Model Checking A Parameterized Directory-based Cache Protocol

Allen E. Emerson, Arun Venkataramani

Abstract

We present the mechanical verification of a parametrized distributed directory-based cacshe protocol, that had been defying naive attempts at mechanical verification for quite some time. The verification of the protocol was accomplished by applying a bunch of known techniques like symetry reduction, repetetive constructors and state abstraction. The clustering of the reachable state space was done manually, guided by counterexamples, and the original program rewritten so as to contain only 14 local client states as opposed to the original 288. These abstractions caused the reduction of the size of the global state space graph to only 1277 and resulted in the successful mechanical verification of the protocol.

1 Introduction

In this paper, we present the verification of a parameterized directory-based cache protocol by model checking. Model checking [3, 4] is an algorithmic method for determining whether a finite state system satisfies a temporal logic formula f. However, naive strategies to model check real system designs are often impractical because of the state explosion problem. Though the complexity of model checking is dominated by $|M|$, *i.e.* it is linear in the size of the program state space, $|M|$ itself may be exponentially larger than the textual description of M. For example, a cache protocol consisting of a central server and n clients, each of which could possibly be in one of 5 local states is 5^n.

An important technique to alleviate this state explosion problem is abstraction. Abstraction works by clustering the states of a program and performing a syntactic transformation on it, so as to rewrite the original program into one with a smaller state space. It can also construct a finite state abstract program from a given possibly infinite state *concrete* program. In general, an abstract program is always guaranteed to be a conservative approximation of the original with respect to the initial set of predicates AP, *i.e.,* when a specification is true in the abstract model, it will also be true in the concrete model. However, the converse is not true in general. This means that the abstraction has to be further refined to eliminate the *false* counterexample.

A large number of systems such as cache coherence protocols, communication protocols etc are parameterized, *i.e.,* their description gives a family of different systems each member of which has a different number of replicated identical components. Symmetry reduction is a form of abstraction that is often employed to verify the entire family of systems independent of the exact number of replicated components. Synchronization and coordination protocols are often specified in the form of a parallel composition of n processes, identical up to renaming. Such high degree of symmetry can be exploited to compactly represent the state space of the program in an abstracted or symmetry reduced form. Consider for instance, a mutual exclusion protocol that contains the states (C_1, T_2, T_3), (T_1, C_2, T_3) and (C_1, C_2, T_3), representing state where process $1, 2$ and 3 are resp. in the critical section and the other two processes are attempting to enter the critical section. The three states are identical up to a permutation of the process indices. Such isomorphic (up to a permutation) states may be aggregated into one meta-state, thereby factoring out the symmetry. Model checking this symmetry reduced structure may thus result in substantial savings.

Factoring out the symmetry in a system parameterized by the number of replicated identical components, represented by the parameter n still doesn't directly enable us to model check the complete family of program instances for all n. This is because the state space of such a parameterized system is essentially infinite. In order to overcome this hurdle, one may use the method of *truncated counters*, also introduced previously by Dill et al. as *repetitive constructors* [9] in the context of Murϕ. Truncated counters go one step beyond symmetry reduction in abstracting the program state space. The latter factors out symmetry by maintaining only a count of the number of components n_{s_k} in a local component state s_k, instead of actually remembering which state each component is in. By using truncated counters, with each local component state s_k, there is associated a number t_{s_k} such that any program state with more than t_{s_k} components in the local component state s_k is considered equivalent to the corresponding program state with exactly t_{s_k} components in s_k. This technique can, independent of the parameter n - the number of replicated identical

components, represent the entire infinite family of parameterized program instances as a single abstracted *finite* state machine.

The verification of the protocol under consideration was accomplished by a combination of the techniques mentioned above. The protocol specification consists of a central server and n clients, for an arbitrary n. We manually cluster the reachable local client states into 14 states. This state clustering was done manually and required considerable thought and insight into the protocol specification. Then we used symmetry reduction and truncated counters to rewrite the protocol into an abstracted form. Finally we model checked this abstracted protocol, resulting in the generation of a state space of size only 1277. The correctness or coherence property was verified to hold in each one of these states. The correctness of the original concrete protocol specification thus follows. We mention here that though heuristics for automated abstraction have been recently discussed in the literature [2], at the time this problem was being worked upon, no tools for performing automatic abstraction were publicly available.

