Going with the Flow: Parameterized Verification using Message Flows

Murali Talupur
Intel
murali.talupur@intel.com

Mark R. Tuttle
Intel
tuttle@acm.org

***Abstract*—A message flow is a sequence of messages sent among processors during the execution of a protocol, usually illustrated with something like a message sequence chart. Protocol designers use message flows to describe and reason about their protocols. We show how to derive high-quality invariants from message flows and use these invariants to accelerate a state-of-the-art method for parameterized protocol verification called the CMP method. The CMP method works by iteratively strengthening and abstracting a protocol. The labor-intensive portion of the method is finding the protocol invariants needed for each iteration. We provide a new analysis of the CMP method proving it works with any sound abstraction procedure. This facilitates the use of a new abstraction procedure tailored to our protocol invariants in the CMP method. Our experience is that message-flow derived invariants get to the heart of protocol correctness in the sense that only couple of additional invariants are needed for the CMP method to converge.**

I. Introduction

Invariants play a crucial role in most approaches to verification, but finding good invariants is a challenging problem. The methods proposed for finding invariants fall into roughly two classes. The manual approach is to begin with a simple candidate for an invariant, and to elaborate the invariant based on counterexamples produced by a mechanical checker or prover. There are also automatic approaches such as invisible invariants [23], [4], indexed predicates [16], interpolant-based invariant generation [21], and split invariants [12] that try to deduce invariants more or less automatically. In both approaches, the resulting invariant is often unwieldy and rarely insightful. The invariants are usually low-level expressions formulated in terms of the protocol variables and the high-level understanding of why the protocol works is either unused or obscured.

In this paper, we show that *message flows* are a succinct and readily available source of high-level invariants that usually go unused in protocol verification. A message flow is a sequence of messages sent among processors following a protocol that logically constitutes a single transaction of the protocol. Flows are often illustrated by protocol designers in the form of message sequence charts. Figure 1, for example, illustrates two flows from the German cache coherence protocol.

Message flows are linear, local, and typically involve only a small number of processes. They arise naturally in the context of protocols and embody local ordering relations among the actions of different agents. Message flows are found everywhere from protocol design documents to conference papers to informal explanations on white boards. Yet, in spite being a compelling way of understanding protocols, flows almost never appear in that ultimate explanation of why a protocol works, namely, the correctness proof itself. We show that message flows easily yield high-quality invariants, and use them to accelerate a state-of-the-art approach to parameterized protocol verification called the CMP method.

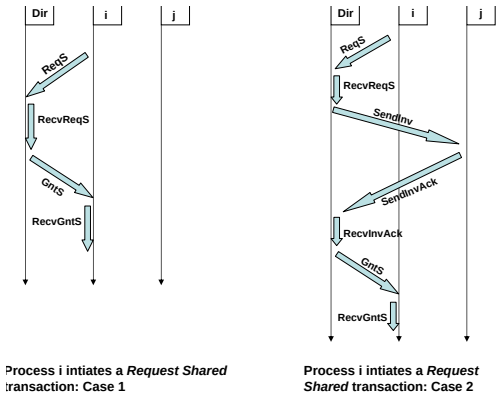

Fig. 1. Two typical message flows

The *CMP method* [20], [9], [15] is one of the most successful methods for parameterized protocol verification. Most protocols are naturally described in terms of a few parameters such as the number of processors n. Parameterized verification is the problem of proving protocol correctness for all values of n, and not just the small values that can be checked with modern model checkers. The CMP approach to parameterized verification is a combination of *data type reduction* [19] and *compositional reasoning* [18] first proposed by McMillan [18], later elaborated by Chou, Mannava, and Park [9], and formalized by Krstic [15]. In this approach, a model checker is used as proof assistant and the user guides the proof by supplying invariants or "noninterference lemmas." Speaking informally, CMP works by taking a protocol for n processors, retaining two or three processors and replacing the remaining processors with a single, highly-nondeterministic processor called $Other$, and then formulating a set of protocol invariants that serve to constrain the behavior of $Other$ sufficiently to allow model checking to validate the desired property.

The CMP method scales very well. It has been used to verify Flash protocol [20], [9], a protocol sufficiently complex that

only two or three other methods have been able to fully validate the protocol parametrically. At Intel, we have used CMP to verify an industrial-strength cache protocol several orders of magnitude larger than even the Flash protocol [24]. Although manual guidance is necessary in the form of noninterference lemmas, the lemmas supplied by the user do not have to add up to an inductive invariant. Consequently, CMP is much easier on the user than other theorem-proving style methods such as [22].

The hardest part of using CMP is finding a set of protocol invariants that enable CMP to converge. We show that invariants derived from message flows constrain the *Other* process so well that the burden of manual addition of further invariants is significantly reduced. We could verify the control and data safety properties of German's protocol and Flash protocol by adding manually at most two lemmas. Augmenting the basic CMP method with flow based invariants thus results in an *even more scalable and a much easier to use method.*

In this paper we make the following contributions:

1) *We show how to derive invariants from message flows.* We give a simple language for describing message flows, and show how adding a few auxiliary variables to the protocol description allows us to describe these message flows as ordinary state predicates.
2) *We show that CMP can be generalized.* The CMP method works by iteratively *strengthening* and *abstracting* a protocol. We provide a new analysis of CMP that cleanly separates the strengthening and abstraction procedures used by CMP. We prove that CMP works with any sound abstraction procedure, in contrast to the original analysis that depended heavily on protocol symmetry and the use of data type reduction as the abstraction procedure.
3) *We show how to use message flow invariants with CMP.* We provide a new abstraction procedure for use with our message flow invariants, and show how to use them with CMP.
4) *We demonstrate that message flow invariants simplify and accelerate CMP.* We apply our ideas to the German and Flash protocols. We show that starting with the invariants derived from message flows, couple of additional invariants are enough for CMP to validate both the data and control properties, in contrast to the several complex invariants needed in the original application of CMP [9], and in contrast to recent automated methods [7], [17] that can verify only the control safety properties.

