Protocol verification using flows: An industrial experience

John O'Leary
Intel
john.w.oleary@intel.com

Murali Talupur
Intel
murali.talupur@intel.com

Mark R. Tuttle
Intel
tuttle@acm.org

***Abstract*—We prove the parameterized correctness of one of the largest cache coherence protocols being used in modern multi-core processors today. Our approach is a generalization of a method we described last year that uses data type reduction and compositional reasoning to iteratively abstract and refine the protocol and uses invariants derived from protocol "flows" to make the abstraction-refinement loop converge. Our prior work demonstrated the value of sequencing information that appeared within the linear flows describing a protocol in design documents. This paper extends the notion of flows to capture intricate scenarios seen in real industrial protocols and demonstrates that there is also valuable information in the interaction among flows. We further show that judicious use of flows is required to make the method converge and identify which flows are most suitable.**

I. Introduction

We validated an extremely complex cache coherence protocol that will soon appear in multi-core processors from Intel. We used a generalization of the method we reported last year [1] based on the CMP method [2], [3], [4] augmented with message flows. This protocol, which we call LCP, is a high-performance protocol that is designed to be scalable to a large number of cores. Such intricate distributed protocols are especially susceptible to functional bugs that standard techniques like testing and simulation are unlikely to find and consequently formal verification is indispensable in their validation. We think LCP may be one of the largest, most complicated cache coherence protocols ever validated with formal methods. As one measure, the Flash cache coherence protocol, to which only a handful of formal methods have been successfully applied, has about 10 Boolean state variables per process and 16 different message types in all. In contrast, with over 70 Boolean state variables per process and around 50 message types, the state space for LCP is several orders of magnitude larger than Flash (see Section II).

While many techniques [5], [6], [7], [8], [9] have been proposed for parametric protocol verification, none of them scale well to large protocols, and those that do scale [10], [11] require an inordinate amount of manual effort to succeed. The CMP method [1], [2], [3], [4] is the only method for parametric verification we are aware of that scales to large protocols and is easy to use. It is an interactive proof method based on compositional reasoning that uses a model checker as a proof assistant. Though it combines the best of theorem proving and model checking, the main difficulty in applying this method is coming up the *non-interference lemmas* or invariants to guide the proof. As in theorem proving, this is a time-consuming process requiring a thorough knowledge of the protocol. Moreover, adding one wrong invariant can lead the proof astray and render subsequent work useless.

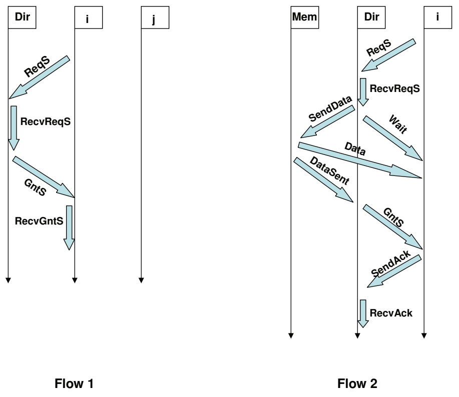

Fig. 1. Message flows as a linear sequence or acyclic graph of events.

In our earlier paper [1], we showed that the burden of generating the non-interference lemmas required by CMP can be significantly reduced by using the message *flows* typically found in industrial design documents. Flows are linear sequences of system events such as sending and receiving messages in the case of distributed message-passing systems, as illustrated on the left of Figure 1. We demonstrated the efficacy of our method by applying it to academic protocols, namely, German's protocol and the Flash protocol.

In this paper, we describe a generalization of the method presented in [1] and its application to the LCP cache coherence protocol. The primary contributions of this paper are:

1) Generalizing flows from linear traces to directed acyclic graphs, like the flow on the right of Figure 1, and a simple language for describing flows.
2) Demonstrating that we can derive powerful non-interference lemmas from constraints on events occurring in different flows, and not just constraints on events occurring within a single flow. Simply stating that two flows cannot be in progress at the same time, for example, can dramatically speed up the convergence of the CMP method.
3) Demonstrating that not all flows are equally useful, and

