the corresponding transition. $T = P$ means transition P is the transiting process. $T \to y = P$ means process P is a reference of the transiting process. $T = P \to y$ means the transiting process is a reference of process P.

A *transiting global data-structure image (TGDSI)* χ^T is a mapping from ${U_A}^{(B)} \cup \{\mu \mid \mu \text{ is a TPDSI.}\}$ to $[0, B]^{(\infty)}$. Then $\mathbf{xtion}_{\eta \to [\kappa]}(\chi, \chi')$ is defined as.

$$\mathbf{xtion}_{\eta \to [\kappa]} \equiv \exists \chi^T, \dot{\chi}^T \in {X_A}^{(B)} \exists (\theta, b) \in {U_A}^{(B)} \begin{pmatrix} \chi^T((\theta, b)) = 1 \\ \wedge \, ((\bigwedge_{a \in \theta} a) \Rightarrow (\eta \wedge T = P)) \\ \wedge \, \mathbf{lift}(\chi, \chi^T) \\ \wedge \, \mathbf{xting}_{\eta \to [\kappa]}(\chi^T, \dot{\chi}^T) \\ \wedge \, \mathbf{lift}(\chi', \dot{\chi}^T) \end{pmatrix}$$

Relation $\mathbf{lift}(\chi, \chi^T)$, defined in table 1, serves to filter out atoms with occurrences of T from χ^T and collapse χ^T down to a GDSI χ.

We can also define relation $\mathbf{xting}_{\eta \to [\kappa]}(\chi^T, \dot{\chi}^T)$ which is true if transition rule $\eta \to [\kappa]$ transforms TGDSI χ^T to another TGDSI $\dot{\chi}^T$. In table 2, we have

lift(χ, χ^T)
{
(1) Let $\chi_1 := \chi^T$;
(2) For each $\mu = ((\tau \cup \theta, b), c) \in \chi^T$ such that τ is composed of atoms solely from TAS and θ is composed of no atoms from TAS, do {
(1) Let $\chi_1 := \chi_1 - \{\mu\}$;
(2) If $((\theta, b), c') \in \chi_1$ for some $c' \in [0, B]^{(\infty)}$,
then replace it with $((\theta, b), (c + c')^{(B)})$; else add $((\theta, b, c)$ to χ_1;
}
(3) If $\chi = \chi_1$, return *true*; else return *false*;
}

Table 1. Implementation of **lift**(χ, χ^T)

xting$_{\eta \to [\alpha_1 \alpha_2 \ldots \alpha_n]}(\chi^T, \dot{\chi}^T)$
{
(1) Let $\chi_1 := \chi^T$;
(2) Sequentially for $i := 1$ to n, let $\chi_1 := \chi_1 \alpha_i$;
(3) If $\chi_1 = \dot{\chi}^T$, return *true*; else return *false*;
}

Table 2. Implementation of **xting**$_{\eta \to [\kappa]}(\chi, \chi^T)$

an implementation for relation **xting**$_{\eta \to [\kappa]}(\chi^T, \dot{\chi}^T)$. Here $\chi\alpha$ is the result of applying action α to the TPDSI's in χ. The computation of $\chi\alpha$ depends on twenty four cases based on the syntax presented in section 3. However, due to page-limit, we shall only elaborate on two of them. The rest can be done in similar although tedious reasoning.

- **case** α is $y := P$; where $y \in \mathrm{Ptr}_A$. {
  (1) Find $\mu = (\theta, b)$ with $\chi(\mu) = 1$ and $P = T \in \theta$.
  (2) If $y = P \in \theta$, return with $\chi\alpha = \chi$;
  (3) Delete any atom with y from θ.
  (4) Let $\theta := \theta \cup \{y = P\}$;
  (5) Replace θ with its transitivity closure of equivalence induced by new the new atom $y = P$.
  (6) For every $\mu' = (\theta', b')$ with $\chi(\mu') = 1$ and $P \to y_1 = T \in \theta'$, changes its θ' according to the new θ modified by the addition of $y = P$.
  }
- **case** α is $w := y$; where $w \in \mathrm{Ptr}_S$ and $y \in \mathrm{Ptr}_A$. Note in this case, global

variable w is changed. {

(1) Find $\mu_1 = (\theta_1, b_1)$ with $\chi(\mu_1) = 1$ and $P = w \in \theta_1$.