A. Related Work

Parameterized verification, mainly of cache coherence and mutual exclusion protocols, has received considerable attention. Recent methods have been based on invisible invariants [4], counter abstraction [23], indexed predicates [16], ordinary model checking [3], WS1S [5], learning [14], strengthening split invariants [12], and environment abstraction [10], [11]. These techniques have had varying degrees of success, but none of them has been applied to a large industrial-strength protocol like Flash (although environment abstraction [11] has been applied to a simplified version of Flash).

McMillan introduced the CMP method in a series of papers [18], [20], [9] (the proof in [20] does not completely follow the CMP framework: to get BDDs to scale, the Flash model had to be pruned by hand.) Chou, Mannava, and Park [9] elaborated on the method and showed how it can be performed in conjunction with any model checker. Krstic [15] gave a formalization of the method.

As far as we are aware, the CMP method is one of the few methods to handle the full complexity of the Flash protocol. A transaction-based method [22] and a predicate abstraction-based method [13] have both verified the safety properties of Flash. Both the methods required extensive manual guidance and took several days to complete the verification. Recently, some nearly automatic methods based on BDDs have been proposed [17], [7]. As with the other BDD-based methods [20], they have trouble scaling up: they have been able to verify only a control property, but not data. For all of these methods applied to Flash, model checking has taken roughly a couple hours. In contrast, our work is able to check both the data and control properties of Flash in just 120 seconds.

We note that our method of deriving invariants from message flows is not specific to CMP, and can potentially be used in conjunction with any other method of protocol verification.

Finally, the compositional reasoning principle at the heart of our new analysis of CMP is similar to principles proposed by Abadi and Lamport [1], [2], McMillan [18], Krstic [15], Bhattacharya et al. [6] and Chen et al. [8].

We introduce preliminary concepts in Section II, derive protocols invariants from message flows in Section III, present a new analysis of CMP in Section IV, show how to use flow-derived invariants with CMP in Section V, give experimental results in Section VI, and conclude with Section VII.

II. Parameterized Protocols

A parameterized protocol $P(N)$ is a protocol for N processes with unique ids in $\mathbb{N}_N = [1..N]$. We follow Krstic [15] and define the simplest model sufficient to express cache coherence protocols, but our work depends only on the protocol being symmetric and described with guarded commands.

Index variables: Fix a set $\mathcal{I}$ of index variables for quantifying over process ids. When we write $T(i, j)$, we mean that i and j are the only index variables appearing free in T.

States: The state of a protocol consists of global and local variables that can hold either Boolean values or process ids. A global variable (like a directory entry) is a scalar variable, and a local variable (like a cache entry) is an array variable indexed by process id. Formally, the state is determined by four sets of variables, W, X, Y, Z where variables in W are of type $\mathbb{B}$, in X are of type $\mathbb{N}_N \to \mathbb{B}$, in Y are of type $\mathbb{N}_N$ and those in Z are of type $\mathbb{N}_N \to \mathbb{N}_N$. Note that the types of the variables are determined by the parameter N.

Expressions: An expression is made up of combination of the four *basic expressions*

$$w \qquad y = i \qquad x[i] \qquad z[i] = j$$

where $i, j \in \mathcal{I}$ are index variables and $w \in W, x \in X, y \in Y, z \in Z$ in the usual fashion by taking their Boolean combinations and possibly quantifying out some free index variables.

Assignments: An assignment is one of the four *basic assignments*

$$w := b \qquad y := i \qquad x[i] := b \qquad z[i] := j$$

where $i, j \in \mathcal{I}$, where b is either *true* or *false*, and $w \in W, x \in X, y \in Y, z \in Z$, together with the *quantified assignment*

$$\forall I.\phi \Rightarrow v := e$$

where $v := e$ is one of the basic assignments. The latter is just a set of conditional assignments: consider all possible assignments of values to indices in I, and for each such assignment, perform $v := e$ if ϕ is true.

Rules: A rule is a *guarded command* written as

$$rl : \forall i, j.\rho \rightarrow a \qquad \text{or} \qquad rl(i, j) : \rho(i, j) \rightarrow a(i, j)$$

where rl is the *rule name*, ρ is an expression called the *guard*, and a is a list of assignments called the the *action*. We assume that i and j are the only free index variables in ρ and a. Note that we use the ordered pair (i, j) to emphasize that the ordering of these indices is important. We assume that an action never assigns two values to the same variable, ruling out asymmetric assignments like $\forall k.true \Rightarrow y := k$.