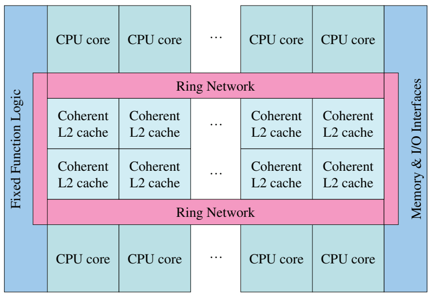

Fig. 2. Schematic of the Larrabee multi-core architecture

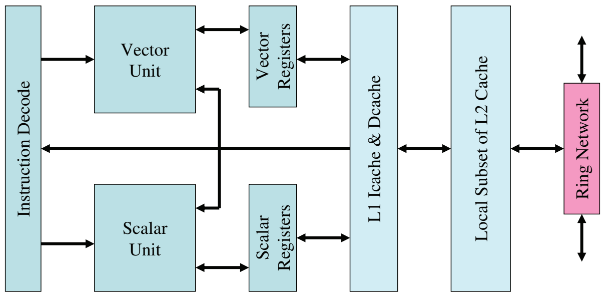

Fig. 3. Larrabee CPU core and associated system blocks

that a more judicious use of the information in flows can also speed the convergence of the CMP method.

4) Parametrically verifying the correctness of the Intel cache coherence protocol LCP for any number of processors.

In verifying LCP we used a total of 15 flows, all easily obtained from the design documents, to derive around 36 lemmas. To make the CMP method converge another 5 lemmas had to be supplied by hand. A similar effort earlier [12] where we verified a cache protocol of comparable size using the CMP method required us to supply nearly 25 lemmas manually. Clearly, flows lead to a dramatic reduction in the number manually supplied lemmas and makes it much easier to use the CMP method.

The rest of the paper is structured as follows. In the next section we describe the salient features of the LCP protocol. In Section III we discuss the possible alternatives to the CMP method and why they are inadequate. An overview of the CMP plus flows method of [1] is given in Section IV followed by a discussion of the extensions required to deal with LCP in the next section. In Section VI, we present a new language to capture richer flows and also show how to derive stronger constraints than just simple precedence constraints. A detailed description of our experience using these extended flows to verify the LCP protocol is given in Section VII. Section VIII concludes the paper.

II. Larrabee and LCP

Larrabee is the code name for a multi-core visual computing architecture under development at Intel Corporation [13]. The Larrabee architecture is based on a set of CPU cores that run the x86 instruction set, extended with support for vector processing operations and some specialized scalar instructions. Figure 2 shows a schematic of the architecture. Each core is associated with its own subset of a coherent L2 cache that affords fast, high-bandwidth data access to each core and simplifies data sharing and synchronization. The number of CPU cores and the number and type of co-processors and I/O blocks are implementation dependent, as are the positions of CPU and non-CPU blocks on the chip. To validate the LCP protocol in full generality we need parametric reasoning.

Figure 3 shows the major functional blocks in a single core. Larrabee's global second-level (L2) cache is divided into separate subsets, one for each CPU core. Each CPU has direct access to its own subset of the L2 cache. Data read by a CPU core is stored in its L2 cache subset and can be accessed quickly, in parallel with other CPUs accessing their own local L2 cache subsets. Data written by a CPU core is stored in its own L2 cache subset and is flushed from other subsets, if necessary. Larrabee uses a bi-directional ring network to allow agents such as CPU cores, L2 caches and other logic blocks to communicate with each other within the chip. The LCP (Larrabee coherence protocol) runs on the ring network and maintains coherency of shared data.

As shown in Figure 4, our model of the Larrabee coherence protocol is organized as a parameterized number of identical caching agents which talk to a central directory that controls access to the data items. For the purpose of verifying the coherence protocol our model abstracts away the ring structure and assumes point-to-point communication links between the agents (including links between the caches and from the off-chip memory to individual caches).

Unlike the Flash protocol where the directory distinguishes between local requests and external requests, the LCP makes no such distinction. This means when verifying two index properties it is enough to retain two cache agents concretely in the abstract model whereas for Flash we had to keep three agents, one local agent and two non-local agents [1]. In addition to these agents, there is also a memory controller that talks to the directory and supplies memory lines that have not yet been imported onto the chip.

The high-level model we verified preserved much of the internal structure of each caching agent. Thus, apart from the L2 cache we also had the L1 cache and actions of the agent depended on the states of both the caches. Further, the various in- and out-message buffers and related bookkeeping data structures were also modeled. Other than assuming point-to-point links between the various agents, we modeled almost every significant detail of the protocol which increased the complexity of flows/transactions considerably.

