Using Flow Specifications of Parameterized Cache Coherence Protocols for Verifying Deadlock Freedom

Divjyot Sethi[1], Muralidhar Talupur[2], and Sharad Malik[1]

[1] Princeton University, Princeton, NJ, USA
[2] Strategic CAD Labs, Intel Corporation, Hillsboro, OR, USA

Abstract. We consider the problem of verifying deadlock freedom for symmetric cache coherence protocols. While there are multiple definitions of deadlock in the literature, we focus on a specific form of deadlock which is useful for the cache coherence protocol domain and is consistent with the internal definition of deadlock in the Murphi model checker: we refer to this deadlock as a *system-wide deadlock (s-deadlock)*. In s-deadlock, the entire system gets blocked and is unable to make any transition. Cache coherence protocols consist of N symmetric cache agents, where N is an unbounded parameter; thus the verification of s-deadlock freedom is naturally a parameterized verification problem.

Parametrized verification techniques work by using sound abstractions to reduce the unbounded model to a bounded model. Efficient abstractions which work well for industrial scale protocols typically bound the model by replacing the state of most of the agents by an abstract environment, while keeping just one or two agents as is. However, leveraging such efficient abstractions becomes a challenge for s-deadlock: a violation of s-deadlock is a state in which the transitions of all of the unbounded number of agents cannot occur and so a simple abstraction like the one above will not preserve this violation. Authors of a prior paper, in fact, proposed using a combination of over and under abstractions for verifying such properties. While quite promising for a large class of deadlock errors, simultaneously tuning over and under abstractions can become complex.

In this work we address this challenge by presenting a technique which leverages high-level information about the protocols, in the form of message sequence diagrams referred to as *flows*, for constructing invariants that are collectively stronger than s-deadlock. Further, violations of these invariants can involve only one or two interacting agents: thus they can be verified using efficient abstractions like the ones described above. We show how such invariants for the German and Flash protocols can be successfully derived using our technique and then be verified.

1 Introduction

We consider the problem of verifying deadlock freedom for symmetric cache coherence protocols. Consider a cache coherence protocol $\mathcal{P}$ (N) where the parameter N represents an unbounded number of cache agents. The protocol implements requests sent by the agents using messages exchanged in the protocol. For a protocol designer, the main property of interest is the request-response property, i.e., every request from an agent eventually gets a response. Since this property is a liveness property which is hard for

F. Cassez and J.-F. Raskin (Eds.): ATVA 2014, LNCS 8837, pp. 330–347, 2014.

existing model checking tools, designers resort to identifying causes for response property failure, such as deadlock-style failures, and verify against them.

The literature is abundant with various definitions of deadlock [8, 22]. We focus on deadlock errors in which the entire protocol gets blocked, i.e., no agent of the protocol can make any transition. We refer to such an error as a *system-wide deadlock (s-deadlock)*. If we model each transition τ of the protocol to have a guard $\tau.g$, which is $false$ if the transition is not enabled, the s-deadlock error occurs if the guards of all the transitions are *false*, i.e., $\bigwedge_{\tau} \neg(\tau.g)$ is *true*. This kind of failure, while weaker than other broader classes of deadlock failures, is commonly observed in industrial computer system designs and is consistent with the internal definition for deadlock used by the Murphi model checker as well [23]. This class of deadlocks is well motivated for parameterized cache coherence protocols as these use a centralized synchronization mechanism (e.g. a directory) and thus any deadlock results in the directory getting blocked. It is highly likely that such a deadlock in the shared directory will end up involving all of the agents of the protocol getting blocked, i.e., unable to make any transition.

Since an s-deadlock error involves all of the unbounded number of agents getting blocked and unable to make any transition, verification of s-deadlock freedom naturally is a parameterized verification problem. Parameterized verification techniques work by using sound abstractions to reduce the unbounded model to a finite bounded model that preserves the property of interest. These abstractions typically tend to be simple over-abstractions such as *data-type reduction* [30]. This abstraction keeps a small number of agents (1 or 2) as is and replaces all the other agents with an abstract environment. Such abstractions along with parameterized techniques like the CMP (CoMPositional) method [11] have had considerable success in verifying key safety properties like mutual exclusion and data integrity even for industrial scale protocols [11, 32, 38].

1.1 Challenge in Verifying S-deadlock

While parameterized techniques are successful for safety properties such as mutual exclusion and data integrity, the application of such abstractions for parameterized verification of properties such as s-deadlock is hard. The key challenge arises from the fact that an s-deadlock violation is a state in which all the guards are *false*, i.e., when $\bigwedge_{\tau} \neg(\tau.g)$ holds; simple over-abstractions such as data-type reduction will easily mask this violation due to the discarded state of agents other than 1 and 2 and the extra transitions of the environment.

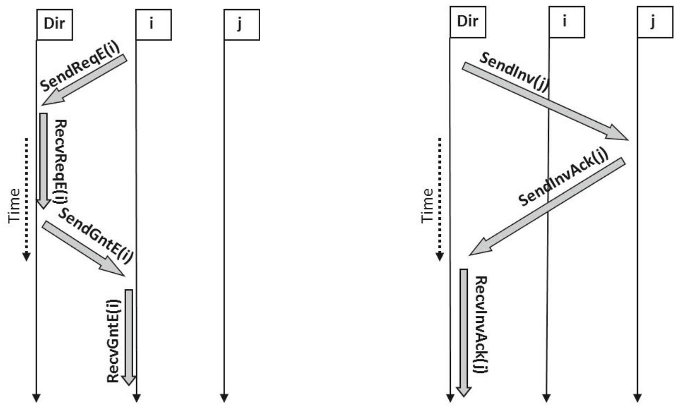
One approach to address the above issue is to use a combination of over and under abstractions (i.e., a *mixed* abstraction) instead of data-type reduction, as described in a prior deadlock verification work [8]. While promising for verifying a large class of deadlock errors, the use of mixed abstraction requires reasoning about over and under abstraction simultaneously and easily becomes fairly complex.

In this paper we take a different approach. We show how high-level information about the protocols, in the form of message sequence diagrams referred to as *flows*, can be leveraged to construct invariants which are collectively stronger than the s-deadlock freedom property. These invariants are amenable to efficient abstractions like data-type reduction which have been used in the past for verifying industrial scale protocols.

1.2 Leveraging Flows for Deadlock Freedom

Cache coherence protocols implement high-level requests for read (termed $Shared$) or write (termed $Exclusive$) access from cache agents, or for invalidating access rights (termed $Invalidate$) of some agent from the central directory. The implementation of these requests is done by using a set of transitions which should occur in a specific protocol order. This ordering information is present in diagrams referred to as *message flows* (or *flows* for brevity). These flows are readily available in industrial documents in the form of message sequence charts and tables [38].