Protocols: A *protocol* is a state transition system (S, Θ, T) where S and Θ are the sets of states and initial states and $T \subseteq S \times S$ is the transition relation given by a set of rules. There is a transition from s to s' if there is a rule $rl(i, j) : \rho(i, j) \rightarrow a(i, j)$ and an instantiation p, q such that s satisfies $\rho(p, q)$ and s' is the result of starting with s and performing the assignments in $a(p, q)$.

From these restrictions on expressions and assignments it is clear that the protocol P must be symmetric. The following lemma, which simply restates symmetry in a different form, will be used later. Let R_i denote the set of all reachable states of P within i steps.

Lemma 1: For all $i \geq 0$ and $M \leq N$:

- $\forall s \in R_i.s \models \phi(1, .., M)$ implies $\forall s \in R_i.s \models \forall i_1, .., i_M.\phi(i_1, .., i_M)$.
- $P \models \phi(1, .., M)$ implies $P \models \forall i_1, .., i_M.\phi(i_1, .., i_M)$.

III. Message Flows

Protocol designers use message flows, like the two flows from German's protocol illustrated in Figure 1, to describe the basic organization of a protocol and to reason about the protocol itself. Message flows implicitly impose constraints on the order in which the actions appearing within them can happen. In the flow on the left of Figure 1, the cache sends the directory a *ReqS* message (requesting shared access). The directory executes a *RecvReqS* action to receive the *ReqS* message, and then sends the cache a *GntS* message (granting shared access). Finally, the requesting cache does a *RecvGntS* action to complete the transaction. Each action can happen only after the preceding action has been executed. For instance, the directory cannot send a *GntS* message before it has executed a *RecvReqS* action. In the flow on the right of Figure 1, the directory can send a *GntS* message only after all the *SendInv* subflows have completed. Thus, flows impose ordering constraints on the rules and subflows appearing in them, and the protocol invariants we define in this section are intended to capture these constraints.

ReqShare(i):
SendReqS(i), RecvReqS(i), **SendInval(k),**
SendGntS(i), RecvGntS(i)
ReqExcl(i):
SendReqE(i), RecvReqE(i), **SendInval(k),**
SendGntE(i), RecvGntE(i)
SendInval(i):
SendInv(i), SendInvAck(i), RecvInvAck(i)

Fig. 2. Three flows from German's protocol. Flow names are in bold.

A. Language for Describing Flows

Each flow is of the form:

$$fl(i, j) : a_1, \ldots, a_n, \ldots, a_m$$

where fl is the name of a flow with two parameters i and j from $\mathcal{I}$ and each a_k is either

- a *rule* of the form $rl(i, j)$ where rl is name of a ruleset with two parameters instantiated with i and j, or
- a *subflow* of the form $sfl(k, l)$ where sfl is the name of another flow in the system with two parameters instantiated with k and l, possibly different from i and j.

We assume for the sake of exposition that every flow and rule has two parameters. In practice, the definition of a rule $rl(i, j)$ might omit reference to one of the parameters and reduce to $rl(i)$ or $rl(j)$, but the modifications to our presentation required to deal with this possibility are elementary. A rule $rl(i, j)$ appearing in a flow represents just a single firing of the rule rl with the quantifiers instantiated to i and j. A subflow $sfl(k, l)$ appearing in a flow can be instantiated zero or more times with varying instantiations of the parameters. For example, Figure 2 illustrates three flows from German's protocol, including the *ReqShare* flows illustrated in Figure 1 in which the subflow *SendInval* occurs zero times on the left and one time on the right.

The reason for allowing flows to occur within other flows is as follows. While flows in a protocol usually involve at most two agents, some times it can happen that additional processes get involved as well. For instance, this happens in the *ReqShare* transaction shown on the right side of Figure 1. The directory has to send an invalidate message to any process holding the data item in an exclusive state. This series of rule firings involving agents different from the primary agents of the flow is modeled in our language by having a *subflow* inside a flow. Note from Figure 2 that the flow *SendInval* occurs in two different flows. In case of the *ReqExcl* transaction, several

$update(p, fl(i,j), rl(i,j))$ =
  assert($p \in \{i,j\}$);
  let $last\_rule$ =
    if $rl'(i,j)$ preceeds $rl(i,j)$ in $fl(i,j)$
    then $\{(fl, rl')\}$
    else $\{\}$;
  let $next\_rule$ =
    if $rl(i,j)$ ends $fl(i,j)$
    then $\{\}$
    else $\{(fl, rl)\}$;
  $Aux(p) := Aux(p) \setminus last\_rule \cup next\_rule$

Fig. 3. Tracking flows with auxiliary variables. If $rl(i,j)$ is a rule appearing in a flow $fl(i,j)$, then $update(p, fl(i,j), rl(i,j))$ updates the value of $Aux(p)$ when the rule fires. Since $Aux(p)$ is a multiset, the set difference operator removes just one copy of $last\_rule$.

instantiations of *SendInval* flow might be forked from the main flow. In case of the *ReqShare* transaction at most only one instantiation of *SendInval* is spawned from the main flow.

Given a cache coherence protocol we can associate a set of *flows* $\{fl_1, fl_2, .., fl_k\}$ with it. For this set of flows to be *legal*, there should be a topological ordering $\prec$ of the flows such that a flow fl_m can occur inside another flow fl_n only if $fl_m \prec fl_n$. This prevents circular specification of flows.

