Environment Abstraction with State Clustering and Parameter Truncating

Hong Pan, Yi Lv, Huimin Lin
State Key Laboratory of Computer Science, Institute of Software
Chinese Academy of Sciences
P.O.Box 8718, Beijing 100190, China
{ph, lvyi, lhm}@ios.ac.cn

Abstract

Environment abstraction enriches predicate abstraction by idea from counter abstraction to develop a framework for verification of parameterized systems. However, despite various effects, the constructed abstractions still go beyond the capability of the usual model checkers for many realistic systems. In this paper, a new technique, called state clustering*, is proposed to group local states into a small number of clusters, by purely syntactic analysis. The size of array variables in the resulting abstractions are further reduced using parameter abstraction technique. By combining different abstraction techniques, real-life cache coherence protocols such as FLASH have been successfully verified.*

1. Introduction

A concurrent *parameterized system*, $\mathcal{P}(N)$, consists of N replicated processes executing concurrently. In general, besides these homogenous processes, the systems may also include a constant number of heterogeneous processes in parallel composition. Parameterized systems are common in practice, including cache coherence protocols, network protocols, communication protocols, etc. Model checking parameterized systems is challenging because they lack fixed state space. A property ϕ is satisfied by any finite number of system instances cannot guarantee it is also satisfied for the whole family of system instances. We denote by $\mathcal{P}(k)$ a concrete system instance with k replicated processes, and write $\mathcal{P}(N) \models \phi$ to mean $\forall k.\mathcal{P}(k) \models \phi$.

Environment abstraction([4], [5], [18]) combines *predicate abstraction*([6], [8]) with ideas from *counter abstraction*([15]) for verifying concurrent parameterized systems. It tries to simulate the way a human designer thinks, by focusing on the execution of some *reference* processes (typically one or two) and see how the other processes (abstracted as the *environment* for those reference processes) might interfere with their execution. Following this idea, the abstract system maintains the details on the reference processes while abstracts away the other processes by counting the number of the environment processes satisfying the given environment predicates. The abstract model is built once for all in an existential style, constituting a conservative abstraction, so that if a universal property holds in the abstract system then it also holds in the parameterized systems.

The most difficult problem with the environment abstraction technique is that the constructed abstraction may still be too complex to be tackled by the usual model checkers. For instance when the system involves any scenario in which the replicated processes contending access for a common resource, the abstraction need count the number of those environment processes on all possible control locations. Consequently the size of abstract state space increases exponentially with the number of *local states* of each replicated process. Moreover, the number of array variables may also cause the size the abstract state space to increase dramatically.

The aim of this paper is to present two techniques to make environment abstraction more effective. The first technique is to reduce the state space of local processes by *state clustering*. The idea is based on the observation that in realistic systems, for a local process, although its state space might consists of many local control locations and various message buffers, these data are closely related. Thus it is highly possible that, with respect to the property under verification, these states may fall into a small number of representative configurations. Then the local states can be partitioned into non-intersecting clusters, each represented by a *unit-state*. The clustering process is performed by a heuristic procedure based on pure syntactic analysis on the system description. Combined with eliminating unreachable local states, the method is capable of reducing hundreds of thousands local states to only tens of unit-states.

To overcome the difficulty caused by array variables, we apply *parameter truncating* technique, which is adopted from the *parameter abstraction* approach ([2], [9], [12]). The idea is only to maintain, in the environment component, the bits corresponding to the reference processes and omits the other parts. We then refine the abstraction by *invariant computing* and *guard strengthening*, until it is precise enough to prove the desired properties.

To illustrate the power of the proposed techniques, a number of case studies have been carried out. These include the FLASH and GERMAN cache coherence protocols. We use the TLV tool([17]) to perform both local reachability

978-0-7695-3757-3/09 $25.00 © 2009 IEEE
DOI 10.1109/TASE.2009.16

and invariant computing, and Cadence SMV([16]) to verify auxiliary invariants and the desired properties. For the two protocols, both cache coherence and data consistency properties are model checked. In the case of FLASH, our model keep the most characteristic "three-hop" transaction of this protocol which was omitted in all previous verification efforts because of its complexity. To our best known, this is the first time both control and data aspects of the FLASH protocol have been verified at this level of precision.

The rest of this paper is organized as follows. Related work is discussed in the remaining part of this section. In Section 2 environment abstraction and parameter abstraction are briefly reviewed. The state clustering and parameter truncating methods are presented in Section 3 where the soundness results are also stated. The results of the case studies are reported in Section 4. The paper is concluded with Section 5.

Related Work A number of abstraction based verification methods have been developed for parameterized systems in recent years. Reference [3] proposes heuristic algorithms to perform counter-example guided abstraction refinement, it also shows that the problem of finding abstraction with the least number of equivalence classes is *NP-hard*. In "*predicate abstraction*" ([6], [8]) a small abstract system is established with valuating the satisfaction on several predicates by calling decision procedures. Reachability analysis is then performed on the abstract system to generate the strongest invariant expressible over the given predicates. Lahiri and Bryant have extended this method by "*indexed predicates*" technique ([10], [11]), which uses simple predicates containing free-index variables and constructs universally quantified invariants for the system. In "*Parameter abstraction*" ([2], [9]) only a few normal nodes are maintained and all others are merged into a single abstract node, the abstract system is then refined iteratively using "non-interference lemma" generated from manual analysis of spurious counter-examples. In [12] it is proposed to generate such auxiliary invariants semi-automatically by computing invariants in a reference model; because the reference model escape the capability of TLV, this method does not work for FLASH with data path; recently[1] provides a method to generate the strongest non-interference lemma fully automatically, but it has only been applied on FLASH without data.