The complexity of a protocol can judged by the number of different types of messages that are exchanged by the agents. German's protocol for example has only 7 different messages, the Flash protocol, considered hard to verify, has 16 different types and LCP has around 50 different message types (comparable to the protocol we verified in [12]). In terms

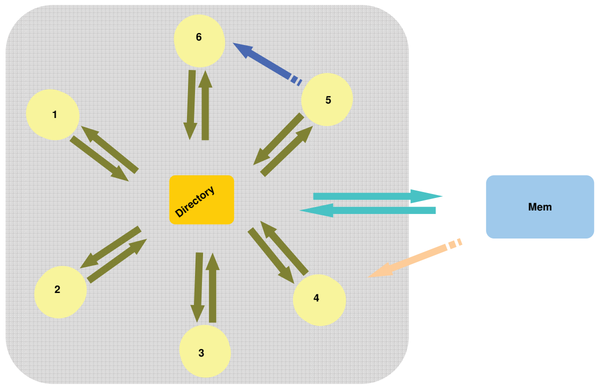

Fig. 4. Basic Organization of the LCP model

of Boolean variables, each process in LCP has approximately 70 variables. In contrast the Flash protocol has around 10 state variables per process. The Murphi description of the LCP protocol actions was about 3000 lines whereas Flash has around 1000 lines.

III. Protocol Verification Techniques

Broadly, there are two classes of techniques to verify distributed protocols: model checking methods that aim for maximum automation and theorem proving methods that aim for scalability.

A. Model Checking Methods

Several techniques like Indexed Predicates [6], [14], Counter Abstraction [5] and related methods [15], Regular Model Checking [16], [17], the Split Invariants method [18] and Environment Abstraction [19], [20] have been proposed to deal with verification of distributed protocols. While these methods have a higher degree of automation than the one we present, their scalability remains to be proven.

1) Indexed Predicates Method: Indexed predicate abstraction generalizes predicate abstraction to predicates that have free index variables. Given a protocol $P(N)$ and collection $\mathcal{I}$ of indexed predicates this method automatically produces the strongest universally quantified invariant of $P(N)$ over $\mathcal{I}$. Intricate systems like the Bakery protocol and Tomasulo's out-of-order processor have been verified by using indexed predicates [6], but scalability remains an issue. Even for as small an algorithm as German's protocol the indexed predicates method takes several hours to produce an invariant.

2) Cutoffs based approach: Another approach to verifying a parameterized system $P(N)$ is to find a cutoff k such that verifying $P(k)$ is enough to guarantee the correctness of $P(N)$ for any value of N. There has been some work on this topic [7], [21] and related topics [22], [23] but the cutoffs are large making them impractical to use. For example, in [21] a cutoff of 7 was found for a directory based protocol. But real protocols are so large that even verifying a system with 3 agents is often not possible.

3) Counter and Environment Abstractions: Counter Abstraction [5] and its generalization Environment Abstraction [19] are based on the idea of partitioning a collection of identical agents into equivalence classes based on the predicates they satisfy and for each partition tracking only one representative. The abstract models produced by these methods tend to be very detailed and consequently too large as we look at bigger protocols. Environment abstraction, for example, is just barely able to handle a simplified version of the Flash protocol [24].

B. Theorem Proving Methods

Apart from classical theorem proving there are methods like aggregated transactions [11] and disjunctive invariants [25] that are user-guided. These suffer from the well-known problem of having to provide guidance in minute detail to get the proof through. For example, the disjunctive invariants method requires the user to supply a set of additional predicates whose disjunction is inductive. It is unlikely that theorem proving style methods can be used practically to verify large protocols given that just to verify the Flash protocol the aggregated transactions method took a couple of days of effort.

C. The CMP Method

The CMP method straddles the categories of model checking and theorem proving based methods: it uses a model checker as a proof assistant to carry out user guided proof. The crucial advantage of the CMP method is that the user supplied lemmas don't have to comprise an inductive invariant [1]. This means the amount of guidance required is a lot less than in theorem proving methods. Using the CMP method we have earlier verified the Flash protocol [1] and another large cache coherence protocol within Intel [12]. The latter protocol is comparable in size to the LCP protocol. In our experience, CMP method is the only viable method currently for handling large protocols and our effort was to make it more usable by reducing the lemma burden as much as possible.