Fig. 1 shows two of the flows for the German cache coherence protocol describing the processing of the $Exclusive$ and $Invalidate$ requests. Each figure has a directory Dir, and two agents i and j. The downward vertical direction indicates the passage of time. The $Exclusive$ request is sent by the cache agent i to the directory Dir to request a write access. The $Exclusive$ flow in Fig. 1(a) describes the temporal sequence of transitions which occur in the implementation in order to process this request: each message is a transition of the protocol. The message $SendReqE(i)$ is sent by the agent i to Dir which receives this message by executing the transition $RecvReqE(i)$. Next, if the directory is able to grant $Exclusive$ access, it sends the message $SendGntE(i)$ to agent i which receives this grant by executing $RecvGntE(i)$. However, in case the directory is unable to send the grant since another agent j has access to the cache line, the directory sends a request to invalidate the access rights of j. The temporal sequence of transitions which occurs in the implementation in this case is shown in the $Invalidate$ flow in Fig. 1(b). This flow proceeds by the directory sending the $SendInv(j)$ message, the agent j sending the acknowledgment message $SendInvAck(j)$, and the directory receiving it by executing $RecvInvAck(j)$ transition.



(a) $Exclusive$ flow (b) $Invalidate$ flow

Fig. 1. Flows for the German protocol

Freedom from S-deadlock. At a high-level, our method tries to exploit the fact that if the protocol is s-deadlock free, when none of the transitions of an agent are enabled, another agent can be identified which must have a transition enabled. This identification leverages the key insight that in any state of the protocol, if all the transitions of some agent, say a_1, cannot occur, then, some flow of that agent must be blocked since it depends on another flow of another agent, say a_2, to finish. Then, there are two possibilities: (1) the agent a_2 is enabled, in which case the state is not an s-deadlock state, or (2) the agent a_2 is blocked as well, in which case it depends on another agent a_3. If this dependence chain is acyclic, with the final agent in the chain enabled, the protocol is s-deadlock free. However, if the final agent is not enabled, or if the dependence chain has a cycle, the protocol may either have an s-deadlock error or there may be an error in the flow diagrams used.

As an example, for the German protocol, if the $Exclusive$ flow of agent i is blocked since the transition $SendGntE(i)$ cannot occur, it is waiting for j to get invalidated. In the protocol, at least some transition of the $Invalidate$ flow on agent j can occur. This enables proving freedom from s-deadlock for the protocol.

Using the above insight, by analyzing the dependence between blocked agents, our method is able to point to an agent which must have at least one transition enabled in every reachable state of the protocol. Specifically, our method enables the derivation of a set of invariants $\mathcal{I}$ which collectively partition the reachable state of the protocol. Each invariant then points to the agents which must have at least one transition enabled when the protocol is in a state belonging to its partition. These invariants are derived in a loop by iteratively model checking them on a protocol model with c agents, where c is heuristically chosen as discussed in Section 3.

Verifying for an Unbounded Number of Agents. Once the invariants in $\mathcal{I}$ are derived, they hold for a model with c agents. These invariants use just one index (i.e., they are of the form $\forall i : \phi(i)$) and thus, they can be verified for an unbounded number of agents by using efficient parameterized verification techniques such as data-type reduction along with the CMP (CoMPositional) method [11]. This technique has previously been successful for verifying mutual exclusion for industrial protocols [32]. We note that our approach is not limited to the CMP method: the invariants derived may be verified by using any parameterized safety verification technique [15, 27, 34, 35].

1.3 Key Contributions

Our method proves s-deadlock freedom for parameterized protocols (formalized in Section 2). It takes a Murphi model of the protocol as input. As shown in Fig. 2, first, a set of invariants $\mathcal{I}$ which collectively imply s-deadlock freedom are derived on a model with c agents (Section 3). These invariants are verified for an unbounded number of agents by using state-of-the-art parameterized verification techniques (Section 4). We verified Murphi implementations of two challenging protocols, the German and Flash protocols using our method (Section 5).

Limitation: The key limitation of our approach is that the invariants have to be derived manually by inspecting counterexamples. This can be automated if additional information about conflicting flows is available in the flow diagram itself.

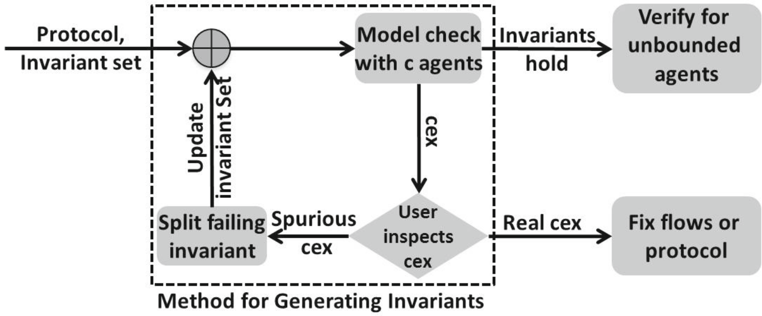


Fig. 2. Experimental Flow

1.4 Relevant Related Work

Deadlock Verification: The work closest to ours is by Bingham *et al.* [7, 8]. They formally verify deadlock as a safety property for protocols by specifying it using user-identified Quiescent states (*i.e.*, a state in which no resources are held): they specify a protocol state to be a deadlock state if no Quiescent state is reachable from it. They prove freedom from such a deadlock by using a combination of over and under abstractions (*i.e.*, a *mixed* abstraction [16]). Their approach is promising for verifying deadlock freedom and scales to the Flash protocol. However, the required tuning of both under and over abstractions simultaneously can be complex. In contrast, we take the flow-based alternative to enable simpler abstractions like data-type reduction.

Since the ultimate goal of any deadlock verification effort is to verify the response property (i.e. every high-level request eventually gets a response), we contrast our work with liveness verification efforts as well. Among techniques for parameterized verification of liveness, McMillan has verified liveness properties of the Flash protocol [28,29]. The proof is manual and works on the basis of user supplied lemmas and fairness assumptions. In contrast, our method reduces manual effort by leveraging information from flows along with the CMP method. Among automatic approaches for verifying liveness properties, Baukus *et al.* verified liveness properties of the German protocol [6] using a specialized logic called WSIS. Fang *et al.* used automatically deduced ranking functions [21] and, in a prior work, counter abstraction [35] to verify liveness properties. While fully automatic, these approaches tend to exhibit limited scalability for larger protocols such as Flash, due to the inherent complexity of the liveness verification problem. In contrast to these, our approach, while requiring some user guidance, achieves much greater scalability and enables us to verify the Flash protocol.