The rest of this paper is organized as follows. In section 2 we describe related research involving abstraction techniques used in this paper. Section 3 introduces the program variables used to describe the protocol. The actual protocol specification itself is included in the appendix for aesthetic reasons. In section **??** we describe the verification methodology in greater detail and describe the constraints used to cluster the local client states. Sections **??** describe the verification of the abstracted program and conclude the correctness of the original protocol therefrom. The appendix also carries a brief description of the hand proof of the protocol.

2 Related Work

The general problem of verifying systems with replicated components is known to be undecidable [13, 11]. Some induction based approaches proposed in [14, 5, 6, 19] for verifying particular classes of problems require an invariant process or a network invariant. The generation of such invariants is non-trivial and it's automation is restricted and expensive [17, 1, 18, 11].

The idea of exploiting symmetry to reduce the size of the state space in automatic verification is well known [7, 16, 15]. Lubachevsky performed the verification of a parameterized concurrent program by collapsing all reachable states into a fixed number of *metastates*. Dijkstra used regular expressions to represent classes of similar states. Pong et al. used a set of repetition constructors to abstract away the exact number of components, for the verification of cache coherence protocols. In [9, 8] Ip et al. present the *RepetetiveId* data type incorporated in Murϕ that is used to model programs with symmetric components obeying a certain set of restrictions. Murϕ also uses the idea of *repetition constructors* to abstract away the exact number of components in a particular local component state. However, even Murϕ is still unable to verify the protocol under consideration because of the large number of possible local states in which each of the symmetric clients could be. In spite of factoring out the symmetry and using *repetitive constructors*, the complexity of the verification process is exponential in the number of local client states. Hence, further abstraction is necessitated for protocols with even a modest number of local client states.

Abstraction based verification algorithms have been described earlier by Graf [12] and applied to a distributed cache memory. Automatic abstraction techniques have recently been proposed in the literature [12]. Namjoshi et al. give an algorithm that constructs a finite state abstract program from a given possibly infinite concrete program by means of a *syntactic* program transformation. It works by starting with an initial set of predicates from the correctness properties, and then iteratively computes the full set of predicates required for the abstraction. Concurrent with our work, Clarke et al. [2] proposed heuristic algorithms to perform counterexample-guided automatic abstraction refinement. An initial abstraction is obtained by examining the transition blocks corresponding to the variables in the program. This is followed by iteratively model checking the abstracted structure and heuristically refining the abstraction in case a spurious counterexample is generated. They also show that the problem of finding the *coarsest* refinement *i.e.*, the abstraction with the least number of equivalence classes is *NP-hard*.

In general, verification of real systems by completely automated abstraction is difficult and calls for sophisticated heuristic algorithms. Manual abstraction based on human insight may still offer significant benefits in several practical verification problems.

3 Preliminaries

The directory-based cache protocol consists of a central server H, and a set of client caches $C_1, \ldots, C_n$. Each client C_i communicates with the server through a set of three message channels namely $ch1_i, ch2_i, ch3_i$. Thus, the network

topology is a star graph as shown in Fig 1.

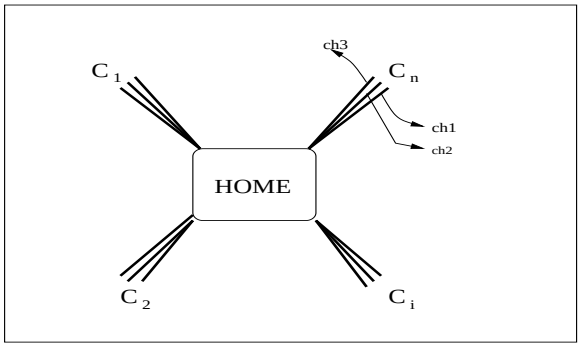

Figure 1: The network topology

The set of messages M is $\{null, req\_sh, req\_ex, inv, inv\_ack, gr\_sh, gr\_ex\}$. Each client could be in one of the three states - $\{$*shared, invalid , exclusive*$\}$, denoted by S, I, E respectively. The *home* H maintains the following variables:

bool hsl[$1 \ldots n$**]** : a boolean array where the i'th bit is set to 1 if C_i is in state S, else 0.
bool hil[$1 \ldots n$**]** : a boolean array where the i'th bit is set to 1 if C_i is in state I, else 0.
bool heg: a bit which is set to 1 whenever H grants exclusive access to some client.
msg hcm: a message ($\in M$) that stores the last command the *home* read from any of the channels.
int hcc: an integer that stores the id of the last client that the *home* serviced.