(2) If $T \to y = P \in \theta_1$, return with $\chi\alpha = \chi$;

(3) Delete all atoms related to w from all images in χ;

(4) Find $\mu_2 = (\theta_2, b_2)$ with $\chi(\mu_2) = 1$ and $T \to y = P \in \theta_2$;

(5) replace θ_2 with $\theta_2 \cup \{w = P\}$;

(6) For all $\mu_3 = (\theta_3, b_3)$ with $\chi(\mu_3) = 1$ and $P \to y_1 = T \to y \in \theta_3$, adjust elements in θ_3 to reflect that now $P \to y_1$ is pointing to w.

}

5 Safety bound verfication with CIS

By just naive enumerating all the GDSI's reachable from the initial state in an algorithm implementation, we will easily bump into combinatorial explosion of complexity because each PDSI can be mapped to any number in $[0, B]^{(\infty)}$. However, we can take advantage of our interleaving semantics to eliminate much of such complexity. The idea is based on lemma 2. A GDSI χ' contains another GDSI χ, in symbols $\chi \subseteq \chi'$, if for every PDSI $\mu \in {U_A}^{(B)}$, $\chi(\mu) \leq \chi'(\mu)$.

Lemma 2. *Suppose we have two GDSI $\chi \subseteq \chi'$ and a PDSI μ. Then for every GDSI sequence $\chi_0\chi_1 \ldots\ldots$ with $\chi_0 = \chi$, we can construct another GDSI sequence $\chi'_0\chi'_1 \ldots\ldots$ with $\chi'_0 = \chi'$ such that for all $k \geq 0$, $\chi_k \subseteq \chi_k$ and χ_k and χ'_k may go to χ_{k+1} and χ'_{k+1} respectively with the same transition rule.*

Proof : A pictorial explanation of this fact is in figure 4. The relation can happen because in a concurrent system, we can withhold those PDSI's in χ' but not in χ from firing transitions. ||

Now we shall present our approximation algorithm for safety bound problem with finite GDSI set as our CIS. We shall take advantage of lemma 2 such that two GDSI's χ, χ' will not be in the CIS simultaneously if $\chi \subseteq \chi'$.

Given a $\chi \in {X_A}^{(B)}$, $\mathbf{count}_\eta(\chi)$ is the number, respecting bound B, of PDSI's satisfying η in χ. Formally speaking,

$$\mathbf{count}_\eta(\chi) = \left(\sum_{(\theta,\beta) \in U_A^{(B)}; (\bigwedge_{a \in \theta} a) \Rightarrow \eta} \chi((\theta, \beta)) \right)^{(B)}$$

Now we have the procedure **Safety_Bound**() in table 3 to embody our safety bound verification method in details. Note in statements (2.2.2) and (2.2.3), we delete those GDSI's contained by other GDSI's in V according to lemma 2. Also, careful implementation of **lift**() is needed so that in the generation of $\bar{H}$, we don't have to go through all elements in ${U_A}^{(B)}$.

The complexity of the method is polynomial to the number of GDSI's of states which then depends on $A(P)$ and B. A rough complexity analysis follows. The equivalence relation among pointers in a PDSI basically partitions global pointers, local pointers of P, the local pointers of references of P, the local pointers of references of those process pointed to by global pointers. The total