Parameterized Verification Techniques: We note that the invariants derived using our method can be verified for an unbounded number of caches by any parameterized safety verification technique, it is not dependent on the CMP method which we used. Our choice of using the CMP method was motivated by the fact that it is the only state-of-the-art method we are aware of which has been used successfully for verifying protocols like Flash and other industrial scale protocols. Among other techniques, an important technique by Conchon *et al.* [15] uses a backward reachability algorithm to automatically prove a simplified version of the Flash protocol. Next, there are numerous other prior approaches in literature for parameterized verification of safety properties. The CMP method falls in the broad category of approaches which use *compositional reasoning* [1,2] and *abstraction* based techniques to verify parameterized systems; the literature is abundant with examples of these [13,14,17,20,27,28,34,35]. Next, another category of approaches work by computing a *cutoff* bound k and showing that if the verification succeeds for k agents, then the protocol is correct for an arbitrary number of agents [3,5,12,18,19,24]. Finally, there are approaches based on *regular model checking* which use automata-based algorithms to verify parameterized systems [4,9,10,36]. To the best of our knowledge, the CMP method is the state-of-the-art for protocol verification in contrast to these methods and has been used to successfully verify larger protocols such as Flash with minimal manual effort. (Other key methods which verify Flash protocol in full complexity are by Park *et al.* [17,33]. However, as described by Talupur *et al.* [38], these are significantly manual and take more time to finish verification of the Flash protocol compared to the CMP method.)

2 Protocols, Flows and S-deadlock Freedom: Background

2.1 Preliminaries

A protocol $\mathcal{P}$ (N) consists of N symmetric cache agents, with *ids* from the set $\mathbb{N}_N = \{1, 2, 3, \ldots, N\}$. We follow our prior approach [38] (which was inspired by the approach of Kristic [25]) in formalizing cache coherence protocols.

Index Variables: The protocol uses index variables quantified over the set of index values $\mathbb{N}_N$. Thus, if i is an index variable, then i takes values from the domain $\mathbb{N}_N$.

State Variables: The state of the protocol is encoded as local variables and global variables shared between the agents. These variables either are Boolean variables, or, are pointers which can hold agent *ids* and thus have values in $\mathbb{N}_N \cup \{null\}$ where $null$ represents that the variable does not hold any index value. The global Boolean variables are denoted by G_B and pointers by G_P. The local Boolean variables of agent i are denoted by $L_B[i]$ and the local pointer variables by $L_P[i]$.

Expressions: An expression is a, possibly quantified, propositional formula with atoms G_B, $G_P = j$, $L_B[i]$ and $L_P[i] = j$, where, i and j are index variables.

Assignments: Assignments are of the form $G_B := b$, or $G_P := j$, $L_B[i] := b$ or $L_P[i] := j$, where, b is a variable with Boolean value and i, j are index variables.

Rules: Each agent i consists of a set of rules $rl_1(i), rl_2(i), rl_3(i), \ldots, rl_k(i)$. Each rule $rl_j(i)$ can be written as: $rl_j(i) : rl_j(i).\rho \rightarrow rl_j(i).a$, where, $rl_j(i)$ is the rule name, the guard $rl_j(i).\rho$ is an expression, and $rl_j(i).a$ is a list of assignments, such that these assignments are restricted to only update the global variables or the local variables of agent i. The local variables and rules for all agents i are symmetric.

Protocol: The above defined variables and rules naturally induce a state transition system. A *protocol*, then, is a state transition system (S, Θ, T), where S is the set of protocol states, $\Theta \subseteq S$ is the set of initial states, and $T \subseteq S \times S$ is the transition relation. Each protocol state $s \in S$ is a valuation of the variables G_B, G_P, and $L_B[i]$, $L_P[i]$ for each agent i. There exists a transition $\tau(i_v) = (s, s'), (s, s') \in T$ from state s to s' if there is a rule $rl_j(i)$ and value of index variable $i = i_v$, s.t. $rl_j(i_v).\rho$ holds in s, and s' is obtained by applying $rl_j(i_v).a$ to s. In state s, we say that the rule $rl_j(i)$

is *enabled* for agent with *id* i_v if the guard $rl_j(i_v).\rho$ is *true*. When the enabled rule is executed, its action is applied to update the state and we say that the rule $rl_j(i)$ has *fired* for agent i_v. The action is applied atomically to update the state, thus the transitions of the protocol have interleaving semantics. Finally, we define an execution *trace* of the protocol as a series of transitions where each transition is a fired rule. Thus, a trace can be represented by a series ($rl_a(i_0)$, $rl_b(i_1)$, ..., $rl_s(i_k)$), where the transition $rl_m(i_n)$ is the rule rl_m fired for the agent with *id* i_n.

S-deadlock Definition. We define a protocol state s to be an *s-deadlock state* if no rule in that state is enabled. Then, a protocol is s-deadlock free if in all states, there exists at least one rule which is enabled. This can be expressed as the invariant: $\bigvee_i \bigvee_j rl_j(i).\rho$, i.e., the protocol is s-deadlock free if the disjunction of the guards of all the rules of all the agents is *true* for all the reachable states.

Flows. Flows describe the basic organization of rules for implementing the high-level requests in a protocol (for example a request for $Exclusive$ access or an $Invalidate$). We model a flow as a set of rules $\mathcal{F}(i)$ of the form $\{rl_a(i), rl_b(i), rl_c(i), \ldots, rl_n(i)\}$ which accomplish a high-level request of agent i.[1] The rules in a flow are partially ordered, with the partial order relation denoted as $\prec_{\mathcal{F}(i)}$. For example, in the $Exclusive$ flow in Fig. 1(a), the rules (arrows) are totally ordered along the downward direction. Thus $SendReqE(i) \prec_{\mathcal{F}_E(i)} RecvReqE(i)$, where $\mathcal{F}_E$ denotes the set of rules for $Exclusive$ flow. For every rule $rl_k(i)$ in the flow $\mathcal{F}(i)$, the partial order naturally induces the following *precondition*: for the rule $rl_k(i)$ to fire, all the rules preceding that rule in the partial order of the flow $\mathcal{F}(i)$ must have already been fired. This precondition is denoted by $rl_k(i).p_{\mathcal{F}(i)}$ and, formally, can be written as:

$$rl_k(i).p_{\mathcal{F}(i)} = \forall j : \big(\{(rl_j(i) \in \mathcal{F}(i)) \wedge (rl_j(i) \prec_{\mathcal{F}(i)} rl_k(i))\} \Rightarrow (rl_j(i).fired = true)\big),$$

where $rl_j(i).fired$ is an auxiliary variable which is initially set to *false* when the flow $\mathcal{F}(i)$ starts and is set to *true* when the rule $rl_j(i)$ has fired for that flow.

Designs of protocols are presented in industrial documents as a set of flows $\mathcal{F}_1(i)$, $\mathcal{F}_2(i)$, $\mathcal{F}_3(i)$, ..., $\mathcal{F}_k(i)$. In order to process a high-level request, a protocol may use a combination of these flows, e.g. in order to execute a request for $Exclusive$ access the German protocol uses the $Exclusive$ and $Invalidate$ flows. Each flow in a protocol represents an execution scenario of the protocol for processing some high-level request. Thus many of the flows of a protocol tend to exhibit a lot of similarity as they are different execution scenarios of the same high-level request. This makes them fairly easy to understand. In Section 3, we show how a set of invariants collectively implying s-deadlock freedom can be derived from these flows.