IV. Description of the CMP method

For the rest of the paper we will use the same system model as in [1]. In particular, we consider a symmetric protocol P with N processors $[1..N]$ whose transition relation is given as a collection of rules. Each 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*, a is a list of assignments called the the *action* and i, j are the process index variables.

The CMP method is a compositional reasoning based method and it consists of two basic steps — *abstraction* and *strengthening* — that are applied iteratively to a protocol as shown in Figure 5.

Given a property $I = \forall i, j \in [1..N].I(i, j)$ that we want to prove is an invariant of P, the CMP method creates an

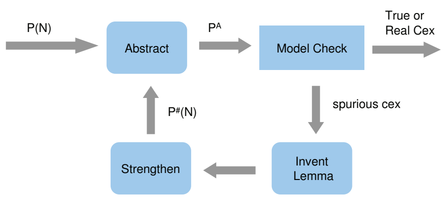

Fig. 5. The CMP method

abstract model $\widehat{P}$ that retains two agents, say $1, 2$ without loss of generality, and replaces the rest of the processes with a highly non-deterministic process $Other$. Intuitively, in a symmetric system if a property $\phi(1, 2)$ holds for processes $1, 2$ then $\phi(i, j)$ holds for any other pair of processes i, j as well. Thus, it is enough to consider an abstract model with detailed information on $1, 2$ and check if $I(1, 2)$ holds. The abstraction process in CMP method is inexpensive as it is almost syntactic and the bulk of the state space of $\widehat{P}$ comes from processes 1 and 2. On the flip side $\widehat{P}$ tends to be very coarse as the behavior of $Other$ is completely unconstrained. To get rid of the resulting spurious counterexamples, the CMP method requires the user to provide *non-interference lemmas* or invariants that are used to refine the abstract model. This process is continued iteratively till we find a real counterexample or prove that $I(1, 2)$ holds. Pseudo-code for the method is given below.

```
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 McMillan's work [2] the abstraction operation used was data type reduction [26] which essentially throws away all the state variables of processes $[3..N]$. Our analysis of the CMP method in [1] allows us to use richer abstractions than data type reduction. In particular our method allows meaningful abstraction of the auxiliary variables used to track flows.

A. *Making* CMP *better using Flows*

Not surprisingly the main difficulty in applying the CMP method is coming up with the non-interference lemmas. As demonstrated in [1], message flows or simply flows yield powerful invariants that can be used as non-interference lemmas in the CMP method and reduce the number of lemmas we need to supply by hand.

The flows used in [1] are linear orderings of events usually involving two agents, see Figure 1. Each entry in a flow is either a simple event, corresponding to a single rule firing in the protocol, or a *sub-flow* where a sub-flow is itself a flow composed of simple events. The notion of sub-flow serves to chop up a complicated flow into smaller units such that each unit shows interaction between two agents.

The constraints derived from flows are the implicit precedence constraints between events occurring in the flows. For example, according to the first flow in Figure 1, $ReqS$ has to happen before $RecvReqS$ can happen. This can be converted into a precise lemma by having a set $Aux(i)$ of auxiliary variables for each process i that track all the flows i is involved in and for each flow the last rule that was fired by i. In case of $RecvReqS$ action the precedence constraint is simply that if, for process i, the $RecvReqS$ action is enabled then there must be an auxiliary variable recording the fact that i was involved in $ReqS$ action earlier. These simple precedence constraints turn out to be surprisingly powerful as invariants.

The advantage of flows is that they are intuitive to understand and readily available in the design documents. Flows in fact allow us to state the core ideas that go into the design of a protocol in terms of higher level concepts while avoiding the specific implementation details. This means flows are quite robust and resistant to changes in the protocol design which makes them very attractive as user supplied annotations.

V. Extensions to Flows

Apart from the local caching agents and the directory, real cache protocols have other types of agents, like the off-die memory controller, which add to the complexity of the interaction between the agents. Flows designed to capture two agent interactions are no longer sufficient.

Consider Flow 2 of Figure 1 which calls for an extension to our notion of flows. Here a process i requests access to an item that has not yet been fetched onto the chip. The on-chip directory forwards the request to the off-chip memory controller along with the id of the requesting agent. The directory also sends an acknowledgment to the caching agent. The memory controller for its part sends the required item to the caching agent and also sends a message completion to the directory. On receiving the message completion the directory sends a grant message to the agent. On receiving both the grant message from directory and the data message from the memory controller agent i transitions to shared state and sends a completion message to the directory. The transaction ends when the directory receives the completion message.