Each of the n clients maintains a state variable:
c_state c[$1 \ldots n$**]**: a state $\in \{S, I, E\}$

The state of each channel is represented by an array of message variables:
msg ch1[$1 \ldots n$**], ch2[**$1 \ldots n$**], ch3[**$1 \ldots n$**]** : the current message $\in M$ present in the channel.

The actual protocol specification in Murϕ is given in the appendix. It consists of 10 rules of the form:

rule i:
$\wp_i$
$\Longrightarrow$
begin
a_1;
a_2;
.
.
end

where $\wp_i$ is a predicate over the program variables described above and the a_i's enclosed between the $begin$ and end constructs denote an atomic set of actions that modify the values of one or more program variables.

As the rule descriptions suggest, each rule can be thought of as an action either of a client or of the *home*. The Murϕ rules determine atomic transitions between states according to non-deterministic interleaving. More precisely, if a rule is enabled in global state s, then there is a transition from s to t, where t results from applying all the assignments within the *begin* or *end* block of the rule.
Let

$$coherent \equiv (\forall i \neq j)(c[i] = E \Rightarrow c[j] = I)$$

The specification of correctness is that $coherent$ be an invariant of the protocol. The correctness has to be proved for the complete family of program instances generated by the parameter n, the number of clients in the system.

A global state of the program is a valuation of the variables introduced above, including the program parameter n. It follows that the state space of this system is infinite since n could take any positive integral value. We denote by M_k the finite state system obtained by restricting the state space to a particular value of the parameter n, namely k.

4 Verification Methodology

We verify the coherence property by applying a series of transformations to the original program, which we refer to as P henceforth and reducing it to a considerably simpler protocol $P^{'}$. We then verify the correctness of $P^{'}$ by model checking. Thereafter we show that the correctness of $P^{'}$ implies the correctness of P.

In order to understand the intractability of the problem to naive approaches, we compute the size of the state space graph for a fixed instance of the program, *i.e.* for a given n. We introduce the following lemma that describes the range of values of some program variables. The lemma follows straightforwardly from an inspection of the input program.

Lemma 1: The following are invariants of the protocol:
(i) $(\forall i)(ch1[i] \in \{null, req\_sh, req\_ex\})$
(ii) $hcm \in \{null, req\_sh, req\_ex\}$
(iii) $(\forall i)(ch2[i] \in \{null, inv, gr\_sh, gr\_ex\})$
(iv) $(\forall i)(ch3[i] \in \{null, inv\_ack\})$

From the above lemma, it follows that the size of the state space graph for a given instance of the program with n clients is given by $(2 \cdot 3 \cdot n \cdot 2^n \cdot 2^n) \cdot (3^n) \cdot (3 \cdot 4 \cdot 2)^n = 6 \cdot n \cdot 288^n$. The first parenthesized term represents the number of valuations of the *home* variables - $heg, hcm, hcc, hsl[1 \ldots n], hil[1 \ldots n]$. The second term represents the number of valuations of *client* states since each client could be in one of the three states - *invalid, shared, exclusive*. The last term represents the size of the state space generated by the values of the contents of channels *ch1, ch2, ch3*.

It is thus clear that a naive model checker can handle instances of the program with n at most 4 or 5. Hence, we need more sophisticated methods to verify the coherence property over the infinite state space represented by the complete parameterized family of programs. In the next few subsections we explain some standard procedures to reduce the size of the state space graph and transform it to a more manageable form.

Variable Redistribution

We observe that the *home* stores an unbounded set of information in the form of the bit arrays $hsl[1 \cdots n]$ and $hil[1 \cdots n]$. Thus, the number of local states for *home* is infinite. In order to make the verification problem tractable we therefore apply the following transformations to the structuring of the program variables:
(i) We distribute the arrays $hil[]$ and $hsl[]$, one bit each to among each of the n clients and consider them as local client variables.
(ii) We associate the channel content information with the client instead of considering them as separate entities.
Thus, the state of a client C_i is now represented by the following tuple:
$\{hsl[i], hil[i], ch1[i], ch2[i], ch3[i], c\_state[i]\}$. It is easy to convince oneself that there exists a one-to-one mapping from every state in the global state space graph of the original program P to a corresponding state in the program obtained by applying transformations (i) and (ii). It follows that the number of local client states in the transformed program is now 288 (instead of 3 earlier). The size of the global state space graph for each instance of the program still remains unchanged. The reason for this redistribution of variables is that the symmetry of clients in program P can be exploited by applying a bunch of techniques like symmetry reduction, abstraction etc. and converted to a program $P^{'}$ which has a finite sized global state space graph.