Some Definitions: We define the union of all the flows of agent i by $\mathcal{R}(i)$, i.e., $\mathcal{R}(i) = \bigcup_k \mathcal{F}_k(i)$. Next, we define the operator $\widehat{en}$ which is *true* for a set of rules, if at least one rule in the set is enabled, else it is *false*. Thus, for example, $\widehat{en}(\mathcal{R}(i))$ holds if at

[1] For ease of exposition we assume that the guard and action of a rule are over the variables of a single agent. Thus, a flow containing such rules also involves a single agent. In general, a rule and thus a flow can involve a larger but fixed number of interacting agents as well. Our approach can be easily generalized to that case.

least one of the rules in $\mathcal{R}(i)$ is enabled. In this case, we say that the agent i is enabled. Similarly, we say that a flow $\mathcal{F}(i)$ is enabled if at least one of its rules is enabled, i.e., $\widehat{en}(\mathcal{F}(i))$ holds. In case a flow $\mathcal{F}(i)$ is not enabled, we say that it is *blocked* on some rule $rl_j(i) \in \mathcal{F}(i)$ if the precondition of the rule $rl_j(i).p_{\mathcal{F}(i)}$ holds but the guard of the rule $rl_j(i).\rho$ is *false*.

2.2 German Protocol Implementation

The German protocol consists of agents such that each agent can have $Exclusive$ (E), $Shared$ (S) or $Invalid$ (I) access to a cache line, as stored in the variable $Cache[i].State$. An agent i requests these access rights by sending messages on a channel $ReqChannel[i]$ to a shared directory which sends corresponding grants along the channel $GntChannel[i]$. The directory is modeled as a set of global variables which serves one agent at a time: it stores the *id* of the agent being served in the variable $CurPtr$. It also stores the nature of the request in the variable $CurCmd$ with values in $\{ReqE, ReqS, Empty\}$, where $ReqE$ represents a request for $Exclusive$ access, $ReqS$ for $Shared$ and $Empty$ for no request. Finally, the directory tracks if $Exclusive$ access is granted to some agent or not using the variable $ExGntd$: it is *true* if access is granted and *false* otherwise. A simplified version of the code for the $Exclusive$ request is shown in Fig. 3 (full version available in [11]).

In processing the $Exclusive$ request, before sending the grant $SendGntE(i)$, the directory checks if there are any sharers of the cache line (by checking $ShrSet = \{\}$). If there are sharers, the $Invalidate$ flow is invoked for each agent in $ShrSet$. Upon invalidation of all the agents in $ShrSet$, the $ShrSet$ becomes empty and so the $SendGntE(i)$ rule becomes enabled for execution. We show the code for the $SendInv(i)$ rule below.

```
∀ i : ℕ_N; do Rule SendInv(i)
  InvChannel[i].cmd = Empty ∧ i ∈ ShrSet ∧
    ((CurCmd = ReqE) ∨ (CurCmd = ReqS ∧ ExGntd = true))
 →
  InvChannel[i].cmd := Invalidate;
End;
```

We note a condition Inv_Cond, which must be *true* for invoking the $Invalidate$ flow and can be identified from the guard of $SendInv(i)$; $Inv\_Cond : \big(((CurCmd = ReqE) \vee ((CurCmd = ReqS) \wedge (ExGntd = true))\big) \wedge (ShrSet \neq \{\})\big)$.

3 Deriving Invariants for Proving S-deadlock Freedom

In this section, we show how a set of invariants $\mathcal{I}$ can be derived from flows such that the invariants in $\mathcal{I}$ collectively imply s-deadlock freedom. At a high-level, our method tries to show s-deadlock freedom by partitioning the global state of the protocol using predicates, such that for each partition, some agent i has at least one transition enabled. Each invariant inv is of the form $inv.pred \Rightarrow \big(\forall i \in In^{inv} : \widehat{en}(\mathcal{R}(i))\big)$, where $inv.pred$ is a predicate on the global variables of the protocol, $In^{inv} \subseteq \mathbb{N}_N$ s.t. $\neg(In^{inv} = \{\})$ (this is discharged as a separate assertion for model checking) and

```
∀ i : ℕ_N; do Rule SendReqE(i)
  ReqChannel[i].cmd=Empty ∧
   (Cache[i].State=I ∨ Cache[i].State=S)
 →
  ReqChannel[i].cmd := ReqE;
End;

∀ i : ℕ_N; do Rule RecvReqE(i)
  ReqChannel[i].cmd=ReqE ∧ CurCmd=Empty
 →
  CurCmd := ReqE; CurPtr := i;
   ReqChannel[i].cmd := Empty;
End;

∀ i : ℕ_N; do Rule SendGntE(i)
  CurCmd=ReqE ∧ CurPtr=i ∧
    GntChannel[i]=Empty ∧ Exgntd=false
   ∧ ShrSet={}
 →
  GntChannel[i] := GntE; ShrSet := {i};
  ExGntd := true; CurCmd := Empty;
  CurPtr := NULL;
End;

∀ i : ℕ_N; do Rule RecvGntE(i)
  GntChannel[i]=GntE
 →
  Cache[i].State := E; GntChannel[i] := Empty;
End;
```

Fig. 3. Implementation of the $Exclusive$ Request

$\widehat{en}(\mathcal{R}(i))$ denotes a disjunction of the guards of the rules in $\mathcal{R}(i)$. *The key insight is that since* $\widehat{en}(\mathcal{R}(i))$ *has transitions from a single agent, the abstractions required for model checking* inv *for an unbounded number of agents are significantly simpler than those for checking the original s-deadlock property,*[2] *as discussed in Section 4.*

Our method iteratively model checks each invariant in $\mathcal{I}$ to refine it. Suppose, the invariant $inv \in \mathcal{I}$ fails on model checking with the state of the protocol at failure being s_f. Then, there exists some agent i_f such that when $inv.pred$ holds in s_f, $i_f \in In^{inv}$ is *true* and $\widehat{en}(\mathcal{R}(i_f))$ is *false* in s_f. This can happen due to two reasons: first, there may be a mismatch between the flow specification and the rule-based protocol description. Thus $\widehat{en}(\mathcal{R}(i_f))$ can be *false* due to a missing rule in some flow, a missing flow all together, or an implementation error: the cause for the mismatch can be discovered from the counterexample. As an example, the counterexample may show that all flows of the agent i_f are not enabled even when the agent has some rule $rl_e(i_f)$ enabled. This rule may be a part of a flow missing from the specification. Second, the invariant inv may

[2] In the case of rules involving more than one agent (say c), the corresponding invariants may involve transitions from c agents as well. Since c is small for practical protocols, the abstraction constructed for verifying such invariants will be simple as well.

fail as some flow $\mathcal{F}$ of the agent i_f is blocked (*i.e.*, it has a rule with precondition *true* but with guard *false*) as it is waiting for another flow $\mathcal{F}$' of another agent i_s to complete. As an example, for the German protocol, the $Exclusive$ flow may be blocked for agent i_f with the rule $SendGntE(i_f)$ having precondition *true* but guard *false* and waiting for an $Invalidate$ request to complete for another agent i_s in the set $Sharers$. In this case, the set $\mathcal{I}$ is refined by splitting the invariant inv.