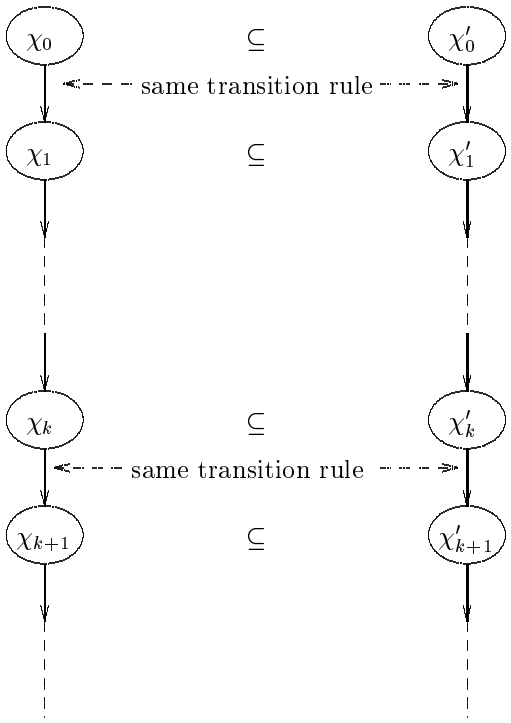

Fig. 4. Containing relation between GDSI sequences

number of pointers involved is $H = 1 + (|\mathrm{Ptr}_S| + |\mathrm{Ptr}_A|)(1 + |\mathrm{Ptr}_A|)$ which is square to the size of S. The number of different partitions on these many pointers is roughly in the complexity of factorial to H. This will be the dominating factor in the complexity. Considering the values of $\mathrm{ILM}^{(B)}$, we can deduce that the number of different PDSI is roughly $O((B+1)^{|L_A|}2^H) = O(2^{|S|^2 \log B})$. Since $GDSI$ are mappings from PDSI's to $[0,B]^{(\infty)}$, the total number of different GDSI's is then $(B+2)^{O(2^{|S|^2 \log B})} = 2^{2^{O(|S|^2 \log B + \log\log B)}} = 2^{2^{O(|S|^2 \log B)}}$. Thus our approach in each iteration of B value is of complexity doubly exponential to $|S|^2 \log B$.

For a lot of mutual exclusion protocols, small value of B like 1 will work and the CIS's exhibit simple regularity. The complexity analyses for MCS mutual-exclusion algorithm in section 6 shall justify our claim.

6 On Mellor-Crummy & Scott's algorithm

We shall prove that our method indeed can verify Mellor-Crummy & Scott's (MCS) locking algorithm[18] for mutual exclusion in concurrent systems. MCS

/* S is an algorithm with transition rule set E.
/* η is a process predicate describing the dangerous property.
/* C is the number of processes allowed in critical section.
/* B is the bound used in GDSI's. It is assumed $C \leq B$. */
Safety_Bound(S, η, C, B) {
(1) Generate the initial GDSI v_0; let $V := \{v_0\}; W := V$;
(2) Repeat until $W = \emptyset$. {
(1) Let $\bar{W} := \emptyset$;
(2) For every $v_1 \in W$, do {
(1) Calculate $\bar{H} := \{v_2 \mid v_2 \in {X_A}^{(B)}; \exists e \in E(\mathbf{xtion}_e(v_1, v_2))\}$, i.e. the set of GDSI can be reached from v in one step transition.
(2) Let $\bar{H} := \{v_3 \mid v_3 \in \bar{H}; \forall v_4 \in V(v_3 \not\subseteq v_4)\}$;
(3) Let $V := \{v_5 \mid v_5 \in V; \forall v_6 \in \bar{H}(v_5 \not\subseteq v_6)\} \cup \bar{H}$;
(4) Let $\bar{W} := \bar{W} \cup \bar{H}$;
}
(3) Let $W := \bar{W}$;
}
(3) If there is no $v \in V$ such that $\mathbf{count}_\eta(v) > C$, then report "SAFE;" else report "don't know."
}

Table 3. Safety analysis with CIS

locking algorithm is a provenly correct algorithm requiring little shared memory. We believe that our method can verify many such algorithms with small B values regardless of the number of processes.

MCS locking algorithm is an example protocol in which explicitly a queue is used. In Figure 5, a modified version of MCS locking algorithm for a process is drawn while the original version is given in figure 7 in the appendix. (Note *true* and *false* for variable `locked` is interchanged in the modified version to be consistent with our initial state restrictions.) There is one global pointer `L` to the tail of the queue. Each process has one Boolean variable `locked` and two local pointers: `next` and `prev` which respectively point to the successor and predecessor processes of the local process in the queue. We modify the algorithm by setting local pointers to null as soon as the contents of the local pointers will not be used again. This is consistent with good programming practice. For example, in our modified algorithm, when a process releases the lock, it then also set its `next` to null because it is not meant in the queue already. However, in the original algorithm, this local `next` can still outdately point to some random process. Without cautious management, such "stray" local pointers can be mistakenly used.