Symmetry Reduction

Consider a program parameterized by n, the number of symmetric components in the program. Let each of the n components be in at most one of k different states. Then the size of the state space graph is $O(k^n)$, exponential in n. However, if we exploit the symmetry of the components and instead maintain only a count of the number of clients in each of the k states rather than explicitly remembering what state each client is in, then the size of the state space graph is only n^k, which is polynomial in n. In other words, we maintain an array $s[1 \cdots k]$, where $0 \leq s[i] \leq n, 1 \leq i \leq k$, where the value $s[i]$ represents the number of clients in state i.

Such a symmetry reduction transformation has been used before and is incorporated in Murϕ. It is also a straight forward procedure to automatically rewrite the input program to correspond to the symmetry reduced version. Murϕ imposes a set of restrictions a program with symmetric components must exhibit to be reducible thus. Intuitively these

restrictions make sure that the symmetric components can be reordered arbitrarily without changing the behavior of the system.

Repetitive Constructors

The above abstraction reduces the state space for each instance of the program generated by a value of the parameter n from a term exponential in n to one polynomial in n. However, the global state space is still infinite. In the abstraction using repetitive constructors we abstract away the exact number of components in each component state s and replace it with instead with either 0, if there are no clients in s, or $+$ if there are one or more clients in that state.

Thus, an abstract state constructed using repetition constructors is a group of concrete states in which the array of abstractable components has been replaced with a mapping of each possible component state to either 0 or $+$. Note that with this abstraction the size of the global state space graph becomes independent of n. In order to calculate the total number of global states, we observe that each local client state could be in at most one of 288 states. Each of these states could be associated with one of two repetitive constructor symbols. The total number of global states is therefore $6 \cdot 288 \cdot 2^{288}$ (since there are $6 \cdot 288$ local states for *home* generated by valuations of the variables *heg, hcm, hcc*). This is still an unmanageably large number. However the important point is that we have now reduced the parameterized family of programs to a finite state space.

State Clustering

In order to make the state space further tractable, we first proceed to determine how many of the 288 local client states are actually reachable from the start state. We denote a local client state by the tuple

$$\{hsl[i], hil[i], ch1[i], ch2[i], ch3[i], c\_state[i]\}.$$

The initial state for each client is : $\{0, 0, NULL, NULL, NULL, I\}$. In each of the rules in the protocol specification, we simply drop the conditions involving any of the *home* variables hcm, hcc or heg. Since each of the conditions involving hcm, hcc or heg appears only in conjunction with other conditions in each of the 10 rules, it is clear that the set of all reachable states thus generated will be a superset of the set of states that are actually reachable in the original program $P^{'}$. The number of reachable local client states with such a relaxed set of rules is found to be 177.

Finally, we groups all of these 177 *potentially* reachable states into 13 *meta-states* containing a total of 42 states and 1 error state comprising the remaining 135 states. The actually grouping of these states is given below in the appendix. Below we list the constraints that describe each clustered metastate. This grouping was done manually. The intuition for coming up with these constraints was essentially gathered by a counterexample guided approach where we kept refining the clustering till we were able to derive one that satisfied the coherence property. Of greatest help was the insight into the 42 states that are actually reachable from the start state and clustering the remaining 135 state into an *error* state. This insight was gathered from the hand proof of the protocol wherein we established some invariants of the protocol, which we list below:

(i) $(\forall i)(c[i] = E \Rightarrow heg)$
(ii) $(\forall i)(c[i] = E \Rightarrow hsl[i])$
(iii) $(\forall i)(c[i] = S \Rightarrow hsl[i])$
(iv) $hcm = null \Rightarrow (\forall i)(ch3[i] = null \wedge ch2[i] \neq inv)$

Any local state that did not satisfy any of these properties was filtered out and included in the *error* metastate. Observe that the correctness of any of the above claimed invariants is immaterial to the correctness of our verification methodology itself. This is because any abstraction only computes a conservative approximation of the original program *w.r.t* to the coherence property.