2. Preliminaries

A parameterized system $\mathcal{P}(N)$ consists of a small number of heterogeneous processes, called the central nodes, together with a set of homogeneous processes, called the local nodes, communicating via shared variables. In a system instance $\mathcal{P}(k)$, each replicated process has a unique index taken from the parameter set $\{1,\ldots,k\}$. To simplify the notation and without loss of generality, in this paper we shall assume there is only one central node in a parameterized system.

2.1. System Description

We call the variables maintained by the central process as *central variables*, denoted V_c, and the variables on the local processes as *local variables*, denoted V_l. Since all local processes share the same variable names, we shall distinguish them by index when necessary. Then the set of all variables of the system $\mathcal{P}(N)$ is $V = V_c \cup \bigcup_{i=1}^{N} V_{l_i}$. Variables can divided into *control variables*, which determine the internal control location of processes and usually have finite ranges, and *data variables*, which may take values from infinite domains. Variables have *types*. We take Boolean type B as the only basic type other than parameter N, since any finite enumeration type can be encoded by multiple Booleans. We call variables of type B as *simple variables*, ranged over by u, variables of parameter type N as *pointer variables*, ranged over by ptr, and *array variables* are of type $N \Rightarrow B$ or $N \Rightarrow N$, ranged over by arr. *Terms* are constructed from constants and variables by function applications. *Predicates*, or *formulas* are terms of type B. In our systems, each formula is required to be *admissible*, i.e., all atomic formulas are of the following forms: $u = 0/1$, $ptr = i$, $arr[i] = 0/1$ or $arr[i] = j$, where i, j are constants from $[1..N]$; quantifier-free formulas are Boolean combination of such atomic formulas. With conjunctive guard and broadcast communication primitives, this syntax is expressive enough to describe a wide-range of parameterized systems including those well-known cache protocols examined in our case studies.

A *system presentation* is a triple $\mathcal{P} = (V, \Theta, \Delta)$ where

- V is the set of system variables.
- $\Theta(V)$ is the *initial predicate*.
- $\Delta(V, V')$ is the set of *transition rules*, each rule $\delta \in \Delta(V, V')$ being of the form $\rho(V) \rightarrow a(V, V')$, where $\rho(V)$ is the *guard predicate* and $a(V, V')$ is the *transition relation* of the rule.

A system presentation induces a state transition system $|\mathcal{P}| = (S, I, R)$ in the usual way: S is the set of system states, where a *state* s is a type-respecting valuation of system variables V. We write $s(e)$ for the value of term e evaluated in state s, and $s \models f$ for $s(f) = true$. We will identity subsets of S with logic formulas that characterize them, called *state formulas*. I is the subset of S that satisfies Θ. For a state pair $s, s' \in S$, $(s, s') \in R$ if there exists some transition rule $\delta : \rho(V) \rightarrow a(V, V') \in \Delta(V, V')$ such that $s \models \rho(V)$ and $a(V, V')$ evaluates to $true$ when each $v \in V$ is assigned the value $s(v)$ and each $v' \in V'$ is assigned the value $s'(v')$; we will call formulas δ as *transition formulas* and write $(s, s') \models \delta$.

For directory-based cache coherence protocols as FLASH or GERMAN, the transition rules fall into two categories:

one describes the local state transition for some client node, which takes the form $\delta_i : \rho(V_{l_i}) \rightarrow a(V_{l_i}, V'_{l_i})$, while the other describe the actions of the Home node, including updating the directory, maintaining the central control information and communicating with the i_th local node via message buffers, which takes the form $\delta_H : \rho(V_{l_i} \cup V_c) \rightarrow a(V_{l_i} \cup V_c, V'_{l_i} \cup V'_c)$.

2.2. Environment Abstraction

Suppose each local process has been encoded by one control variable pc_L with control location ranging over $\{1, \ldots, T\}$. Here local processes do not maintain any pointer variable, i.e., T dos not depend on parameter N. The central process includes a variable pc_C for central control locations with range $\{1, \ldots, F\}$, some pointer variables $ptr_1, \ldots, ptr_b$, and array variables $arr_1, \ldots, arr_c$. A global state s of a system instance $\mathcal{P}(k)$ is represented as a tuple $< pc_{L_1}, \ldots, pc_{L_k};\ pc_C, ptr_1, \ldots, ptr_b, arr_1, \ldots, arr_c >$. In *environment abstraction* ([4]) only the central process and one local process, called the *reference process*, are presented in detail, while all other local processes, referred to as *environment processes*, are represented by a single component according to their state configurations on all possible local states. An abstract state $\widetilde{s}$ is denoted as a tuple $< pc_L, e_1, \ldots, e_T; pc_C, \widetilde{ptr}_1, \ldots, \widetilde{ptr}_b, \widetilde{arr}_1, \ldots, \widetilde{arr}_c >$, where the environmental bit e_i indicates whether there exists some environment process in location i; the abstract pointer variable $\widetilde{ptr}_i$ shows the abstract location in the range $\{\textbf{ref}\} \cup \{1, \ldots, T\}$, with **ref** referring to the concrete reference process. Note that to make the abstraction more precise, the environment bits e_i may be generalized to *counter* variables c_i with a wider range, e.g. $\{0, 1, 2\}$, where $c_i = 1$ means there is exactly one process in location i, and $c_i = 2$ means there are at least two processes there.