This scenario is similar to, and typical of, the complex interactions present in LCP and it differs from the flow shown in the left of Figure 1 in crucial ways. The interaction between the three agents is tightly coupled and it is not possible to identify meaningful sub-flows that have only two agents involved. In [1], even though we had more than two agents involved, it was to easy see that each flow was made up of logical sub-units consisting of only two agents.

Further, an event might have multiple preceding actions. For instance, event *SendAck* can happen only after the agent has received both the data and grant messages. Receiving only one of these is not sufficient to enable the *SendAck* message. Linear flows cannot capture multiple dependency.

Similarly, an event in the flow might have more than one succeeding event. For instance, the $RecvReqS$ event leads to two further events $SendData$ and $Wait$. With linear flows the number of events depending on a given event is at most one.

Finally, unlike Flow 1 of Figure 1 which totally orders its events, Flow 2 imposes only a partial order. For instance, there is no ordering between the events $Data$ and $GntS$; they can be received by i in any order. One way to deal with this is to flatten the partial order into a collection of total orders. But this runs into two issues: first, the number of resulting flows might be large and unnecessarily obscure the simplicity present in the dag representation. Second, having more flows also means we will have to introduce more auxiliary variables or extend their ranges and thus, we will also end up making the augmented model bigger.

Thus, it is clear that a new notion of flows is needed that is expressive enough to capture directed acyclic graphs (dags). Moreover, note that in Flow 2 of Figure 1 there are three primary agents in the flow but that each event involves at most two agents. So apart from specifying events we also have to specify which agents are involved in the events. A new flow language that takes into account these extensions is described in the next section.

VI. Generalized Flows

To keep the presentation simple we omit treatment of subflows in this section since they are not required for LCP.

A. Language for Flows

A flow is denoted by

$$(\mathit{flow}, \mathit{conflicts}) : \{prec_1, \ldots, prec_n\}$$

where *flow* is the name of the flow, *conflicts* is a set of flows, and each $prec_i$ is a *precedence* of the form

$$event : \{event_1, \ldots, (event_m\}$$

where each $event$ is a triple $(rule, id, agents)$ specifying a particular firing of rule $rule$ by processes in the list $agents$. The meaning of a precedence is that each rule $rule_i$ instantiated with the list of agents $agents_i$ must occur in an execution before the rule $rule$ instantiated with the list of agents $agents$ can occur. We say each $(rule_i, id_i, agents_i)$ *precedes* $(rule, id, agents)$ in the flow. For example, in the flow on the right side of Figure 1, the first precedence is

$$(ReqS, id_1, \langle Dir, i\rangle) : \{\}$$

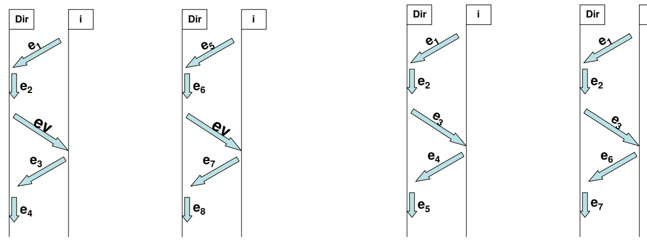

Fig. 6. Flows can share a common event (left) and a common prefix (right).

and the third precedence is

$$(SendData, id_2, \langle Mem, Dir\rangle) : \{(RecvReqS, id_3, \langle Dir\rangle)\}$$

The meaning of a conflict set is that a flow f and a flow f' in the conflict set of f cannot be alive at the same time. A flow is said to be *alive* if some rule in the flow has occurred and one of its successors has not occurred. The conflict set of f might contain f itself. In this case the meaning is that there can be only one instantiation of f alive in the system. The constraints resulting from these conflict sets turn out to be very powerful in constraining the $Other$ process.

Finally, we associate an id with each event since a particular instantiation of a rule may occur in multiple flows. The left side of Figure 6 illustrates that we must distinguish the occurrences of ev in the flows f_1 and f_2, so that the occurrence of ev in f_1 will enable the event e_3 in f_1 and not e_7 in f_2. The right side of Figure 6 illustrates that we cannot use the flow name as the event id, since different flows can share a common prefix. As long as the system is in the prefix, we don't know whether the system is in flow f_1 or f_2. In fact, because we don't know which flow the system is in, an event in the prefix must have the same event id in both flows.

B. Tracking Flows

We use auxiliary variables to track active flows as in [1]. We will assume that a pair (rl, id) appears only once in a flow.

Define $last(rl, id, p)$ in a flow f to be the set of pairs (rl', id') for which there exist agent lists ag and ag' both containing p such that (rl', id', ag') precedes (rl, id, ag) in f. We require that $last(rl, id, p)$ be the same for all flows containing (rl, id). Intuitively, this means the prefix for (rl, id) must be the same in all flows in which it appears.

Define $next(rl, id, p)$ in a flow f to be the set of pairs (rl', id') for which there exist agent lists ag and ag' both containing p such that (rl, id, ag) precedes (rl', id', ag') in f.

Define the *out degree* of (rl, id) for p in f to be the size of $next(rl, id, p)$ in f. We require that the out degree of (rl, id) for p be the same in all flows containing (rl, id).

Define $\mathit{flows}(rl, id)$ to be the set of flows f containing an event (rl, id, ag) for some agent list ag. This is the set of all flows containing (rl, id) and by definition one of them is active when an agent executes (rl, id).

For each process p, we maintain a set $Aux(p)$ of tuples of the form (rl, id, od, fls) where (rl, id) is an instance of