Next we describe the set of constraints that characterize the meta-states that represent the local client states in an abstracted form. There are 13 reachable meta-states and one *error* state. A constraint describing an abstracted local client state is written in terms of the local client variables $\{hsl, hil, ch1, ch2, ch3, c_{state}\}$. The following lists the constraint corresponding to each abstracted state.

0.	$!hsl \ \wedge \ !hil \ \wedge \ (ch1 = null) \ \wedge \ (ch2 = null) \ \wedge \ (ch3 = null) \ \wedge \ (c_{state} = I)$
1.	$!hsl \ \wedge \ !hil \ \wedge \ (ch1! = null) \ \wedge \ (ch2 = null) \ \wedge \ (ch3 = null) \ \wedge \ (c_{state} = I)$
2.	$hsl \ \wedge \ (ch2 = gr\_sh) \ \wedge \ (ch3 = null) \ \wedge \ (c_{state} = I)$
3.	$hsl \ \wedge \ !hil \ \wedge \ (ch2 = gr\_ex) \ \wedge \ (ch3 = null) \ \wedge \ (c_{state} = I)$
4.	$hsl \ \wedge \ hil \ \wedge \ (ch2 = gr\_ex) \ \wedge \ (ch3 = null) \ \wedge \ (c_{state} = I)$
5.	$hsl \ \wedge \ !hil \ \wedge \ (ch2 = null) \ \wedge \ (inv_ack) \ \wedge \ (c_{state} = I)$
6.	$hsl \ \wedge \ !hil \ \wedge \ (ch2 = null) \ \wedge \ (ch3 = null) \ \wedge \ (c_{state} = S)$
7.	$hsl \ \wedge \ hil \ \wedge \ (ch2 = null) \ \wedge \ (ch3 = null) \ \wedge \ (c_{state} = S)$
8.	$hsl \ \wedge \ (ch2 = gr_sh) \ \wedge \ (ch3 = null) \ \wedge \ (c_{state} = S)$
9.	$hsl \ \wedge \ !hil \ \wedge \ (ch2 = inv) \ \wedge \ (ch3 = null) \ \wedge \ (c_{state} = S)$
10.	$hsl \ \wedge \ !hil \ \wedge \ (ch2 = null) \ \wedge \ (ch3 = null) \ \wedge \ (c_{state} = E)$
11.	$hsl \ \wedge \ hil \ \wedge \ (ch2 = null) \ \wedge \ (ch3 = null) \ \wedge \ (c_{state} = E)$
12.	$hsl \ \wedge \ !hil \ \wedge \ (ch2 = inv) \ \wedge \ (ch3 = null) \ \wedge \ (c_{state} = E)$

The *error* metastate consists of all states that do not satisfy *any* of the above constraints.

Protocol Rewriting

In this section we show hot to rewrite the program itself to incorporate the abstraction techniques explained above. There are 14 local states in which each of the clients could be in. A global state is represented by a tuple $\{S[], hcm, hcc, heg\}$, where $S[]$ is a bit array of size 14 such that $S[i] = 0$, if no client is in state i, else $S[i] = +$. The variables hcm, hcc and heg are maintained by *home*. However, note that now hcc is not a client-id, but a state-id corresponding to the state to which the client belongs, *i.e.* $0 \leq hcc \leq 13$ now represents all clients that are in local state hcc. We denote the rewritten program as $P^{'}$ henceforth.

The rewriting procedure generates transitions in the abstracted program $P^{'}$ from transitions in the original program P by simulating transitions from the concrete states comprising each abstract state. A transition in the abstract state graph (s, t) is introduced iff there are concrete states u, v such that $u \equiv s$, $v \equiv t$, and there exists a transition (u, v) in the corresponding concrete state graph. For truncated counters this implies that if a client makes a transition (l_i, l_j), where L_i and L_j are local component states in the abstracted program, then the counter $s[j]$ is set to $+$, while $s[i]$ is non-deterministically set to both 0 and $+$.