From the definition of abstract states it can be seen that even if all array variables have type $N \Rightarrow B$, the size of the abstract state space is $T * \underline{2^T} * F * (T+1)^b * \underline{2^{(T+1)*c}}$ which is exponential in the number of local states (T) and the number of array variables (c). In fact, these two factors directly determine whether the environment abstraction method could work effectively. In order to reduce the abstract state space, techniques such as eliminating unreachable environments and excluding redundant array variables have been developed. However, these techniques only have limited effects, as in complicated systems the number of reachable local states may still be too large to handle.

2.3. Parameter Abstraction

When verifying parameterized systems $\mathcal{P}(N)$ using parameter abstraction ([2], [9]), one first decides on a small number m, by a heuristic analysis of system description and property specification, and keep the description of m nodes in detail, while merging all the other nodes into an *abstract node* using a *abstraction function*. The abstract system, consisting of m regular nodes and one abstract node, is denoted as $\{P_1, \ldots, P_m, P_*\}$, where P_* maintains no local states nor internal transitions but only relates with those regular nodes via central variables. As a consequence, the abstract pointer will have value no greater than $m+1$. Any concrete variable with type constructed with parameter N will be approximated by an abstract variable with threshold m, which is called *parameter truncating*. It is highly possible that this might cause spurious counter-examples, and one can use auxiliary invariants generated manually ([2]) or semi-automatically ([12], [1]) to strengthen the guards of transition rules, until the abstract model is precise enough. If the refined abstraction satisfies the desires property as well as all the auxiliary invariants, the property will be guaranteed to hold in the original concrete system.

In the present work, we also handle array variables by parameter truncating and consider all the environment nodes as a single component. However, instead of keeping the abstract node "empty" inside, we maintain its local states as well as internal transitions. Based on the observation that these nodes are identical in execution, we need not record exactly which node is over what kind of local states, but only count the number of the nodes in the certain location using counter variables $c_1, \ldots, c_T$, i.e., under the framework of environment abstraction.

3. State Clustering and Parameter Truncating

3.1. Reachable Local States

A *local state* is a valuation for all local variables in V_l. Recall that a global state for a concrete system instance $\mathcal{P}(k) : (S_k, I_k, R_k)$ is a tuple $s = < pc_{L_1}, \ldots, pc_{L_k}; pc_C, ptr_1, \ldots, ptr_b, arr_1, \ldots, arr_c >$, where the local state of process i as $pc_{L_i} = < u_1(i), \ldots, u_p(i), ptr_1(i), \ldots, ptr_a(i) >$. A global state s^r is *reachable* if there is a sequence of states $s^0, s^1, \ldots, s^n$ satisfying $s^0 \in I_k$ and $(s^{i-1}, s^i) \in R_k$ where $i \in [1..n]$ and $s^r = s^n$. Suppose $s^r = < pc^r_{L_1}, \ldots, pc^r_{L_k}; pc^r_C, ptr^r_1, \ldots, ptr^r_b, arr^r_1, \ldots, arr^r_c >$, then each $pc^r_{L_i}$, $i \in [1..k]$, is a *reachable local state*.

In environment abstraction, we will have an environment bit (or counter) $e_{pc_{L_i}}$ for each possible local state pc_{L_i}. But usually not all of the local states are actually reachable, which makes their corresponding environment bits redundant. To exclude the unreachable local states from the abstraction, we perform local state reachability analysis first, to compute the set of local reachable states as a quantifier elimination formula from the global reachable states. In a *reference system* ([12]) $\mathcal{P}(m+1)$, we define:

$$\textit{reach}_{l_i} = \exists V_c, V_{l_1}, \ldots, V_{l_{i-1}}, V_{l_{i+1}}, \ldots, V_{l_n} : \textit{reach}(m+1)$$

where *reach*$(m+1)$ is the set of global reachable states, and *reach*$_{l_i}$ is the set of local reachable states of the i_th process. Note that because all local processes are identical, they have the same local reachable states.

In our experiences, local state reachability analysis can result in a quite dramatic reduction on the size of local states: usually from hundreds of thousands to about one hundred. However, even after eliminating those environment bits corresponding to the unreachable local states, the abstraction is still too large to handle. To further reduce the size of the local state space, we introduce a new technique called *state clustering*, to be explained in the following sub-section.

3.2. State Clustering

The ideal of state clustering is based on the observation that the internal control status and the various message buffers on each local process are usually closely related, probably falling into a small number of representative configurations such that the local states in each configuration have the same behavior with respect to the property to be verified. Therefore we may accordingly group such states into a cluster represented by a sample state called *unit-state*.

Although we do not have precise rules for state clustering, there are some general principals and heuristics which can be followed in practice.

First, when generating the transition graph for local process i, two kinds of transitions can be distinguished: a transition of the first kind is induced by the local process itself, updating its control location to a new position, while a transition of the second kind is induced by the central process communicating with the process i via shared message buffers. Thus a transition $(pc_{L_i}, pc'_{L_i}) \in E^c_{l_i}$ usually involves updating the central state simultaneously, i.e., it affects the central control status, therefore we tend to distinguish them and group such local states into different clusters. For each sub-graph $G'_i = (S_{l_i}, E^l_{l_i})$, we will consider every connected component separately and take the nodes in the same component for further clustering. Note that because all the local processes are homogenous, their transition graphs are isomorphic.