The following lemma shows that our method can verify the modified version.

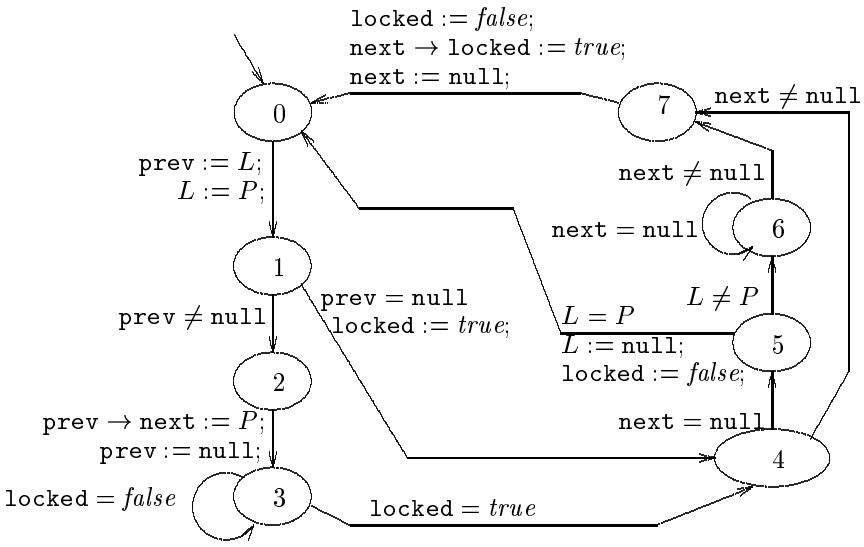

Fig. 5. modified MCS locking algorithm

A similar one can be used to prove for the original version.

Lemma 3. *: In the CIS constructed for MCS locking algorithm as shown in figure 5 with $B = 1$ and lemma 2, there is no image v with* $\mathbf{count}_{\texttt{mode}=4}(v) > 1$.
Proof : This is true because only one PDSI in a GDSI can hold `locked` = *true* with `mode` $\in [4, 7]$. To see this, we notice that only the transiting PDSI with `locked` = *true* can make its successor in the queue's `locked` true. Intially, all PDSI are in `mode` 0. Note that, the first PDSI detecting `prev` = `null` while leaving `mode` 1, will enter `mode` 4 with `locked` to `true`. Our definitions of $\mathbf{xtion}_{\eta\to[\kappa]}()$, $\mathbf{lift}()$, and $\mathbf{xting}_{\eta\to[\kappa]}()$, ensures that in any reachable GDSI, only one PDSI will have `prev=null`.

Following the definition of $\mathbf{xtion}_{\eta\to[\kappa]}()$, we find that all other PDSI's will be kept in `mode`= 0, 1, 2, or 3. Then when the PDSI corresponding to the process in critical section leaves `mode` 7, it sets the `locked` of the PDSI pointing to by its `next`. Moreover, this PDSI, say $\mathcal{P}$, pointed to by the `next` pointer is unique in the GDSI because it is either with `prev=null` or with `prev` set to a PDSI in `mode` 7. By detailed checking that our definition of $\mathbf{lift}()$ and $\mathbf{xting}_{\eta\to[\kappa]}()$ indeed ensure all the facts that we mentioned in the above true, we can then prove the lemma. ||

We want to point out that our proof for lemma 3 is very much like a human proof for MCS algorithm. This shows that our method indeed reasons at an abstractness level similar to that of humans. Now we proceed to analyze the size of the CIS with $B = 1$ for the modified MCS locking algorithm, in figure 5, which has the good property that once a process is in `mode`=0, its local pointers