As an example we show how rule 3 is rewritten after incorporating all of the 3 abstractions in the original program P. Rule 3 in P looks as follows:

```
rule 3  /* home picks new request */
  hcm = null & ch1[i] != null
==>
  begin
    hcm := ch1[i];
        ch1[i] := null;
        hcc := i;
        hil := hsl;
  end;
```

In $P^{'}$, rule 3 is rewritten as several rules, one for each of the 14 states for which it is applicable. We show below examples of rule 3 rewritten for state 1:

```
rule 3
  hcm = null & state[0] = 1
==>
  begin
    hcm := req_sh;          /* since ch1[0] = req_sh */
    hcc := 0;
    state[1] := 0;
  end;
||
```

```
==>
  begin
    hcm := req_ex;
    hcc := 0;
    state[1] := 0;
  end;
||
==>
  begin
    hcm := req_sh;
    hcc := 0;
    state[1] := 1;
  end;
||
==>
  begin
    hcm := req_ex;
    hcc := 0;
    state[1] := 1;
  end
```

It is clear that the size of the program increases because of such a transformation. However, in general and for the protocol under consideration, this is not a concern. The size blow up in the length of the program is shown in [10] to be bounded by a term polynomial in the size of the original program . We observe that such rewriting may be automated easily. In fact the work in [2] on counterexample guided abstraction refinement attempts to automate not only the rewriting but also the state clustering itself through heuristics based on *abstraction functions*.

5 Model Checking $P^{'}$

We initialize the global state to $(s[i] = 0, 0 \leq i \leq 13, 0, 0, 0)$ and computed the number of distinct global states generated. Note that the number of reachable states after incorporating all of the abstraction techniques described in the previous section is $2^{14} \cdot 2 \cdot 2 \cdot 14 = 57344$. However, it turns out that there are only 1277 reachable states. In each of these states, it is easily and explicitly checked that if there is any client in an *exclusive* state ($s[10] := +$ or $s[11] := +$ or $s[12] := +$), then all other clients are in $invalid$ state.

However, $s[i] = +, 10 \leq i \leq 12$ could still mean that multiple clients co-exist in an *exclusive* state. To handle this problem, during the course of the state space generation, every time $s[10]$ or $s[11]$ or $s[12]$ is set to $+$, we check in the program whether it is already $+$, in which case we abort the program and announce a counterexample. With the abstraction containing 14 states described above, this case never occured in the program! Hence, it follows that the coherence property holds in the abstracted program $P^{'}$.

Number of local client states initially	288
Number of abstracted meta-states	14
Total number of global states	1277

The original specification of the protocol was in Murϕ. However, the rewritten program $P^{'}$ was coded in C and used to enumerate all possible reachable global states. The program to model check $P^{'}$ takes about 1.1 sec on a Pentium II 450 MHz processor with 128 MB RAM.

Property 1: Given two abstract states A and B, if there exists concrete state $a \in A$ and $b \in B$ such that (a, b) is a transition in the original state graph (A, B) is a transition in the abstract state graph.

Theorem 1: If an invariant holds true in the abstracted program $P^{'}$, then it also hold true in the original program $P^{'}$.

Proof: The proof of this theorem follows straightforwardly from property mentioned above. Using inductive reasoning one can show that if the invariant is ever violated in the abstract program $P^{'}$, then one can construct a path in the original program P as well leading to state where the invariant is violated. Detailed formalisms and proofs may be obtained in [8].

This leaves only a small technicality to be handled, that of redistribution of the $home$ variabled to the client. However, such a redistribution preserves the state space of the original program P up to isomorphism and doesn't modify any of the transitions. Thus, any invariant that holds in the program with redistributed variables extends to the original program as well.

6 Conclusion

In this paper, we presented the mechanical verification of the correctness of a parametrized distributed directory-based cache protocol. The state explosion problem compelled us to use several known abstraction techniques to reduce the size of the global state space graph. We used symmetry reduction, repetetive constructors and ultimately counterexample guided manual state clustering and managed to successfully verify the protocol. The abstracted version of the protocol had only 1277 global states! We believe that the techniques used in this paper can be generalized and automated. For future work we plan to focus on sophisticted heuristics for automated counterexample guided abstraction refinement.

References

[1] F. Balarin and A.L. Sangiovanni-Vinventelli. On the automatic computation of network invariants. In *6'th Int'l Conf. on Computer-Aided Verification*, June 1994.

[2] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV'2000*, 2000.

[3] E.M. Clarke and E.A. Emerson. Design and verification of synchronization skeletons using branshing time temporal logic. In *In Logics of Programs Workshop*, volume LNCS no 131, pages 52–71. Springer, May 1981.

[4] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent system using temporal logic. In *In ACM Transactions on Prog. Lang. and Sys.*, volume vol 8, no 2, pages 244–263, April 1986.