The invariant inv is split by, (1) splitting the predicate $inv.pred$ to further partition the global state, and (2) updating the set In^{inv} for each partition. To accomplish this, the user identifies a pointer variable from G_P or $L_P[i]$ (or an auxiliary variable) $\hat{w}$, such that it has the value i_s in the failing state s_f (and so acts as a *witness* variable for i_s). The user also identifies a conflict condition $conf$ on the global state which indicates when i_s is enabled and i_f fails. This is done by using the heuristic that if the rule $rl_f(i_f)$ of flow $\mathcal{F}$ of agent i_f is blocked, $conf$ can be derived by inspecting the guard of $rl_f(i_f)$; the condition $conf$ generally is the cause for the falsification of $rl_f(i_f).\rho$. For example, for the German protocol, $conf$ is derived from the guard of $SendGntE$ and $\hat{w}$ points to some sharer which is being invalidated.

Using $conf$ and $\hat{w}$, the invariant can be split into two invariants. (1) The first invariant excludes the case when conflict happens from the original invariant, i.e., $inv1$: $(inv.pred \wedge \neg conf) \Rightarrow \big(\forall i \in In^{inv1} : \widehat{en}(\mathcal{R}(i))\big)$, where $In^{inv1} = In^{inv}$. (2) The second invariant shows that when a conflict happens, the agent pointed to by $\hat{w}$ must be enabled and so the protocol is still s-deadlock-free, i.e., $inv2$: $(inv.pred \wedge conf) \Rightarrow \big(\forall i \in In^{inv2} : \widehat{en}(\mathcal{R}(i))\big)$, where $In^{inv2} = \{i | \, (i \in \mathbb{N}_N) \wedge (i = \hat{w})\}$. For both the invariants, assertions which check that the corresponding set of indices are non-empty are also verified. For example, for $inv1$, this assertion is $(inv.pred \wedge \neg conf) \Rightarrow In^{inv1}$.

Our method derives these invariants by iteratively model checking with a small number c (3 for German protocol) of agents. (Once the invariants are derived for c agents, they are verified for an unbounded number of agents, as shown is Section 4.) This number c needs to be chosen to be large enough such that the proof of s-deadlock freedom is expected to generalize to an unbounded number of agents. For the protocols we verified, we found that as a heuristic, c should be one more than the maximum number of agents involved in processing a high-level request. For the German protocol, an $Exclusive$ request may involve two agents, a requesting agent i and an agent j getting invalidated, so we chose c to be equal to 3.

Fig. 4 shows the details of the method. It starts with an initial broad guess invariant, *true* $\Rightarrow \big(\forall i \in \mathbb{N}_N : \widehat{en}(\mathcal{R}(i))\big)$ (line 1). This indicates that in all reachable states, every agent has at least one transition enabled. As this invariant is *false*, this broad guess invariant is refined into finer invariants, using the loop. On finishing, the user is able to derive a set of invariants, $\mathcal{I}$, which collectively imply s-deadlock freedom. Further, the user is also able to derive an assertion set, $\mathcal{A}$, such that for each invariant inv in $\mathcal{I}$, an assertion in $\mathcal{A}$ checks if the set of indices In^{inv} is non-empty when $inv.pred$ holds.

Soundness of the Method. The following theorem (proof in the extended version [37]) shows that the invariants in $\mathcal{I}$ along with the assertions in $\mathcal{A}$ collectively imply s-deadlock freedom.

DERIVE_INVARIANTS($\mathcal{P}(c)$):

$\mathcal{I} = \{\mathit{true} \Rightarrow \big(\forall i \in \mathbb{N}_N : \widehat{en}(\mathcal{R}(i))\big)\}$
$\mathcal{A} = \{\}$
while $\mathcal{P}(c) \not\models \mathcal{I}$ do
  Let $inv \in \mathcal{I} : \mathcal{P}(c) \not\models inv$ and
    $inv : inv.pred \Rightarrow \big(\forall i \in In^{inv} : \widehat{en}(\mathcal{R}(i))\big)$, where, $In^{inv} \subseteq \mathbb{N}_N$
  Inspect counterexample cex and failing state s_f:
    Case 1: mismatch between flows and protocol
      Exit loop and fix flows or protocol
    Case 2: identify conflicting agents i_f and i_s s.t.
      (1) $i_f : \big((i_f \in In^{inv}) \wedge (\neg\widehat{en}(\mathcal{R}(i_f)))\big)$ holds in s_f.
      (2) $\exists rl_f \in \mathcal{F}(i_f)$ s.t. $\big(rl_f(i_f).p_{\mathcal{F}(i_f)} \wedge \neg(\widehat{en}(\mathcal{F}(i_f)))\big)$ holds in s_f.
      (3) $\widehat{en}(\mathcal{R}(i_s))$ holds in s_f.
    Identify $conf$ and witness $\hat{w}$ from above information
    $inv1 : (\neg conf \wedge inv.pred) \Rightarrow \big(\forall i \in In^{inv} : \widehat{en}(\mathcal{R}(i))\big)$
    $inv2 : (conf \wedge inv.pred) \Rightarrow \big(\forall i \in In^{inv2} : \widehat{en}(\mathcal{R}(i))\big)$, where,
      $In^{inv2} = \{i |\, i = \hat{w}\}$
    $\mathcal{I} = \{\mathcal{I} \setminus inv\} \cup \{inv1, inv2\}$
    $\mathcal{A} = \big(\mathcal{A} \setminus \big(inv.pred \Rightarrow (In^{inv} \neq \{\})\big)\big) \cup$
      $\{\big(inv1.pred \Rightarrow (In^{inv1} \neq \{\})\big), \big(inv2.pred \Rightarrow (In^{inv2} \neq \{\})\big)\}$

Fig. 4. Method for Deriving Invariants from Flows

Theorem. *If the set of invariants $\mathcal{I}$ along with the set of assertions $\mathcal{A}$ hold, they collectively imply s-deadlock freedom, i.e.,* $\big(\big(\bigwedge_{inv\in\mathcal{I}}(\mathcal{P} \models inv)\big) \wedge \big(\bigwedge_{asrt\in\mathcal{A}}(\mathcal{P} \models asrt)\big)\big) \Rightarrow \big(\mathcal{P} \models (\bigvee_i \bigvee_j rl_j(i).\rho)\big)$.

3.1 Specifying Invariants for the German Protocol

We derive the invariants for a model of the German protocol with 3 cache agents. We start with the initial invariant that for all agents, some flow is enabled, i.e., INV-1: $true \Rightarrow \big(\forall i \in \mathbb{N}_N : \widehat{en}(\mathcal{R}(i))\big)$.