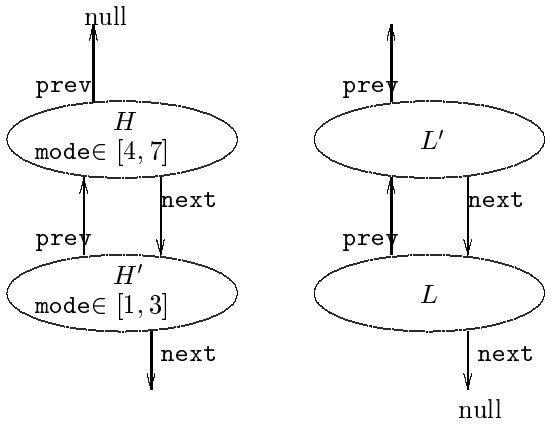

Fig. 6. PDSI pattern for modified MCS algorithm

will all be set to null and no other processes will have local pointers pointing to it again. This make the data-structure pretty much "clean" without "stray" pointers. Thus the number of different GDSI's solely depends on the different possibilities of PDSI's near the queue head, noted by H, and tail as shown in figure 6. We thus have the following case analysis.

- *In case there is only one PDSI in the queue.* Then $H \rightarrow \texttt{mode} \in \{1\} \cup [4, 7]$ and there are 5 possibilities.
- *In case there are two PDSI's in the queue.* The values of $H \rightarrow \texttt{next}$ and $L \rightarrow \texttt{prev}$ depend on $H \rightarrow \texttt{mode}$ and $L \rightarrow \texttt{mode}$. $H \rightarrow \texttt{mode} \in [1, 7]$ and $L \rightarrow \texttt{mode} \in [1, 3]$. This accounts for $7 \times 3 = 21$ possibilities.
- *In case there are three PDSI's in the queue.* Similar to the reasoning in last item, we have $7 \times 3 \times 3 = 63$ possibilities.
- *In case there are more than three PDSI's in the queue.* Assume the second PDSI in the queue is H' while the last second is L'. The values of $H' \rightarrow \texttt{next}$ and $L' \rightarrow \texttt{prev}$ depends on the modes of the third and the last third PDSI's in the queue. Also the ILM's of H' and L' also depend on the third and the last third PDSI's in the queue. Moreover, all other PDSI's will be mapped to ∞ according to lemma 2. Again, we have $7 \times 3 \times 3 \times 3 \times \times 3 = 567$ possibilities.

Summing up all the possibilities in addition to the initial GDSI, we have $5 + 21 + 63 + 567 + 1 = 657$ different GDSI's in our final CIS where the "1" represents the initial GDSI which maps the initial PDSI to ∞ and everything else to zero.

7 Conclusion

With the known worst-case complexities of most verification problems in theory, it is apparent that the current technology of model-checking is incapable of

verifying nontrivial software systems. We believe such a dilemma results from the fact that current verification theory does not distinguish "good" design from "bad" design. We argue our CIS technology is a successful example to verify well-designed concurrent systems in which relations among different PDSI groups are more important than both the actuagl numbers of processes in each PDSI group and the actual values of all pointers. We feel hopeful our technology can be extended to verify well-designed concurrent systems with other types of infinite behaviors.