[5] E.M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking algorithms. In *Proceedings of the 6'th Annual ACM Symposium on Principles of Distributed Computing*, 1987.

[6] E.M. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks using abstraction and regular languages. In *CONCUR'95*, 1995.

[7] E.J. Dijkstra. Invariance and nondeterminacy. In *In mathematical Logic and Programming Languages. Prentice-Hall*, 1985.

[8] David Dill and Ip Norris. Better verification through symmetry. In *Formal Methods in System Design*, volume 9, Numbers 1/2, 1996.

[9] David Dill and Ip Norris. Verifying systems with replicated components in murphi. In *Int'l Conf on Computer-Aided Verification*, 1996.

[10] Allen E. Emerson and Richard Trefler. From asymmetry to full symmetry. In *CHARME'99*, 1999.

[11] Steve German and A. P. Sistla. Reasoning about systems with many processes. In *Journal of Association for Computing Machinery*, volume 39(3), pages 675–735, 1992.

[12] Susanne Graf. Verification of a distributed cache memory by using abstractions. In *6'th Int'l Conf. on Computer-Aided Verification*, 1994.

[13] R. Apt Krzysztof and C. Dexter Kozen. Limits for automatic verification of finite state concurrent programs. In *Information Processing Letters*, volume 22, pages 307–309, 1986.

[14] Robert Kurshan, Michael Merritt, Ariel Orda, and Sonia R. Sachs. A structural linearization principle for processors. In *Formal Methods in System Design*, volume 5, 1994.

[15] Boris D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs. In *Acta Informatica*, volume 21:125-169, 1984.

[16] F. Pong and M. Dubois. A new approach for the verification of cache coherence protocols. In *I. Acta Informatica*, volume 21, pages 125–169, 1984.

[17] June-Kyung Rho and Fabio Somenzi. Automatic generation of network invariants for the verification of iterative sequential systems. In *5'th Int'l Conf. on Computer-Aided Verification*, June 1987.

[18] A.P. Sistla and S.M. German. Reasoning with many processes. In *Symp. on Logic in Computer Science*, 1987.

[19] Pierre Wolper and Vinciane Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems*, volume 407 of LNCS, Springer-Verlag, 1989.

Appendix

In this section we give the specification of the protocol in Murϕ-like pseudo-code, the format in which the protocol was originally communicated to us.

```
type
  msg:            enum{ null,         /* empty */
                        req_sh,       /* request_shared */
                        req_ex,       /* request_exclusive */
                        inv,          /* invalidate */
                        inv_ack       /* invalidate_ack */
                        gr_sh         /* grant_shared */
                        gr_ex         /* grant_exclusive */
                      };
  c_state:        enum{ I,     /* invalid   */
                        S,     /* shared    */
                        E,     /* exclusive */
                      };
  client:         1...n;

var
  ch1:      array[client] of msg;        /* channel 1 */
  ch2:      array[client] of msg;        /* channel 2 */
  ch3:      array[client] of msg;        /* channel 3 */
  hsl:      array[client] of boolean     /* home_shared_list */
  hil:      array[client] of boolean     /* home_invalidate_list */
  heg:      boolean;                     /* home_exclusive_granted */
  hcm:      msg;                         /* home_current_command */
  hcc:      client;                      /* home_current_client */
  c:        array[client] of c_state     /* distributed cache state */


ruleset i: client do

  rule 1  /* client requests shared access */
    c[i] = I & ch1[i] = null
  ==>
  begin
```

```
    ch1[i] := req_sh;
  end;

rule 2  /* client requests exclusive access */
  c[i] = I & ch1[i] = null;
==>
  begin
    ch1[i] := req_ex;
  end;

rule 3  /* home picks new request */
  hcm = null & ch1[i] != null
==>
  begin
    hcm := ch1[i];
    ch1[i] := null;
    hcc := i;
    hil := hsl;
  end;

rule 4  /* home sends invalidate message */
  hil[i] & ch2[i] = null & ((hcm = req_sh & heg) | hcm = req-ex)
==>
  begin
    ch2[i] = inv;
    hil[i] := 0;
  end;

rule 5  /* home receives invalidate acknowledgement */
  hcm != null & ch3[i] = inv_ack
==>
  begin
    hsl[i] := 0;
    heg := 0;
    ch3[i] := null;
  end;

rule 6  /* sharer invalidates cache */
  ch2[i] = inv & ch3[i] = null
==>
  begin
    ch2[i] := null;
    ch3[i] := inv_ack;
    c[i] := I;
  end;

rule 7  /* client receives shared grant */
  ch2[i] = gr_sh
==>
  begin
    c[i] := S;
    ch2[i] := null;
  end;

rule 8 /* client receives exclusive grant */
  ch2[i] = gr_ex
==>
  begin
    c[i] := E;
```

```
    ch2[i] := null;
  end;

end;  /* ruleset */

rule 9  /* home grants share */
  hcm = req-sh & !heg & ch2[hcc] = null
==>
  begin
    hsl[hcc] := 1;
    hcm := null;
    ch2[hcc] := gr_sh;
  end;

rule 10  /* home grants exclusive */
  hcm = req_ex & (forall j: client do hsl[j] = 0) & ch2[hcc] = null
==>
  begin
    hsl[hcc] := 1;
    hcm := null;
    heg := 1;
    ch2[hcc] := gr_ex;
  end;


startstate
  begin
    forall i: client do
    begin
      ch1[i] := null;
      ch2[i] := null;
      ch3[i] := null;
      c[i] := I;
      hsl[i] := 0;
      hil[i] := 0;
    end;
    hcm := null;
    hcc := 1;
    heg := 0;
  end;
```