Remark 1: We can enrich our language by specifying the precise number of times a subflow can be instantiated within a flow. Or by having more than two entities be involved with in the main part of a flow. While many such generalizations are possible, the information contained in these simple flows is enough for the application in this paper.

B. Tracking Flows using Auxiliary Variables

We add auxiliary variables to a protocol to track the state of the various flows. Let P be a protocol, and let P_a be the augmented protocol obtained by adding auxiliary variables to P as follows. For each process p, let $Aux(p)$ be a multiset consisting of pairs (fl, rl) where $fl(i,j)$ is a flow and $rl(i,j)$ is a rule appearing in $fl(i,j)$. Initially, $Aux(p)$ is empty.

Whenever a rule instantiation $rl(p_1, p_2)$ fires the sets $Aux(p_1)$ and $Aux(p_2)$ are updated using the procedure given in Figure 3 by calling the $update(p, fl(p_1,p_2), rl(p_1,p_2)$ for both $p = p_1$ and $p = p_2$. In the update procedure we say a rule $rl'(i,j)$ or subflow $sfl(k,l)$ *immediately precedes* or *precedes* $rl(i,j)$ in flow $fl(i,j)$ if there is no other rule between them (although there can be subflows between them).

C. Noninterference Lemmas from Flows

A flow implicitly orders the rules appearing in the flow. A flow $fl(i,j)$ induces two preconditions on the firing of any rule $rl(i,j)$ appearing within it:

1) If rule $rl'(i,j)$ precedes $rl(i,j)$, then rl can fire only after rl'. That is,

$$(fl, rl') \in Aux(i) \wedge (fl, rl') \in Aux(j).$$

If no rule precedes $rl(i,j)$, then the precondition is *true*.

2) If subflow $sfl(k,l)$ precedes $rl(i,j)$, then all instances of sfl started by fl must have ended before rl can fire. That is, assuming sfl occurs only in fl,

$$\forall k \in \mathbb{N}_N.(sfl, \_) \notin Aux(k).$$

If no subflow precedes $rl(i,j)$, then the precondition is *true*.

Thus, each rule $rl(i,j)$ in a flow $fl(i,j)$ leads to a condition $c(fl, rl)$ that is a conjunction of the two conditions above. If the rule $rl(i,j)$ appears in more than one flow, then we find $c(fl, rl)$ for each flow $fl(i,j)$ that $rl(i,j)$ appears in and the disjunction of all such conditions is defined to be the precondition $p_{rl}(i,j)$ for the firing of rule $rl(i,j)$.

We transform the preconditions $p_{rl}(i,j)$'s into noninterference lemmas as follows. Consider the rule rl and the associated precondition $p_{rl}(i,j)$. Suppose rl is of the form $rl : \forall(i,j).\rho \rightarrow a$. The noninterference lemma is simply

$$\forall i, j.\rho(i,j) \Rightarrow p_{rl}(i,j)$$

This lemma says if the rule rl is enabled for processes i and j — if $\rho(i,j)$ is true — then the precondition associated with the rule must be true as well. Note that the lemmas derived from flows involve both the variables of the protocol itself and the auxiliary variables. *Thus, they are bridging two levels: the informal, metalevel consisting of the rule names and the formal level consisting of the model under verification itself.*

As we will see in Section V, these lemmas are used exactly like any other noninterference lemma in the CMP method. If any of the lemmas is wrong then our framework, which both uses and validates the lemmas, will catch the violating lemma.

Remark 2: In practice, only protocol actions that involve the directory yield useful constraints and to optimize the abstract models we use only those in our experiments.

IV. Generalized CMP

The CMP method consists of two basic steps — *abstraction* and *strengthening* — that are applied iteratively to a protocol. The abstraction procedure used in CMP retains detailed information on a small number of processes, and abstracts away the remaining processes. Since our protocols are symmetric there is no loss in generality in focussing on processes 1 and 2. Given a symmetric protocol P with N processors $[1..N]$ and a property $I = \forall i, j \in [1..N].I(i,j)$ that we want to prove is an invariant of P, the method is as shown below. We assume that the abstraction procedure

CMP(P,I) =
  $P^{\#} = P$; $I^{\#} = I$
  while $abstract(P^{\#}) \not\models abstract(I^{\#}(1,2))$ do
    examine counterexample *cex*
    exit if *cex* is a real counterexample
    find $L = \forall i, j.L(i,j)$ ruling out *cex*
    $P^{\#} = strengthen(P^{\#}, L)$
    $I^{\#}(i,j) = I^{\#}(i,j) \wedge L(i,j)$
  end

If the loop terminates normally, the method and protocol symmetry allow us to conclude that $I^{\#}$ and consequently I are invariants of P. If the loop terminates via the exit, then either I or one of the proposed lemmas L is not an invariant of the protocol, and the user must back up and try again.

In this section, we present a new analysis of CMP that improves over the original in several ways. The main advantage of our analysis is that it allows any sound abstraction to be used as the abstraction step, and not just the data type reduction used in the original paper. In particular, in Section V, we will use message flow invariants as one of the proposed properties L in the strengthening step, but we will define a new abstraction (a variant on data type reduction) that treats in a special way the auxiliary variables used to track message flows.