Second, for each local variables on a local process, we assign it a priority according to their role and importance in the system. To this end, we divide V_l into two subsets V_{ll} and V_{lb}, V_{ll} including variables indicating local control locations while V_{lb} containing shared message buffers. Generally speaking, control variables have higher priority than message buffers because the desired properties are significantly dependent on control locations, such as cache-state on local processes in cache coherence protocol. The priorities of variables in V_{lb} are determined by analyzing transition rules: the more central states a variable affects, the higher priority it has. To estimate the influences of local message buffers on the central states, the dominating-set $\mathcal{DM}(V_c/V_{lb})$ of central variables over local message buffers is generated as follows: when a local buffer occurs in the right-hand side of an assignment or condition of some central variable, its weight is incremented. By scanning the transition rules, all buffer variables can be ordered according to their frequency of reference in communications. Similarly, the dominating-set $\mathcal{DM}(V_{lb}/V_{ll})$ can be computed, so that variables in V_{ll} are weighed by their influences on local buffers which transfer such influences onto central variables. Now the concatenation of (the ordered versions) of V_{ll} and V_{lb}, in that order, gives the priority ordering for all local variables.

Finally, inside each local connect component, we merge local states which differ only in their values on the variables of lowest priority into a cluster. In case the resulting transition system is still too large then we further merge those

```
Generate local state transition graph $G_i = (S_{l_i}, E_{l_i})$
  let $S_{l_i} = reach_{l_i}$;
  let $E_{l_i} = E^l_{l_i} \cup E^c_{l_i}, E^l_{l_i} \cap E^c_{l_i} = \emptyset$;
    for any $pc_{L_i}, pc'_{L_i} \in S_{l_i}$
      if exists $\delta_i : \rho(V_{l_i}) \rightarrow a(V_{l_i}, V'_{l_i}) \in \Delta$
                 and $(pc_{L_i}, pc'_{L_i}) \models \delta_i$
      then $(pc_{L_i}, pc'_{L_i}) \in E^l_{l_i}$;
      if exists $\delta_H : \rho(V) \rightarrow a(V, V') \in \Delta$
                 and $\rho(V) = \rho_1(V_{l_i}) \wedge \rho_2(V - V_{l_i})$
         and $a(V, V') = a_1(V_{l_i}, V'_{l_i}) \wedge a_2(V - V_{l_i}, V' - V'_{l_i})$
            and $(pc_{L_i}, pc'_{L_i}) \models \delta'_H : \rho_1(V_{l_i}) \rightarrow a_1(V_{l_i}, V'_{l_i})$
      then $(pc_{L_i}, pc'_{L_i}) \in E^c_{l_i}$;
  return sub-graph $G'_i = (S_{l_i}, E^l_{l_i})$ and its connected
          component
Compute the priority for local variables $V_l$
  let $V_l = V_{ll} \cup V_{lb}, V_{ll} \cap V_{lb} = \emptyset$;
  procedure dominating-set($V_A, V_B$)
    for each $\delta : \rho \rightarrow a \in \Delta$
      if atomic formula of $a$ as $v'_j = e_j$ with $v_j \in V_A$
         and any occurrence of variable $v_i \in V_B$ in
         formula $\rho$ and $v_r \in V_B$ in term $e_j$
      then $v_i, v_r \in \mathcal{DM}(V_A/V_B)$;
           add $weight(v_i), weight(v_r)$ by one;
    return elements of $\mathcal{DM}(V_A/V_B)$ in sequence
          with weight from big to small
  merge the result of $\mathcal{DM}(V_{lb}/V_{ll})$ and $\mathcal{DM}(V_c/V_{lb})$
Construct unit-states
  for each connected component in $G'_i = (S_{l_i}, E^l_{l_i})$
    enumerate its elements in sequence according to
           priority on $V_l$;
    cluster those adjacent local states into one unit-state
           with different values merely on variables of
           lower priority
```

Figure 1. The procedure of local state clustering

states with different values on the variables of second lowest priority, and so on, until model checking can go through.

The pseudo-code implementing the above three steps is presented in Figure 1. In our experiments, to be reported in Section 4, the clustering method can classify hundreds of reachable local states into only tens of unit-states.

Suppose the unit-state set is given as $\{S_j | j \in [1..t]\}$, we define an operator $\mathcal{MS}$ that maps a reachable local state to its representative unit-state: $\mathcal{MS}(pc_{L_i}) = S_j$ with $pc_{L_i} \equiv \bigwedge_{l,m=1,1}^{p,a}(u_l(i) = e_l \wedge ptr_m(i) = r_m)$, where $i \in [1..k]$ and $j \in [1..t]$.

Under such an environment abstraction with state clustering, the abstract state is denoted as $\tilde{s} = < \widetilde{pc_L}, e_1, \ldots, e_t; pc_C, \widetilde{ptr}_1, \ldots, \widetilde{ptr}_b, \widetilde{arr}_1, \ldots, \widetilde{arr}_c >$, where $\widetilde{pc_L} = < u_1(w), \ldots, u_p(w); \widetilde{ptr}_1, \ldots, \widetilde{ptr}_a >$ (we assume a constant w and take the w_th process as the reference process), with abstract variables defined as follows:

1) *environment bits*

$$e_j = 1 \text{ iff there exists some } \mathcal{MS}(pc_{L_i}) = S_j,$$

$$j \in [1..t], i \in [1..k], i \neq w$$

2) *abstract pointers* Each abstract pointer $\widetilde{ptr_i}$ either refers to the reference process with the special index **ref**, or represents the matched unit-state corresponding with the location pointed by ptr_i, i.e.,

$$\widetilde{ptr_i} = \begin{cases} \textbf{ref} & \text{if } ptr_i = w \\ j & \text{if } \mathcal{MS}(pc_{L_{ptr_i}}) = S_j, \\ & j \in [1..t], ptr_i \neq w \end{cases}$$