The State Clustering

Given below is the clustering of the local client states into a total of 13 reachable metastates. The *error* state consists of all the states not satisfying any of these constraints.

```
------------------------------------------------------------------------
0. { 0, 0, NUL, NUL, NUL, I }           1. { 0, 0, REQ_SH, NUL, NUL, I }
                                           { 0, 0, REQ_EX, NUL, NUL, I }

2. { 1, 0, NUL, GR_SH, NUL, I }         3. { 1, 0, NUL, GR_EX, NUL, I }
   { 1, 0, REQ_SH, GR_SH, NUL, I }         { 1, 0, REQ_SH, GR_EX, NUL, I }
   { 1, 0, REQ_EX, GR_SH, NUL, I }         { 1, 0, REQ_EX, GR_EX, NUL, I }
   { 1, 1, NUL, GR_SH, NUL, I }
   { 1, 1, REQ_SH, GR_SH, NUL, I }
   { 1, 1, REQ_EX, GR_SH, NUL, I }
```

```
4. { 1, 1, NUL, GR_EX, NUL, I }        5. { 1, 0, NUL, NUL, INV_ACK, I }
   { 1, 1, REQ_SH, GR_EX, NUL, I }        { 1, 0, REQ_SH, NUL, INV_ACK, I }
   { 1, 1, REQ_EX, GR_EX, NUL, I }        { 1, 0, REQ_EX, NUL, INV_ACK, I }

-------------------------------------------------------------------------

6. { 1, 0, NUL, NUL, NUL, S }          7. { 1, 1, NUL, NUL, NUL, S }
   { 1, 0, REQ_SH, NUL, NUL, S }          { 1, 1, REQ_SH, NUL, NUL, S }
   { 1, 0, REQ_EX, NUL, NUL, S }          { 1, 1, REQ_EX, NUL, NUL, S }

8. { 1, 0, NUL, GR_SH, NUL, S }        9. { 1, 0, REQ_SH, INV, NUL, S }
   { 1, 0, REQ_SH, GR_SH, NUL, S }        { 1, 0, REQ_EX, INV, NUL, S }
   { 1, 0, REQ_EX, GR_SH, NUL, S }        { 1, 0, NUL, INV, NUL, S }
   { 1, 1, NUL, GR_SH, NUL, S }
   { 1, 1, REQ_SH, GR_SH, NUL, S }
   { 1, 1, REQ_EX, GR_SH, NUL, S }

-------------------------------------------------------------------------

10.{ 1, 0, NUL, NUL, NUL, E }          11. { 1, 1, NUL, NUL, NUL, E }
   { 1, 0, REQ_SH, NUL, NUL, E }           { 1, 1, REQ_SH, NUL, NUL, E }
   { 1, 0, REQ_EX, NUL, NUL, E }           { 1, 1, REQ_EX, NUL, NUL, E }

12.{ 1, 0, NUL, INV, NUL, E }
   { 1, 0, REQ_SH, INV, NUL, E }
   { 1, 0, REQ_EX, INV, NUL, E }
-------------------------------------------------------------------------
```