Iteration 1: Model checking the invariant INV-1 returns a counterexample trace ($SendReqE(1)$, $RecvReqE(1)$, $SendReqE(2)$). Since the index of the last rule in the trace is 2, $\widehat{en}(\mathcal{R}(2))$ must be *false*. This is because the rule $RecvReqE(2)$ of the $Exclusive$ flow of cache 2 is not fired and thus has precondition *true* but guard *false*. The user identifies the conflict condition $conf = \neg(CurCmd = Empty)$ from the guard of the blocked rule $RecvReqE(2)$. Since $CurPtr$ is the witness pointer in the protocol for the variable $CurCmd$, the witness $\hat{w}$ is set to $CurPtr$. Thus, the invariant is split as follows:

- INV-1.1: $(CurCmd = Empty) \Rightarrow (\forall i \in \mathbb{N}_N : \widehat{en}(\mathcal{R}(i)))$.
- INV-1.2: $\neg(CurCmd = Empty) \Rightarrow (\forall i \in In^{inv-1.2} : \widehat{en}(\mathcal{R}(i)))$, where $In^{inv-1.2} = \{i |\, (i \in \mathbb{N}_N) \wedge (i = CurPtr)\}$. The assertion $\neg(CurCmd = Empty) \Rightarrow \neg(In^{inv-1.2} = \{\})$ is also checked.

Iteration 2: Next, on model checking the invariants INV-1.1 and INV-1.2, the invariant INV-1.2 fails. The counterexample trace returned is ($SendReqE(1)$, $RecvReqE(1)$,

$SendGntE(1)$, $SendReqE(2)$, $RecvReqE(2)$, $SendReqE(2)$). Since the last rule of the counterexample is from cache 2, $\widehat{en}(\mathcal{R}(2))$ must be *false* even when $CurPtr =$ 2. Further, there are two flows for two $Exclusive$ requests by cache 2 active in the counterexample, the first with $SendReqE(2)$ fired and the second with $SendReqE(2)$, $RecvReqE(2)$ fired. Since the first flow is blocked on the rule $RecvReqE(2)$, the guard of this rule is inspected. The guard is *false* as $CurCmd$ is not empty. However, since the corresponding witness variable for $CurCmd$ is $CurPtr$ which is already 2 (due to the processing of the second flow), this is not a conflict with another cache. The conflict must then be for the second $Exclusive$ flow. The second flow is blocked on the rule $SendGntE(2)$ with precondition *true* but guard *false*: the user identifies the conflict condition $conf$ from the guard of $SendGntE$ to be Inv_Cond. Now, if Inv_Cond is *true*, the $Invalidate$ flow for some sharer cache (cache 1 in this trace) must be active. Thus, the user identifies $\hat{w}$ to point to a sharer which must be invalidated: this is done using the auxiliary variable $Sharer$, which points to the last sharer to be invalidated in $ShrSet$. Thus, the invariant INV-1.2 is split as follows:

- INV-1.2.1: $\big(\neg(CurCmd = Empty) \wedge (\neg Inv\_Cond)\big) \Rightarrow (\forall i \in In^{inv-1.2.1} : \widehat{en}(\mathcal{R}(i)))$, where, $In^{inv-1.2.1} = In^{inv-1.2}$. An assertion that the precondition implies the index set is non-empty is also checked.
- INV-1.2.2: $\big(\neg(CurCmd = Empty) \wedge (Inv\_Cond)\big) \Rightarrow (\forall i \in In^{inv-1.2.2} : \widehat{en}(\mathcal{R}(i)))$, where, $In^{inv-1.2.2} = \{i |\, (i \in \mathbb{N}_N) \wedge (i \in ShrSet)\}$. An assertion that the precondition implies the index set is non-empty is also checked.

Iteration 3: Next, on model checking, the invariants INV-1.1, INV-1.2.1, INV-1.2.2, along with the added assertions hold for a model with 3 caches. Then, to prove s-deadlock freedom, this set of invariants form a candidate set to verify a protocol model with an unbounded number of agents. The property is checked for unbounded agents using techniques described in Section 4.

4 Verifying Flow Properties for Unbounded Agents

We now show how to verify the invariants in $\mathcal{I}$ for an unbounded number of agents by leveraging the data-type reduction abstraction along with the CMP method.

Abstraction: Data-Type Reduction. Since the invariant is of the form $inv.pred \Rightarrow \big(\forall i \in In^{inv} : \widehat{en}(\mathcal{R}(i))\big)$, by symmetry, it is sufficient to check: $inv.pred \Rightarrow \big((1 \in In^{inv}) \Rightarrow \big(\widehat{en}(\mathcal{R}(1))\big)\big)$. In order to verify this invariant, just the variables of agent 1 are required. Then, our abstraction keeps just the agent 1, and discards the variables of all the other agents by replacing them with a state-less *environment* agent. We refer to agent 1 as a *concrete* agent and the environment as *Other* with *id o*.

In the original protocol, since all the agents other than agent 1 interact with it by updating the global variables, the actions of these agents on the global variables are over-approximated by the environment agent. This environment agent does not have any local state. The construction of this agent *Other* is automatic and accomplished syntactically: further details on the automatic construction are available in [38]. The final constructed abstraction then consists of: (1) a concrete agent 1, (2) an environment agent *Other* with *id* o, and (3) invariants specified on variables of agent 1 and global variables. This abstraction is referred to as *data-type reduction*. If the original protocol is $\mathcal{P}$, and invariant set $\mathcal{I}$, we denote this abstraction by $data\_type$ and thus the abstract model by $data\_type(\mathcal{P})$ and the abstracted invariants on agent 1 by $data\_type(\mathcal{I})$.

Abstraction for German Protocol. We now describe how the rule $SendGntE(i)$ gets abstracted in $data\_type(\mathcal{P})$. In the abstract model, there is one concrete agent 1, which has the rule $SendGntE(1)$. Next, $SendGntE(o)$ is constructed as follows. (1) The guard is abstracted by replacing all atoms consisting of local variables (e.g. $GntChannel[i] = Empty$) with *true* or *false* depending on which results in an over-abstraction and by replacing any usage of i in atoms with global variables (e.g. $CurPtr = i$) with o (i.e. $CurPtr = o$). (2) The action is abstracted by discarding any assignments to local variables. Further, assignments to global pointer variables are abstracted as well: any usage of i (e.g. $CurPtr := i$) is replaced by o (i.e. $CurPtr := o$). The rule for agent *Other* is shown below:

```
Rule SendGntE(o)
    CurCmd = ReqE ∧ CurPtr = o ∧ true ∧ Exgntd = false ∧
     ShrSet = {}
  →
    no-op; ShrSet := {o}; ExGntd := true;
    CurCmd := Empty; CurPtr := NULL;
End;
```

The Abstraction-Refinement Loop of the CMP Method. The CMP method works as an abstraction-refinement loop, as shown in Fig. 5. In the loop, the protocol and invariants are abstracted using data-type reduction. If the proof does not succeed, the user inspects the returned counterexample cex and following possibilities arise. (1) Counterexample cex is real, in which case an error is found and so the loop exits. (2) Counterexample cex is spurious and so the user refines the protocol by adding a *non-interference lemma* lem. The function *strengthen* updates the guard $rl_j(i).\rho$ of every rule $rl_j(i)$ of the protocol to $rl_j(i).\rho \wedge lem(j)$; this way, on re-abstraction with $data\_type$ in line 1, the new abstract protocol model is refined. Additional details on the CMP method are available in [11, 25].