```
$update(p, rl(i,j), id)$ =
  for each $(rl', id') \in last(rl, id, p)$
    $aux$ = choose $(rl', id', od, fl\_set) \in Aux(p)$
      such that $flows(rl, id) \cap fl\_set \neq \{\}$
    $new\_aux$ = if $od > 1$
      then $\{(rl', id', od-1, fl\_set \cap flows(rl, id))\}$
      else $\{\}$;
    $Aux(p) := Aux(p) \setminus \{aux\} \cup new\_aux$
  if $(rl, id)$ precedes no other rule then
    return $Aux(p)$ else
    return $Aux(p) \cup (rl, id, outdegree(rl, id), flows(rl, id))$
```

Fig. 7. Tracking flows with auxiliary variables. This procedure describes how p updates $Aux(p)$ when an event (rl, id, ag) involving p is executed. The choose operator throws an error if there is nothing to choose.

a rule rl that has fired with an instantiation including p, od is the number of successors of (rl, id) in a flow that have yet to fire,[1] and fls is the set of candidates for the flow — containing (rl, id) — currently in progress. When (rl, id) fires, we add this tuple to the set with od initialized to the out degree of (rl, id) for p and fls to the set $flows(rl, id)$ of all flows containing (rl, id). As each successor (rl', id') to (rl, id) in a flow fires, we decrement od by 1 and restrict the set fls to those flows containing (rl', id') in addition to (rl, id). When od drops to 0, we remove the tuple from the set. This procedure is given by the *update* procedure in Figure 7.

One unanswered question about the *update* procedure is how we choose the value of id when an instantiation $rl(p_1, p_2)$ of a rule $rl(i,j)$ fires.[2] The answer is that we try every value for id such that (rl, id) appears in a flow. Most values of id will fail because the choose operator will fail to find a satisfactory tuple in the auxiliary set of tuples and throw an exception. We use any value of id that succeeds (and undo the effects of prior attempts that failed). For LCP, corresponding to each rule rl there was only one id that appeared in the flows, so updating auxiliary variables was simpler than the general procedure given here.

C. Lemmas from Flows

We derive two classes of lemmas from flows. The first class is an extension of the "precedence" lemmas derived in [1] to the richer flows. The second class uses the conflict sets.

1) Precedence Constraints: We define the precondition for firing the rule rl with the processes i and j by

$$pre_{i,j}(rl) = \bigvee_{id} pre_{i,j}(rl, id).$$

We define the precondition for firing (rl, id) with i and j by

$$pre_{i,j}(rl, id) = pre_i(rl, id) \wedge pre_j(rl, id).$$

We define the precondition for firing (rl, id) with i by

$$pre_i(rl, id) = \bigwedge_{(rl', id') \in last(rl, id, i)} pre_i(rl, id)(rl', id')$$

[1]Alternatively, we could maintain the actual set of successors yet to fire.
[2]A rule involving more or less than two agents is handled similarly.

where $last(rl, id, i)$ is the set of instances (rl', id') involving i that precede (rl, id) in a flow, and similarly for j. We define the precondition for firing (rl, id) with i given the prior firing of (rl', id') by

$$\begin{aligned} pre_i(rl, id)(rl', id') = & \\ \exists\, od', fls'.\quad & (rl', id', od', fls') \in Aux(i) \\ & \wedge od' > 0 \\ & \wedge fls' \cap flows(rl, id) \neq \{\} \end{aligned}$$

If the guard for the rule $rl(i,j)$ is $\rho(i,j)$, then the precedence non-interference lemma is

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

2) Conflict Constraints: If two flows f^1 and f^2 conflict, then both cannot be active at the same time, meaning that while f^1 is active f^2 cannot start. Let

$$(rl_1^1, id_1^1, \_) \ldots (rl_m^1, id_m^1, \_)$$

be the set of events appearing in f^1 and

$$(rl_1^2, id_1^2, \_) \ldots (rl_n^2, id_n^2, \_)$$