A. Abstraction

An abstraction is a procedure that transforms one protocol P to another protocol $\hat{P} = abstract(P)$, and transforms one property ϕ over states of P to another property $\hat{\phi} = abstract(\phi)$ over states of $\hat{P}$. Apart from being sound, we make the additional requirement that the abstraction focuses on two processes 1 and 2 and abstracts the rest away [1]. One such well-known abstraction is *data type reduction* [19] that reduces data types with large or unbounded ranges to small, finite ranges. Given a variable v with a large range, say $[1..L]$, it can be abstracted to a variable $\hat{v}$ with a small range $\{1, 2, o\}$ which retains two values, 1 and 2, and the rest of the values are lumped into an abstract value o.

Denoting the syntactic abstraction operation in the original CMP by A_{red}, the abstract protocol $A_{red}(P)$ is constructed via a data type reduction which retains a small number of processors, say processors 1 and 2, and replaces the remaining processors $3, .., N$ with a single, highly nondeterministic process called the $Other$ process. The abstraction process is a simple, syntactic procedure: any condition in the protocol code involving processors $3, .., N$ is replaced with *true*, *false* or a nondeterministic Boolean variable as appropriate to conservatively *over-approximate* it. Any assignment to the state variables of $[3..N]$ is deleted. The abstraction $A_{red}(\phi)$ of a property ϕ is defined similarly except that any condition involving processors $3..N$ in property ϕ are replaced with *true*, *false* or a nondeterministic Boolean variable as appropriate to conservatively *under-approximate* it.

This abstraction is best explained with an example. Consider the following ruleset from the Murphi description of German's protocol. This ruleset is just a collection of guarded commands indexed by a process id $i \in$ `NODE`, where `NODE` is the parameterized range $[1..N]$.

[1]Generalization to more than 2 processes is simple.

```
ruleset i : NODE; do
  rule "RecvGntS"
    Chan2[i].Cmd = GntS
    ==>
      begin BODY endrule;
endruleset;
```

We replace this ruleset with N independent rules and apply data type reduction to each rule independently. For processors 1 and 2, the rules remain unchanged (ignoring the effect of data type reduction on the body of the rule), but for processors $i > 2$ the abstracted rule becomes

```
rule "RecvGntS"
  true
  ==>
    begin BODY' endrule;
```

This is because the condition $Chan2[i].Cmd = GntS$ refers to the state variable of an abstracted process and it is conservatively over-approximated to *true*. Thus, after applying data type reduction to the ruleset we will end up with two rulesets: the abstract rule shown above and a ruleset identical to the original ruleset except that the quantifier i ranges over $[1..2]$. The example gives a flavor of the syntactic nature of data type reduction and it is clear that the abstract rules obtained data type reduction are sound abstractions.

Theorem 1: The abstraction A_{red} is sound for $\phi(1, 2)$ for every expression $\phi(i, j)$:

$$A_{red}(P) \models A_{red}(\phi(1, 2)) \;\; \Rightarrow \;\; P \models \phi(1, 2).$$

We refer the reader to Krstic [15] for a proof.

B. Strengthening

A strengthening is a procedure that transforms one protocol P to another protocol $P^{\#} = strengthen(P, \psi)$ by replacing each guarded command $\rho \rightarrow a$ of P with the guarded command $\rho \wedge \psi \rightarrow a$ whose guard has been strengthened by ψ. Given a property ϕ, a strengthening is said to be *sound for* ϕ if it satisfies the property

$$P^{\#} \models \phi \Rightarrow P \models \phi.$$

Returning to the abstraction of the *RecvGntS* ruleset, the abstract rule is clearly too abstract. The guard *true* does not constrain the $Other$ process in any way. This leads to spurious counterexamples, and to eliminate such counterexamples, CMP depends on user-provided noninterference lemmas. Suppose we have the following lemma that we think might be useful.

```
forall p : NODE; i: NODE do
  Chan2[i].Cmd = GntS ->
    i != p -> Cache[p].State != E
end
```

Strengthening the protocol with this lemma — applying it to the *RecvGntS* rule set — and abstracting, we get the following rule set for the concrete processors

```
ruleset i : [1..2]; do
  rule "RecvGntS"
    Chan2[i].Cmd = GntS &
    forall p: NODE do
       i != p -> Cache[p].State != E
    ==>
      begin BODY endrule;
endruleset;
```

and the following rule set for the abstracted process

```
rule "RecvGntS"
  forall p: NODE do
     Cache[p].State != E
  ==>
    begin BODY' endrule;
```

Note that the abstract rule now has a meaningful guard and thus the abstract rule is more refined than previously. This process is continued iteratively by adding more and more lemmas until either a real counterexample is found or all the lemmas and the property of interest are proved.

C. Correctness

The new insight in our analysis of CMP is a generalized understanding of when a strengthening can be declared sound. Looking back at the definition of CMP, in proving $\phi(1,2)$ we are assuming $\forall i,j.\phi(i,j)$ unlike earlier compositional reasoning principles, which would assume only $\phi(1,2)$. This is explained by a property we call *entailment*.

Lemma 2: Let R_i be the states of P reachable within i steps, and let $P^{\#} = strengthen(P,\psi)$. If

$$\forall i.(\forall s \in R_i.s \models \phi) \;\; \Rightarrow \;\; (\forall s \in R_i.s \models \psi)$$

then $P^{\#} \models \phi$ implies $P \models \phi$.