3) *abstract arrays* Let $E = \textbf{ref} \cup [1..t]$ be a new type. An array variables arr_i with type $N \Rightarrow B$ or $N \Rightarrow N$ are abstracted into $\widetilde{arr_i}$ with type $E \Rightarrow B$ or $E \Rightarrow E$, respectively. More specifically, if arr_i has type $N \Rightarrow B$, then

$$\widetilde{arr_i} : \begin{cases} \widetilde{arr_i}[\textbf{ref}] = arr_i[w] & \\ \widetilde{arr_i}[j] = arr_i[ptr] & \text{if } \mathcal{MS}(pc_{L_{ptr}}) = S_j \\ & j \in [1..t], ptr \neq w \end{cases}$$

If arr_i has type $N \Rightarrow N$, then (where $j \in [1..t]$)

$$\widetilde{arr_i} : \begin{cases} \widetilde{arr_i}[\textbf{ref}] = \textbf{ref} & \text{if } arr_i[w] = w \\ \widetilde{arr_i}[\textbf{ref}] = j & \text{if } arr_i[w] \neq w, \\ & \mathcal{MS}(pc_{L_{ptr_{arr_i[w]}}}) = S_j \\ \widetilde{arr_i}[j] = \textbf{ref} & \text{if } arr_i[l] = w, l \neq w, \\ & \mathcal{MS}(pc_{L_{ptr_{arr_i[l]}}}) = S_j \\ \widetilde{arr_i}[j] = j & \text{if } arr_i[l] \neq w, l \neq w, \\ & \mathcal{MS}(pc_{L_{ptr_{arr_i[l]}}}) = S_j \end{cases}$$

Theorem 3.1 (Soundness): Let $\mathcal{P}(k) : (S_k, I_k, R_k)$ be a parameterized system and $\widetilde{\mathcal{P}}$ its environment abstraction with state clustering as defined above. For any **ACTL*** formula ϕ, $\widetilde{\mathcal{P}} \models \phi(\textbf{ref}) \Longrightarrow \mathcal{P}(N) \models \forall x.\phi(x)$.

3.3. Parameter Truncating

In environment abstraction, array variables are eliminated as much as possible to reduce the size of the abstract state space. However, as noted in sub-section 2.2, the remaining abstract array variables may still cause the abstract state space growing too large. To further cut down the size of array variables, we adopt the parameter truncating technique from parameter abstraction, briefly reviewed in sub-section 2.3, and apply it to pointer variables and array variables.

Let $M = \{1, \ldots, m\}$ be a subset of N, and $M_* = M \cup \{\textbf{env}\}$, where m is the number of reference processes reserved in environment abstraction. To choose m representatives out of the N processes, let $\textbf{in} : M \to N$ be an injection and $Imag[\textbf{in}] = \{\textbf{in}(x) | x \in [1..m]\}$ (for symmetric systems, **in** can be the identity function, i.e., the first m processes are chosen as references). For any $i \in [1..N]$, define $\widehat{\textbf{in}}(i)$ to be $\textbf{in}^{-1}(i)$ if $i \notin Imag[\textbf{in}]$, otherwise $\widehat{\textbf{in}}(i) = \textbf{env}$. For each type Tp we define an abstract type $[Tp]$ by letting $[B] = B$, $[N] = M_*$, $[N \Rightarrow B] = M \Rightarrow B$ and $[N \Rightarrow N] = M \Rightarrow M_*$.

The abstract transition system $|\mathcal{P}^{\mathcal{A}}| : (S^{\mathcal{A}}, I^{\mathcal{A}}, R^{\mathcal{A}})$ involves three parts: the central process, the reference processes and the environment component; an abstract state is represented as a tuple $\widehat{s} = < \widehat{pc}_{L_1}, \ldots, \widehat{pc}_{L_m}, c_1, \ldots, c_t; \widehat{pc}_C, \widehat{ptr}_1, \ldots, \widehat{ptr}_b, \widehat{arr}_1, \ldots, \widehat{arr}_c >$, where $\widehat{pc}_{L_i} =< \widehat{u}_1(\textbf{in}(i)), \ldots, \widehat{u}_p(\textbf{in}(i)); \widehat{ptr}_1(\textbf{in}(i)), \ldots, \widehat{ptr}_a(\textbf{in}(i)) >$ and t is the number of unit-states. Then the abstraction map $\alpha : S \to S^{\mathcal{A}}$ can be defined:

$$\alpha(s) = \widehat{s} : \begin{cases} \alpha(s)(\widehat{u}) = s(u) & \\ \alpha(s)(\widehat{ptr}) = \widehat{\textbf{in}}(s(ptr)) & \\ \alpha(s)(\widehat{arr}[i]) & \text{if } i \in [1..m], \\ \quad = s(arr[\textbf{in}(i)]) & arr : N \Rightarrow B \\ \alpha(s)(\widehat{arr}[i]) & \text{if } i \in [1..m], \\ \quad = \widehat{\textbf{in}}(s(arr[\textbf{in}(i)])) & arr : N \Rightarrow N \\ \alpha(s)(c_i) = 1/0 & \text{whether exists} \\ \quad \mathcal{MS}(pc_{L_j}) = S_i, i \in [1..t], & j \notin Imag[\textbf{in}] \end{cases}$$