CMP($\mathcal{P}(N), \mathcal{I}$)
1: $\mathcal{P}^{\#} = \mathcal{P}(N); \mathcal{I}^{\#} = \mathcal{I}$
2: $while\ data\_type(\mathcal{P}^{\#}) \not\models data\_type(\mathcal{I}^{\#})\ do$
3: examine counterexample cex
4: if cex is real, exit
5: if spurious:
6: find lemma $lem = \forall i.lem(i)$
7: $\mathcal{P}^{\#} = strengthen(\mathcal{P}^{\#}, lem)$
8: $\mathcal{I}^{\#} = \mathcal{I}^{\#} \cup lem$

Fig. 5. The CMP method

5 Experiments

Using our approach, we verified Murphi (CMurphi 5.4.6) implementations of the German and Flash protocols (available online [31]). Our experiments were done on a 2.40 GHz Intel Core 2 Quad processor, with 3.74 GB RAM, running Ubuntu 9.10.

German Protocol. We verified the invariants discussed in Section 3.1, in order to prove s-deadlock freedom. We chose to use an abstraction with 2 agents and an environment agent, so that the mutual exclusion property can also be checked.

The proof finished in 217s with 7M states explored. No non-interference lemmas were required to refine the model, in order to verify the invariants presented in Section 3.1. Since typically protocols are also verified for properties like data integrity (i.e. the data stored in the cache is consistent with what the processors intended to write) and mutual exclusion, we model checked the above invariants along with these properties. In this case, the abstract model was constrained and model checking this model was faster and took 0.1 sec with 1763 states explored.

Buggy Version. We injected a simple error in the German protocol in order to introduce an s-deadlock. In the bug, an agent being invalidated drops the acknowledgement $SendInvAck$ it is supposed to send to the directory. This results in the entire protocol getting blocked, hence an s-deadlock situation. This was detected by the failing of the invariant INV-1.2.2, discussed in Section 3.1.

Flash Protocol. Next, we verified the Flash protocol [26] for deadlock freedom. The Flash protocol implements the same high-level requests as the German protocol. It also uses a directory which has a Boolean variable $Pending$ which is *true* if the directory is busy processing a request from an agent pointed to by another variable $CurSrc$ (name changed from original protocol for ease of presentation). However, the Flash protocol uses two key optimizations over the German protocol. First, the Flash protocol enables the cache agents to directly forward data between each other instead of via the directory, for added speed. This is accomplished by the directory by forwarding incoming requests from the agent i to the destination agent, $FwDst(i)$, with the relevant data. Second, the Flash protocol uses non-blocking invalidates, i.e, the $Exclusive$ flow does not have to wait for the $Invalidate$ flow to complete for the sharing agents in $ShrSet$. Due to these optimizations, the flows of the Flash protocol are significantly more complex than those of German protocol. Further, due to forwarding, some rules involve two agents instead of one for the German protocol: thus the flows involve two agents as well. Each flow then is of the form $\mathcal{F}_k(i, j)$, where i is the requesting agent for a flow and $j = FwDst(i)$ is the destination agent to which the request may be forwarded by the directory. Then, we define $\mathcal{R}(i)$ to be equal to $\bigcup_k \mathcal{F}_k(i, FwDst(i))$.

We derived the invariants from the flows by keeping c to be equal to 3, as each request encompasses a maximum of 2 agents (forwarding and invalidation do not happen simultaneously in a flow). The final invariants derived using our method are as follows:

Directory Not Busy: If the directory is not busy (i.e., $Pending$ is *false*), any agent i can send a request. Thus the invariant INVF-1: $\neg(Pending) \Rightarrow \big(\forall i \in \mathbb{N}_N : \widehat{en}(\mathcal{R}(i))\big)$.

However, if the directory is busy (i.e., $Pending$ is *true*), two possibilities arise. (1) It may be busy since it is processing a request from agent $CurSrc$. Or, (2) in case the request from $CurSrc$ requires an invalidate, the directory may remain busy with invalidation even after the request from $CurSrc$ has been served. This is because Flash allows the request from $CurSrc$ to complete before invalidation due to non-blocking invalidates. Hence the following invariants:

Directory Busy with Request: Invariant INVF-2: $\big((Pending) \wedge (ShrSet = \{\})\big) \Rightarrow \big(\forall i \in In^{invF-2}\widehat{en}(\mathcal{R}(i))\big)$, where $In^{invF-2} = \{i | \, (i \in \mathbb{N}_N) \wedge (i = CurSrc)\}$.

Directory Busy with Invalidate: Invariant INVF-3: $\big((Pending) \wedge \neg(ShrSet = \{\})\big) \Rightarrow \big(\forall i \in In^{invF-3}\widehat{en}(\mathcal{R}(i))\big)$, where $In^{invF-3} = \{i | \, (i \in \mathbb{N}_N) \wedge (i \in ShrSet)\}$.

Runtime: We verified the above invariants along with the mutual exclusion and the data integrity properties for an unbounded model abstracted by keeping 3 concrete agents (one agent behaves as a directory) and constructing an environment agent *Other*. The verification took 5127s with about 20.5M states and 152M rules fired. In this case we reused the lemmas used in prior work by Chou *et al.* [11] for verifying the mutual exclusion and data integrity properties in order to refine the agent *Other*.

Verifying Flash vs German Protocol: The flows of the Flash protocol involve two indices: we eliminated the second index by replacing it with the variable $FwDst(i)$ which stores information of the forwarded cache and thus made the verification similar to the German protocol case. Next, Flash protocol uses lazy invalidate: even if the original request has completed, the directory may still be busy with the invalidate. As explained above, this was in contrast to the German protocol and resulted in an additional invariant INVF-3.

Comparison with Other Techniques: The only technique we are aware of which handles Flash with a high degree of automation is by Bingham *et al.* [8]. While a direct comparison of the runtime between their approach and ours is infeasible for this paper, we note that the invariants generated using our approach only require an over-abstraction in contrast to theirs which requires a mixed-abstraction. This is an advantage since development of automatic and scalable over-abstraction based parameterized safety verification techniques is a promising area of ongoing research (e.g. [15]) which our approach directly benefits from.

6 Conclusions and Future Work

In this paper we have presented a method to prove freedom from a practically motivated deadlock error which spans the entire cache coherence protocol, an s-deadlock. Our method exploits high-level information in the form of message sequence diagrams—these are referred to as *flows* and are readily available in industrial documents as charts and tables. Using our method, a set of invariants can be derived which collectively imply s-deadlock freedom. These invariants enable the direct application of industrial scale techniques for parameterized verification.