Proof: The proof is similar to the proof of the guard strengthening principle in [15]. Denote by $R_i^{\#}$ the set of all states reachable in $P^{\#}$ within i steps. We will prove by induction on i that $\forall s \in R_i.s \in R_i^{\#}$ and consequently $\forall s \in R_i.s \models \phi$.

For $i = 0$, if s is an initial state of P, then it is an initial state of $P^{\#}$ as well. So the base case for induction is true.

Assume we have proved the inductive hypothesis for $i = k$. That is, $\forall s \in R_k.s \models \phi$ and $\forall s \in R_k.s \in R_k^{\#}$. We will prove that $\forall s \in R_{k+1}.s \models \phi$ and consequently $\forall s \in R_k.s \in R_k^{\#}$.

Consider any state $s' \in R_{k+1}$ reachable from $s \in R_k$ via a rule $\rho \rightarrow a$. For the rule to fire we must have $s \models \rho$. By the inductive hypothesis, $s \in R_k^{\#}$ as well. Moreover, from $\forall s \in R_k.s \models \phi$ and the condition $\forall i.(\forall s \in R_i.s \models \phi) \Rightarrow (\forall s \in R_i.s \models \psi)$ we have $\forall s \in R_k.s \models \psi$. Consequently, we have $s \models \psi$. Putting all the facts together, we have s is reachable in $P^{\#}$ within k steps and the rule $\rho \wedge \psi \rightarrow a$ is enabled at s. Therefore, s' is reachable in $P^{\#}$ within $k+1$ steps. Since $P^{\#} \models \phi$ we immediately have $s' \models \phi$. ■

We use the phrase *entailment* to refer to the condition

$$\forall i.(\forall s \in R_i.s \models \phi) \Rightarrow (\forall s \in R_i.s \models \psi)$$

It is because of this notion of entailment that our lemma for compositional reasoning given above differs subtly from the compositional reasoning principles considered by McMillan [18], Abadi and Lamport [1], [2], Krstic [15], Bhattacharya et al. [6] and Chen et al. [8]. In our case, in proving a property ϕ not only can we assume ϕ, but also the metaconsequence ψ of ϕ [2]. Moreover, ψ does not have to be discharged explicitly. Including metaconsequences of assumptions allow us to exploit domain specific knowledge such as symmetry.

Theorem 2: CMP is sound for any symmetric protocol and any sound abstraction procedure.

Proof: Since the protocol is symmetric, Lemma 1 proves the entailment precondition

$$\forall i.(\forall s \in R_i.s \models I^{\#}(1,2)) \Rightarrow (\forall s \in R_i.s \models \forall j,k.I^{\#}(j,k))$$

of Lemma 2. Thus,

$$A_{red}(P^{\#}) \models A_{red}(I^{\#}(1,2)) \Rightarrow P^{\#} \models I^{\#}(1,2)$$

by the soundness of abstraction and

$$P^{\#} \models I^{\#}(1,2) \Rightarrow P \models I^{\#}(1,2)$$

by the soundness of strengthening (Lemma 2) and

$$P \models I^{\#}(1,2) \Rightarrow P \models \forall j,k.I^{\#}(j,k)$$

by symmetry. ■

The significance of this analysis is that it shows that CMP is sound for any sound abstraction procedure not just data type reduction. The earlier analysis of CMP in [9], [15] proved

$$A_{red}(P^{\#}) \models A_{red}(I^{\#}(1,2)) \Rightarrow P \models I^{\#}(1,2)$$

with a single, complex proof that depended heavily on symmetry and the use of data type reduction as the abstraction procedure. What has happened here is to realize the soundness of strengthening depends only on entailment (the hypothesis of Lemma 2) which happens to be satisfied by symmetric protocols. Realizing this, we can prove the soundness of strengthening independent of the specific abstraction procedure being used.

Remark 3: Note that, unlike the usual counterexample guided refinement approaches, CMP requires the abstract model be sound only for $I^{\#}(1,2)$ and not for the full property $\forall i,j.I^{\#}(i,j)$. This means the abstraction can record much less information and thus scale to larger examples.

V. Parameterized Verification using Flows

We now show how to use flow-derived lemmas with CMP. We begin with the augmented protocol P_a obtained by augmenting P with auxiliary variables as described in Section IV. We then run CMP, strengthening with the flow-derived lemmas, and abstracting with a new abstraction procedure A_{red}^{a} tailored to auxiliary variables.

We augment our language of expressions to include the flow-derived lemmas as follows. We define an *augmented*

[2]For ϕ to be a logical consequence $\phi \Rightarrow \psi$ should hold. It is clear the set of metaconsequences is richer than the set of logical consequences.

expression to be a Boolean combination with quantification over free indices of the original basic expressions together with

$$(fl, rl) \in Aux(i) \qquad \forall k.(sfl, \_) \notin Aux(k)$$

To ensure sound abstraction we will require that in any augmented expression $\phi(i, j)$, variables $Aux(l), l \notin \{i, j\}$ appear only as part of a condition of the form $\forall k.(sfl, \_) \notin Aux(k)$. We augment our language of assignments in the obvious way to include the assignments to auxiliary variables used in the update procedure in Figure 3.