Now we are ready to construct abstraction operators A_F, A_N and A_R which perform syntactic transformation on formulas, assignments and transition rules, respectively. Note that such syntactic transformations involve two aspects, one performs parameter truncating for the reference processes and the central process, and the other implements state clustering for the environment component. Accordingly, the abstraction operator A_F, A_N will be defined by two sub-operators. Let ϕ be a formula which is a Boolean combination of atomic formulas $\phi_1, \ldots, \phi_n$ such that negation is only applied to atomic formulas. Then $A_F^1(\phi)$ is the same Boolean combination of the atomic formulas $A_F^1(\phi_1), \ldots, A_F^1(\phi_n)$, where (a variable v_k with

$k \in [1..a]$ is abbreviated v_{1-a})

$$A^1_F(\phi_i) \doteq \begin{cases} \phi_i & \text{if } \phi_i \equiv u_{1-p}(j) = 0/1 \\ & \text{or } \phi_i \equiv ptr_{1-a}(j) = j' \\ & \text{or } \phi_i \equiv u_{1-q} = 0/1 \text{ or } \phi_i \equiv ptr_{1-b} = j \\ & \text{or } \phi_i \equiv arr_{1-c}[j] = 0/1 \\ & \text{or } \phi_i \equiv arr_{1-c}[j] = j' \\ & \text{where} j, j' \in Imag[\textbf{in}]; \\ 1 & \text{otherwise and } \phi_i \text{ occurs positively in } \phi \\ 0 & \text{otherwise and } \phi_i \text{ occurs negatively in } \phi \end{cases}$$

For formulas on environment component, define $A^2_F(op(\phi_1, \ldots, \phi_n)) \doteq op(A^2_F(\phi_1), \ldots, A^2_F(\phi_n))$, where $A^2_F(\phi_i) \doteq \bigvee_{r=1}^{k}(c_{j_r} = 1)$ if $\mathcal{MS}(\phi_i) = \{S_{j_1}, \ldots, S_{j_k}\}$, $j_r \in [1..t]$, with $\phi_i \equiv (u_{1-p}(l) = 0/1) \wedge (ptr_{1-a}(l) = l')$ and $l \notin Imag[\textbf{in}]$. For any ϕ, construct $\phi = \phi_1 \wedge \phi_2$ where ϕ_2 is an expression over local variables V_{l_i}, $i \notin Imag[\textbf{in}]$, and ϕ_1 is an expression over the remainder variables $V - V_{l_i}$. The general abstraction operator on formulas is defined as $A_F(\phi) \doteq A^1_F(\phi_1) \wedge A^2_F(\phi_2)$.

For a transition rule $\delta : \rho(V) \rightarrow a(V, V')$, let $\rho(V) = \rho_2(V_{l_i}) \wedge \rho_1(V - V_{l_i})$ and $a(V, V') = a_2(V_{l_i}, V'_{l_i}) \wedge a_1(V - V_{l_i}, V' - V'_{l_i})$, $i \notin Imag[\textbf{in}]$. Without loss of generality, assuming $A_F(\rho_2(V_{l_i})) = (c_l = 1)$ and $A_F(\mathit{UnPrime}(a_2(V_{l_i}, V'_{l_i}))) = (c_r = 1)$, where the operator *UnPrime* eliminates primes on variables, then the relevant assignment transformation is defined as $A^2_N(a_2) \doteq (c'_l = 0 \vee c'_l = 1) \wedge (c'_r = 1)$. Let $A^1_N(a_1) \doteq \mathit{Prime}(A^1_F(\mathit{UnPrime}\ (a_1(V - V_{l_i}, V' - V'_{l_i}))))$, the general abstraction operator on assignments is defined by $A_N(a) \doteq A^1_N(a_1) \wedge A^2_N(a_2)$.

Finally the abstraction operator on transition rules are defined by $A_R(\delta) \doteq A_F(\rho) \rightarrow A_N(a)$. By setting $\Theta_m = A_F(\Theta)$ and $\Delta_m = \{A_R(\delta) | \delta \in \Delta\}$ we obtain the desired abstraction $\mathcal{P}^{\mathcal{A}} = (V \cup \{c_1, \ldots, c_t\}, \Theta_m, \Delta_m)$.

As explained in [2], the abstraction $\mathcal{P}^{\mathcal{A}}$ might need further refinement by guard strengthening with non-interference lemma to exclude spurious counter-examples. This can be done semi-automatically by computing invariants in a reference model [12]. The soundness of parameter truncating is stated below:

Theorem 3.2: Suppose $\varphi(m+1, j)_\delta = \exists V(1), \ldots, V(j-1), V(j+1), \ldots, V(m+1) : (reach \wedge \rho(m+1)_\delta)$ computed in reference system $P(m+1)$, and $\psi(m+1, j)_\delta = \rho(m+1)_\delta \rightarrow \varphi(m+1, j)_\delta$, where $j \in [1..m]$. Let $\mathcal{Q}$ be the abstract system obtained from $\mathcal{P}^{\mathcal{A}}$ by replacing each rule $\delta' : A_F(\rho(m+1)) \rightarrow A_N(\mathbf{a})$ with $A_F(\rho(m+1)_\delta \wedge \bigwedge_{j=1}^{m} \varphi(m+1, j)_\delta) \rightarrow A_N(\mathbf{a})$. Then, for any admissible **ACTL*** formula ϕ, $\mathcal{Q} \models A_F(\phi) \wedge \bigwedge_{j=1}^{m} \psi(m+1, j)$ implies$\mathcal{P}(N) \models \forall x.\phi(x)$.