References

1. R. Alur, C. Courcoubetis, D.L. Dill. Model Checking in Dense Real-Time, Information and Computation **104**, pp.2-34 (1993).
2. R. Alur, T.A. Henzinger. Real-Time Logics: Complexity and Expressiveness. Information and Computation **104**, pp.35-77 (1993).
3. K.R. Apt, D.C. Kozen. Limits for Automatic Verification on finite-state concurrent systems. Information Processing Letters, 22:307-309, 1986.
4. F. Balarin. Approximate Reachability Analysis of Timed Automata. IEEE RTSS, 1996.
5. M.C. Browne, E.M. Clarke, O. Grumberg. Reasoning about Networks with Many Identical Finite State Processes. Information and Computation **81**, 13-31, 1989.
6. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L.Dill, L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond, IEEE LICS, 1990.
7. B. Boigelot, P. Godefroid. Symbolic Verification of Communication Protocols with Infinite State Spaces using QDDs. CAV 1996, LNCS, Springer-Verlag.
8. R.E. Bryant. Graph-based Algorithms for Boolean Function Manipulation, IEEE Trans. Comput., C-35(8), 1986.
9. E. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic, Proceedings of Workshop on Logic of Programs, Lecture Notes in Computer Science 131, Springer-Verlag, 1981.
10. E. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, ACM Transactions on Programming Languages and Systems 8(2), 1986, pp. 244-263.
11. E.M. Clarke, O. Grumberg, S. Jha. Verifying Parameterized Networks using Abstraction and Regular Languages. CONCUR'95, LNCS 962, Springer-Verlag.
12. E.A. Emerson, K.S. Namjoshi. Reasoning about Rings. ACM POPL, 1995.
13. E.A. Emerson, A.P. Sistla. Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach. ACM TOPLAS, Vol. **19**, Nr. 4, July 1997, pp. 617-638.
14. S.M. German, A.P. Sistla. Reasoning about Systems with Many Processes. Journal of ACM, Vol. 39, No. 3, July 1992, pp.675-735.
15. J.E. Hopcroft, J.D. Ullman. Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.
16. R.P. Kurshan, K.L. McMillan. A Structural Induction Theorem for Processes. Information and Computation **117**, 1-11(1995).
17. D. Lesens, N. Halbwachs, P. Raymond. Automatic Verification of Parameterized Linear Networks of Processes. ACM POPL, 1997.

18. J.M. Mellor-Crummey, M.L. Scott. "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors." ACM Transactions on Computer Systems, Vol. 9, No.1, Feb. 1991, pp.21-65.
19. X. Nicolin, J. Sifakis, S. Yovine. Compiling real-time specifications into extended automata. IEEE TSE Special Issue on Real-Time Systems, Sept. 1992.
20. F. Wang. Automatic Verification of Dynamic Linear Lists for All Number of Processes. Technical Report TR-IIS-98-019, Institute of Information Science, Academia Sinica, 1998.
21. F. Wang, C.T. Lo. Procedure-Level Verification of Real-Time Concurrent Systems. to appear in Proceedings of the 3rd FME, Oxford, Britain, March 1996; in LNCS, Springer-Verlag.
22. F. Wang, A. Mok. RTL and Refutation by Positive Cycles, in Proceedings of the Formal Methods Europe Symposium, Barcelona, Spain, Octobor 1994, LNCS 873.
23. F. Wang, A.K. Mok, E.A. Emerson. Real-Time Distributed System Specification and Verification in APTL. ACM TOSEM, Vol. 2, No. 4, Octobor 1993, pp. 346-378.
24. H. Wong-Toi. Symbolic Approximations for Verifying Real-Time Systems. Ph.D. thesis, Stanford University, 1995.

A Original MCS locking algorithm

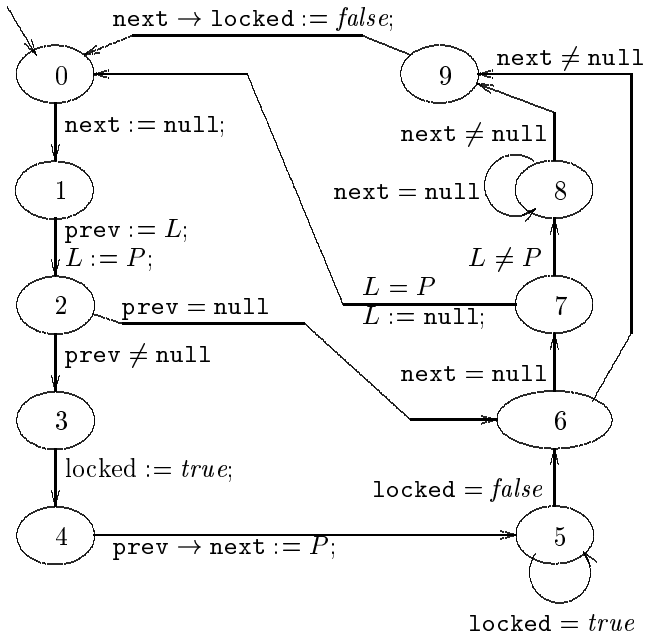

Fig. 7. original MCS locking algorithm