We note that the proof of Theorem 2 depends only on the soundness of abstraction and on the protocol being described with guarded commands. The syntactic restrictions on expressions and assignments are needed only to prove that A_{red} is a sound abstraction. Thus, we can augment expressions and assignments, and prove than an augmented abstraction procedure A^a_{red} is sound, and conclude by Theorem 2 that running CMP with this augmented abstraction A^a_{red} is sound for augmented expressions.

The abstraction A^a_{red} leaves the sets $Aux(i)$ unchanged for $i = 1, 2$ and replaces $Aux(i), i \geq 3$ with a single multiset S_{aux}, but otherwise is identical to A_{red}.

For conditions appearing in the protocol code, the operation A^a_{red} abstracts expressions involving auxiliary variables as follows:

$$(fl, rl) \in Aux(i)$$

is unchanged for $i = 1, 2$ and is abstracted to

$$(fl, rl) \in S_{aux}$$

for $i \geq 3$, and

$$\forall k.(sfl, \_) \notin Aux(k)$$

is abstracted to

$$(sfl, \_) \notin Aux(1) \wedge (sfl, \_) \notin Aux(2) \wedge (sfl, \_) \notin S_{aux}.$$

For properties, the syntactic operation A^a_{red} treats the auxiliary variables in the same as well. Finally, A^a_{red} abstracts assignments involving auxiliary variables by leaving assignments to $Aux(i)$ unchanged for $i = 1, 2$ and modifying assignments to $Aux(i)$ for $i \geq 3$ to update S_{aux} instead.[3]

Theorem 3: The abstraction A^a_{red} is sound for $\phi(1, 2)$ for every augmented expression $\phi(i, j)$:

$$A^a_{red}(P_a) \models A^a_{red}(\phi(1, 2)) \quad \Rightarrow \quad P_a \models \phi(1, 2).$$

The proof is given in the full version of this paper [25].

This theorem says we can use flow-derived lemmas with CMP as follows. First, we augment the protocol P with auxiliary variables to obtain P_a. Then we run CMP strengthening with augmented expressions and abstracting with the augmented abstraction A^a_{red}. When choosing augmented expressions for the strengthening, however, one source of augmented expressions are the flow-derived lemmas. We note that even using all the flow-derived lemmas may not be enough to cause CMP to converge. We claim, however, that the flow information is powerful enough than running CMP with the flow-derived lemmas is likely to require fewer and simpler lemmas that running CMP without them, and our experiments in the next section support this intuition.

[3]In practice, we bound the size of S_{aux} by keeping track of whether an item appears $0, 1,$ or $2+$ times. The remove operator choses between the transitions from $2+$ to $2+$ or 1 non-deterministically.

VI. Experimental Results

We applied the ideas presented in this paper to verify the standard control and data properties of the German's protocol and Flash cache coherence protocol. The control property we are interested in verifying is

$$\forall i. \forall j. i \neq j \Rightarrow (state[i] = E \Rightarrow \neg(state[j] \in \{E, S\}))$$

In words, if process i is in an exclusive state then no other process can be in exclusive or shared state.

The data properties were

$$\begin{array}{l} \forall i : (state[i] = E \Rightarrow data[i] = currdata) \\ \wedge \\ (state[i] = S \Rightarrow (collecting \Rightarrow data[i] = prevdata) \wedge \\ (!collecting \Rightarrow data[i] = currdata)) \end{array}$$

and $!dirty \Rightarrow memdata = currdata.$

The variables $currdata$ and $collecting$ are auxiliary variables introduced to specify the data properties [9]. The first property essentially says that the data value at a cache matches the last written value. The second property says the data value at the directory matches the last written value. (These are data properties for Flash; German data property was similar.) Verification of data properties is harder than control property as it involves a model with bigger state space.

We built a tool in OCaml that takes the Murphi description of a cache protocol along with the associated flows and builds an abstract model with the auxiliary variables and the invariants from flows added. (This was built on top of a pre-existing program that parsed and applied data type reduction to Murphi code.)

To verify German's protocol we used the three flows described in Table 2. The abstract model created using these flows gives us a spurious counterexamples for both the control and data property. To eliminate these we added two lemmas similar to the ones in [9]. With these lemmas added the proofs go through in about 6 seconds. In Chou et al. [9], the work closest to ours, two lemmas were needed as well though their lemmas were a little bit more complicated than ours.

The real strength of our approach is seen in the verification of Flash protocol, which is much larger than German's protocol. For this protocol, we extracted the 6 flows in a couple of hours. To get the proofs for safety properties through we had to add two lemmas on top of the flow constraints. One lemma was the same as a lemma used in [9]. The quantifier free part of our second lemma, written in CNF, has 3 clauses. The rest

of the lemmas in [9] have 12 clauses all together.[4] The running time for our abstract model is just 120 seconds compared to the couple of hours taken by the abstract model in [9]. We ran our models and the Murphi model for Flash protocol used in [9] on a 3 GHz machine with 512 MB memory. The Murphi models used in our experiments are avaiable online [25].

We believe these experimental results are significant and point to the ability of flows to give us the precise constraints needed to reason about correctness of protocols. The hardest invariants usually characterize what messages could be in transit between two processors based on the information recorder in the processors' states. Flows give us such invariants automatically which prevent the *Other* process from sending messages out of turn. This accounted for most of the lemmas used in [9]. Lemmas which related the state of local cache process to the state of the directory could not be replaced using flow constraints.