4. Case Studies

Several case studies have been carried out to illustrate the effectiveness of the proposed techniques. Due to space limitation only the result of one case study, the FLASH cache coherence protocol, is reported in some details.

Unit-	proc			UniMsg	
State	.CacState	.InvMark	.Cmd	.Cmd	.proc
S1	Exclusive	0	NN	UN	0,1
S2	Shared	0	NN	UN	0,1
S3	Invalid	1	Nget	Uget	0
S4	Invalid	1	Nget	Uget	1
S5	Invalid	1	Nget	Uput	0,1
S6	Invalid	1	Nget	Unak	0
S7	Invalid	1	Nget	Unak	1
S8	Invalid	0	NN	UN	0,1,2
S9	Invalid	0	Nget	Uget	0
S10	Invalid	0	Nget	Uget	1
S11	Invalid	0	Nget	Uput	0,1
S12	Invalid	0	Nget	Unak	0
S13	Invalid	0	Nget	Unak	1
S14	Invalid	0	NgetX	UgetX	0
S15	Invalid	0	NgetX	UgetX	1
S16	Invalid	0	NgetX	Unak	0,1
S17	Invalid	0	NgetX	UputX	0,1
S18	Invalid	1	Nget	Uget	2
S19	Invalid	0	Nget	Uget	2
S20	Invalid	0	NgetX	UgetX	2

Figure 2. FLASH State clustering: unit-state

FLASH is a realistic cache coherence protocol used in industry([13], [14]). As a real-life protocol, it has complicated transition rules and uses complex variables. It has been a target for various verification techniques, including parameter abstraction and environment abstraction. However, only the control part of the protocol is considered in these efforts. The data aspect has been excluded to make the size of the problem manageable to the existing model checkers. Even for the control part, the so-called "three-hop transaction", which is the most important feather of the protocol, has also been abstracted away. It is fair to say that this protocol has so far not been verified with its full features. Thus the protocol serves as a good test for the power of the techniques proposed in this paper.

Two attempts have been made in our case studies. The first is to verify the control part of FLASH with three-hop transaction, and the second is to extend it with data.

In the first attempt, we performed local reachability analysis, which generates 74 reachable local states out of the 1944 possible valuations on all the local variables. By state clustering, the size of the abstract node is further cut down to only 17, as shown in Figure 2.

By environment abstraction, we have three kinds of entities in the abstract model: the central process ($Node[0]$), the reference process ($Node[1]$) and the environmental component ($Node[2]$). The verification is carried out using the Cadence SMV model checker. Each transition rule is implemented by an asynchronous module. Note that, since the requester and the owner might be the abstract node itself, 3 extra unit-states, S18 - S20, are added, as shown in the bottom of Figure 2. The abstraction for the control part of FLASH is rewritten with 20 unit-states and counter threshold 2; this has been precise enough to verify cache coherence property. The abstraction got model checked on a 3.2GHz Pentium4 PC with 4GB memory running Linux. The result turned to be true. The CPU time used is 527.53s and the memory cost is 114M.

In the second attempt, the model is extended with data. Accordingly we extend each unit-state with data fields on cache and relevant message buffers. To keep the abstraction simple, we let the data domain contain only two elements, 0 and 1. Note that only when a process has exclusive or shared cache state, or is permitted to access, the data content is significant. Based on this observation, we merely refine unit-states $\{S1, S2, S5, S11, S17\}$ on data fields and their counterparts $\{S21, \dots, S25\}$ by assigning with an appropriate data value, as shown in Figure 3. The desired data consistency property includes two aspects: one is about the freshness of data content on the memory, and the other is about the freshness of data content on all local caches. Using TLV, we generate 72 auxiliary invariants by invariant computing in a reference model which includes one home node and two regular client nodes. After guard strengthening with these invariants, the abstraction is precise enough, since both the formula representing the cache coherence property and formula representing the data consistency property turn out to be true. The CPU time used is 57545.7s and the memory cost is 2.37G.

We also verified the GERMAN 2000 and the GERMAN 2004 protocols ([2], [7], [12]), which are much easier. Their local states are clustered into 9 and 18 unit-states respectively. With 11 and 61, respectively, auxiliary invariants generated for guard strengthening, the two abstractions got model checked with respect to both cache coherence and data consistency. The time and space used for GERMAN 2000 are 0.875s and 6.2M, respectively, and for GERMAN 2004 these numbers are 8125.1s and 965.1M.

5. Conclusion

We have presented two techniques, state clustering and parameter truncating, to enhance the environment abstraction method. Combined with local reachability analysis, state clustering is capable of reducing the size of an abstract node by an order of 3. The idea of parameter truncating is adopted from parameter abstraction. Used in the environment abstraction framework, it effectively cuts down the sizes of array variables which are very sensitive to the overall size of the abstract models. The effectiveness of the techniques have been illustrated with a number of case studies. We have been able to verify the FLASH cache coherence protocol with data and the three-hop transaction feature. To the best of our knowledge, it is the first time that this industrial protocol is verified at this level of precision. Moreover, except for heuristics-based state clustering, the whole verification process is rather mechanical and requires little human intervention.