As part of future work, we plan to take up verification of livelock freedom by exploiting flows. Verifying livelock requires formally defining a notion of the protocol doing useful work. This information is present in flows—efficiently exploiting this is part of our ongoing research.

Acknowledgment. This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA. The authors would also like to thank Sayak Ray for his comments which were very helpful in improving this paper.

References

1. Abadi, M., Lamport, L.: Composing specifications. ACM Trans. Program. Lang. Syst. 15(1), 73–132 (1993), http://doi.acm.org/10.1145/151646.151649
2. Abadi, M., Lamport, L.: Conjoining specifications. ACM Trans. Program. Lang. Syst. 17(3), 507–535 (1995), http://doi.acm.org/10.1145/203095.201069
3. Abdulla, P., Haziza, F., Holk, L.: A model-constructing satisfiability calculus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 1–12. Springer, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-35873-9_28
4. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004), http://dx.doi.org/10.1007/978-3-540-28644-8_3
5. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.D.: Parameterized verification with automatically computed inductive assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 221–234. Springer, Heidelberg (2001), http://dl.acm.org/citation.cfm?id=647770.734120
6. Baukus, K., Lakhnech, Y., Stahl, K.: Parameterized verification of a cache coherence protocol: Safety and liveness. In: Cortesi, A. (ed.) VMCAI 2002. LNCS, vol. 2294, pp. 317–330. Springer, Heidelberg (2002), http://dl.acm.org/citation.cfm?id=646541.696180
7. Bingham, B., Bingham, J., Erickson, J., Greenstreet, M.: Distributed explicit state model checking of deadlock freedom. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 235–241. Springer, Heidelberg (2013)
8. Bingham, B., Greenstreet, M., Bingham, J.: Parameterized verification of deadlock freedom in symmetric cache coherence protocols. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design, FMCAD 2011, pp. 186–195. FMCAD Inc, Austin (2011), http://dl.acm.org/citation.cfm?id=2157654.2157683
9. Boigelot, B., Legay, A., Wolper, P.: Iterating transducers in the large. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 223–235. Springer, Heidelberg (2003), http://dx.doi.org/10.1007/978-3-540-45069-6_24
10. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000), http://dl.acm.org/citation.cfm?id=647769.734106
11. Chou, C.-T., Mannava, P.K., Park, S.: A simple method for parameterized verification of cache coherence protocols. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 382–398. Springer, Heidelberg (2004)
12. Clarke, E.M., Grumberg, O., Browne, M.C.: Reasoning about networks with many identical finite-state processes. In: Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC 1986, pp. 240–248. ACM, New York (1986), http://doi.acm.org/10.1145/10590.10611
13. Clarke, E., Talupur, M., Veith, H.: Proving ptolemy right: the environment abstraction framework for model checking concurrent systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 33–47. Springer, Heidelberg (2008), http://portal.acm.org/citation.cfm?id=1792734.1792740

14. Clarke, E., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 126–141. Springer, Heidelberg (2006), http://dx.doi.org/10.1007/11609773_9
15. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaidi, F.: Invariants for finite instances and beyond. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 61–68 (October 2013)
16. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM Trans. Program. Lang. Syst. 19(2), 253–291 (1997), http://doi.acm.org/10.1145/244795.244800
17. Das, S., Dill, D., Park, S.: Experience with predicate abstraction. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 160–171. Springer, Heidelberg (1999), http://dx.doi.org/10.1007/3-540-48683-6_16
18. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 236–254. Springer, Heidelberg (2000), http://dl.acm.org/citation.cfm?id=648236.753642
19. Emerson, E.A., Kahlon, V.: Exact and efficient verification of parameterized cache coherence protocols. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 247–262. Springer, Heidelberg (2003)
20. Emerson, E.A., Namjoshi, K.S.: Automatic verification of parameterized synchronous systems (extended abstract). In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 87–98. Springer, Heidelberg (1996), http://dl.acm.org/citation.cfm?id=647765.735841
21. Fang, Y., Piterman, N., Pnueli, A., Zuck, L.: Liveness with invisible ranking. Int. J. Softw. Tools Technol. Transf. 8(3), 261–279 (2006), http://dx.doi.org/10.1007/s10009-005-0193-x
22. Holt, R.C.: Some deadlock properties of computer systems. ACM Comput. Surv. 4(3), 179–196 (1972), http://doi.acm.org/10.1145/356603.356607
23. Ip, C.N., Dill, D.L.: Better verification through symmetry. In: Proc. Conf. on Computer Hardware Description Languages and their Applications, pp. 97–111 (1993)
24. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 645–659. Springer, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-14295-6_55
25. Kristic, S.: Parameterized system verification with guard strengthening and parameter abstraction. In: 4th Int. Workshop on Automatic Verification of Finite State Systems (2005)
26. Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., Hennessy, J.: The stanford flash multiprocessor. In: Proceedings the 21st Annual International Symposium on Computer Architecture, pp. 302–313 (1994)
27. Lahiri, S.K., Bryant, R.E.: Predicate abstraction with indexed predicates. ACM Trans. Comput. Logic 9(1) (2007), http://doi.acm.org/10.1145/1297658.1297662
28. Mcmillan, K.L.: Parameterized verification of the flash cache coherence protocol by compositional model checking. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 179–195. Springer, Heidelberg (2001)
29. McMillan, K.L.: Circular compositional reasoning about liveness. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 342–346. Springer, Heidelberg (1999), http://dl.acm.org/citation.cfm?id=646704.701881
30. McMillan, K.L.: Verification of infinite state systems by compositional model checking. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 219–237. Springer, Heidelberg (1999), http://dl.acm.org/citation.cfm?id=646704.702020
31. Murphi source code, https://github.com/dsethi/ProtocolDeadlockFiles

32. O'Leary, J., Talupur, M., Tuttle, M.R.: Protocol verification using flows: An industrial experience. In: Formal Methods in Computer-Aided Design, FMCAD 2009, pp. 172–179 (November 2009)
33. Park, S., Dill, D.L.: Verification of flash cache coherence protocol by aggregation of distributed transactions. In: SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures, pp. 288–296. ACM Press (1996)
34. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001), http://dl.acm.org/citation.cfm?id=646485.694452
35. Pnueli, A., Xu, J., Zuck, L.D.: Liveness with $(0, 1, \infty)$-counter abstraction. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 107–122. Springer, Heidelberg (2002), http://dl.acm.org/citation.cfm?id=647771.734286
36. Resten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 424–435. Springer, Heidelberg (1997), http://dx.doi.org/10.1007/3-540-63166-6_41
37. Sethi, D., Talupur, M., Malik, S.: Using flow specifications of parameterized cache coherence protocols for verifying deadlock freedom. ArXiv:1407.7468
38. Talupur, M., Tuttle, M.R.: Going with the flow: Parameterized verification using message flows. In: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design, FMCAD 2008. FMCAD '08, pp. 1–10. IEEE Press, Piscataway (2008), http://dl.acm.org/citation.cfm?id=1517424.1517434