VII. Conclusion

We have demonstrated that message flows are a readily available source of powerful invariants and how these invariants can replace complicated invariants used in a typical application of the CMP method. The pragmatic implications of the applications in this paper are very encouraging. Message flows — similar to message sequence charts — are a natural, local, linear way for us to think about system behavior in place of global, monolithic, system-wide invariants. It seems likely that many systems — not just cache coherence protocols — can be understood in terms in terms of message flows. In fact, this work opens up many paths to investigate in the future.

Message flows are interesting because they are so common in industrial design documents, but message flows are not the only depictions of dependencies that appear in these documents. Timing diagrams, pipeline diagrams, block diagrams, and simple controllers with small transition systems are other common ways of describing aspects of system behavior, and dependencies described there should be just as useful as message flows when describing system constraints.

Communication events — sending and receiving messages — have a natural partial order, but so do many other system events. Reading and writing shared memory locations, acquiring and releasing a lock, starting, committing, and aborting a transaction, these are all important system events that can contribute to a flow. More abstractly, any system in which there is a clearly defined interaction among system agents should be open to a flow-based analysis and flow-inspired invariants. We have focused on message-passing systems, but it would be very interesting to extend the ideas here to concurrent systems with shared memory, transactional memory, and other systems where the notion of event is more abstract but still well-defined.

Finally, since our formulation of the CMP method separates the strengthening and abstraction phases of CMP so cleanly, we plan to investigate combining CMP with other forms of strengthening and abstraction in the future. For example, we could consider replacing data type abstraction with a form of abstraction that is less coarse and reduce the need for user supplied lemmas even further.

[4]The proof in [9] is infact incomplete; so more lemmas will be needed in their case.

References

[1] M. Abadi and L. Lamport. Composing specifications. In *ACM Transactions on Programming Languages and Systems*, 1993.

[2] M. Abadi and L. Lamport. Conjoining specifications. In *ACM Transactions on Programming Languages and Systems*, 1993.

[3] P. Abdullah, A. Buojjani, B. Jonsson, and M. Nilsson. Handling Global Conditions in Parameterized Verification. In *Proc. CAV*, 1999.

[4] T. Arons, A. Pnueli, S. Ruah, and L. Zuck. Parameterized Verification with Automatically Computed Inductive Assertions. In *Proc. CAV*, 2001.

[5] K. Baukus, Y. Lakhnech, and K. Stahl. Parameterized Verification of Cache Coherence Protocols: Safety and Liveness. In *Proc. VMCAI*, 2002.

[6] R. Bhattacharya, S. M. German, and G. Gopalakrishnan. Exploiting symmetry and transactions for partial order reduction of rule based specifications. In *SPIN*, 2006.

[7] J. Bingham. Automatic invariant generation for parameterized verification. In *Submitted to FMCAD*, 2008.

[8] X. Chen, Y. Yang, M. DeLisi, G. Gopalakrishnan, and C.-T. Chou. Hierarchical Cache Coherence Protocol Verification One Level at a Time Through Assume Guarantee. In *HLDVT*, 2007.

[9] C.-T. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *Proc. FMCAD)*, 2004.

[10] E. Clarke, M. Talupur, and H. Veith. Environment Abstraction for Parmeterized Verification. In *Proc. VMCAI*, 2006.

[11] E. Clarke, M. Talupur, and H. Veith. Proving Ptolemy Right: Environment Abstraction Principle for Parameterized Verification. In *Proc. TACAS*, 2008.

[12] A. Cohen and K. Namjoshi. Local Proofs for Global Safety Properties. In *Proc. CAV*, 2007.

[13] S. Das, D. L. Dill, and S. Park. Experience with Predicate Abstraction. In *CAV*, 1999.

[14] O. Grinchtein, M. Leucker, and N. Piterman. Inferring Network Invariants Automatically. In *Proc. IJCAR*, 2006.

[15] S. Krstic. Parameterized system verification with guard strengthening and parameter abstraction. In *Automated Verification of Infinite State Systems*, 2005.

[16] S. K. Lahiri and R. Bryant. Constructing Quantified Invariants. In *Proc. TACAS*, 2004.

[17] Y. Lv, H. Liu, and H. Pan. Computing invariants for parameter abstraction. In *MEMOCODE*, 2007.

[18] K. L. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In *Proc. Computer Aided Verification*, 1998.

[19] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME*, 1999.

[20] K. L. McMillan. Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In *Proc. Conf. on Correct Hardware Design and Verification Methods (CHARME '01)*, volume 2144 of *LNCS*, pages 179–195. Springer, 2001.

[21] K. L. McMillan. Quantified invariants using interpolants. In *TACAS*, 2008.

[22] S. Park and D. L. Dill. 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*, pages 288–296. ACM Press, 1996.

[23] A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0, 1, \infty)$ Counter Abstraction. In *Proc. CAV*, 2002.

[24] M. Talupur, S. Krstic, J. O'Leary, and M. R. Tuttle. Parameteric Verification of Industrial Strength Cache Coherence Protocols. In *Proc. Workshop on Design of Correct Circuits (DCC)*, 2008.

[25] M. Talupur and M. R. Tuttle. Going with the Flow: Parameterized Verification using Message Flows. `www.markrtuttle.com/fmcad08`.