Unit-	proc				UniMsg		
State	.CS	.IM	.Cmd	.Dt	.Cmd	.proc	.Dt
S1	E	0	NN	0	UN	0,1	-
S2	S	0	NN	0	UN	0,1	-
S3	I	1	Nget	-	Uget	0	-
S4	I	1	Nget	-	Uget	1	-
S5	I	1	Nget	-	Uput	-	0
S6	I	1	Nget	-	Unak	0	-
S7	I	1	Nget	-	Unak	1	-
S8	I	0	NN	-	UN	0,1,2	-
S9	I	0	Nget	-	Uget	0	-
S10	I	0	Nget	-	Uget	1	-
S11	I	0	Nget	-	Uput	-	0
S12	I	0	Nget	-	Unak	0	-
S13	I	0	Nget	-	Unak	1	-
S14	I	0	NgetX	-	UgetX	0	-
S15	I	0	NgetX	-	UgetX	1	-
S16	I	0	NgetX	-	Unak	0,1	-
S17	I	0	NgetX	-	UputX	-	0
S18	I	1	Nget	-	Uget	2	-
S19	I	0	Nget	-	Uget	2	-
S20	I	0	NgetX	-	UgetX	2	-
S21	E	0	NN	1	UN	0,1	-
S22	S	0	NN	1	UN	0,1	-
S23	I	1	Nget	-	Uput	-	1
S24	I	0	Nget	-	Uput	-	1
S25	I	0	NgetX	-	UputX	-	1

Figure 3. FLASH unit-state extended with data

As future work, we would like to investigate a more precise and automatic way to performing state clustering. Besides, our clustering strategy is rather conservative in that we first partition local state into small sets and then perform clustering within individual sets, but in fact sometimes local state in different sets can also be merged to constitute a unit-state, which will definitely get a smaller unit-state set. Therefore an alternative way might be clustering as much as possible at first, and then splitting some unit-state into more precise ones when necessary.

Acknowledgment This work is supported by the National Natural Science Foundation of China (Grants No.60721061 and No.60833001) and the National High Technology Development 863 Program of China (Grant No.2007AA01Z147).

References

[1] J.Bingham(2008), “Automatic non-interference lemmas for parameterized model checking”. In *FMCAD’08*, Portland, OR, USA, in press.

[2] C. T. Chou, P. K. Mannava, and S. Park(2004), “A simple method for parameterized verification of cache coherence protocols”. In *FMCAD’04*, Austin, Texas, USA, volume 3312 of LNCS, pages 382-398. Springer-Verlag, 2004.

[3] E. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith(2003), “Counterexample-guided abstraction refinement for smybolic model checking”. In *Journal of the ACM*, **50**, pages 752-794.

[4] E. Clarke, M. Talupur, and H. Veith(2006), “Environment abstraction for parameterized verification”. In *VMCAI’06*: Verification, Model Checking, and Abstract Interpretation, pages126-141, 2006.

[5] E. Clarke, M. Talupur, and H. Veith(2008), “Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems”. In *TACAS’08*: Tools and Algorithms for the Construction and Analysis of Systems, volume 4963 of LNCS, pages 33-47, 2008.

[6] S. Das, D. Dill, and S. Park(1999), “Experience with predicate abstraction”. In *CAV’99*, Trento, Italy, volume 1633 of LNCS, pages 160-171. Springer-Verlag, 1999.

[7] S. German, G. Janssen(2004), Tutorial on verification of distributed cache memory protocols. In *FMCAD’04*.

[8] S. Graf, and H. Saidi(1997), “Construction of abstract state graphs with PVS”. In *CAV’97*, Haifa, Israel, volume 1254 of LNCS, pages 72-83. Springer-Verlag, 1997.

[9] S. Krstic(2006), “Parameterized system verification with guard strengthening and parameter abstraction”. In *AVIS’05*, ENTCS, in press.

[10] S.K. Lahiri and R. Bryant(2004), “Indexed predicate discovery for unbounded system verification”. In *CAV’04*, Boston, Massachusetts, USA, volume 3114 of LNCS, pages 135-147. Springer-Verlag, 2004.

[11] S.K. Lahiri and R. Bryant(2004), “Constructing quantified invariants via predicate abstraction”. In *VMCAI’04*, Venice, Italy, volume 2937 of LNCS, pages 267-281. Springer-Verlag, 2004.

[12] Yi Lv, Huimin Lin, and Hong Pan(2007), “Computing Invariants for Parameter Abstraction”. In *MEMOCODE’2007*, the Fifth ACM/IEEE International Conference on Formal Methods and Models for Codesign, Page(s):29 - 38, May 30 2007-June 2, 2007.

[13] K. McMillan(2001), “Parameterized verification of the FLASH cache coherence protocol by compositional model checking”. In *CHARME’01*, volume 2144 of LNCS, pages 179-195. Springer-Verlag, 2001.

[14] S. Park and D. L. Dill(1996), “Verification of FLASH cache coherence protocol by aggregation of distributed transactions”. In *SPAA’96*, pages 288-296, Padua, Italy. ACM Press, 1996.

[15] A. Pnueli, J. Xu, and L. Zuck(2002), “Liveness with $(0,1,\infty)$ Counter Abstraction”. In *CAV’02*, Proceeding of the 14th International Conference on Computer Aided Verification, 2002.

[16] http://www.kenmcmil.com/smv.html

[17] http://www.wisdom.weizmann.ac.il/ verify/tlv/

[18] M.Talupur(2006), “Abstraction Techniques for Parameterized Verification”. In *ph.d. thesis*, School of Computer Science, Carnegie Mellon University, November, 2006.