be set of *initial* events appearing in f^2, meaning that these events have no preceding events in f^2. If f^1 is active, then some process p must have fired some rule rl_k^1 in f^1, so some tuple of the form $(rl_k^1, id_k^1, \_, \_)$ must appear in $Aux(p)$. The flow f^2 cannot start if for every rule rl_ℓ^2 starting f^2 and for every pair of processes i and j that might fire rl_ℓ^2, the guard $\rho_\ell^2(i,j)$ of $rl_\ell^2(i,j)$ is false. Our first conflict non-interference lemma is

$$\begin{aligned} \exists p\ \exists (rl_k^1, id_k^1, \_).\ & (rl_k^1, id_k^1, \_, \_) \in Aux(p) \\ & \Rightarrow \forall (rl_\ell^2, id_\ell^2, \_)\ \forall i, j.\ \neg \rho_\ell^2(i,j) \end{aligned}$$

where $\rho_\ell^2(i,j)$ is the guard of the rule $rl_\ell^2(i,j)$.

If the flow f^1 is in conflict with itself, then a process firing a rule in the flow cannot fire that rule again until the flow has ended, since the flow conflicts with itself. Our second conflict non-interference lemma is

$$\exists (rl_k^1, id_k^1, \_, \_) \in Aux(i) \Rightarrow \forall j.\ \neg \rho_k^1(i,j).$$

The two conflict lemmas are similar, but the first prevents firing of initial rules and the second prevents refiring of any rule. The second lemma turned out to be surprisingly helpful in restricting the $Other$ process in the abstract model.

VII. Verifying the LCP Protocol

In this section we describe our experience applying the CMP method in conjunction with flow based lemmas to verify the LCP protocol.

A. *Obtaining the Flows*

The flows we considered during verification were all readily available in the design documents written by the architects. In fact, the scenarios listed in the design documents had more information than we needed. Intuitively, only those parts of the flows that involve the directory (in other words the place where all the coordination happens) yield useful invariants.

Apart from identifying the useful parts of the flows, we had to annotate the events with the agents involved and also identify the conflict set for each flow. Both these steps were straightforward.

B. *Protocol Model*

The Murphi model that we verified was quite hierarchical with each rule having an extremely large body covering a variety of cases (as it was semi-automatically generated from tables describing the protocol). For instance, there is only one rule specifying the behavior of a cache agent in case it receives an invalidate message. The guard of the rule only checks the type of the incoming message and the body of the rule considers the various sub-cases depending on the state of the L1/L2 caches and the data structures. The behavior of the cache in each of the sub-cases might be quite different with differing messages being sent out.

In other words, the events associated with each of the sub-cases are quite different though they all belong to the same Murphi rule syntactically. This is not conducive to our flow based methodology which depends on being able to track events precisely. To allow precise tracking of flows, we broke up large rules into smaller ones so that each rule corresponded to a specific event mentioned in the flows. This was accomplished by simply lifting some of the branch conditions in the body of a rule to the guard.[3] After this step each event mentioned in the flows mapped to a Murphi rule.

C. *Proof details*

The properties that we proved were the standard safety property requiring that if there is a cache with exclusive access to a data item then no other cache has access to that item and properties specifying consistency between the directory's list of caches with access to a data item and the real access each local cache has.

To carry out the abstraction and refinement with lemmas we used a tool called *Abster* written in Ocaml. Abster takes in a parameterized protocol written in Murphi and creates an intermediate abstract syntax tree (AST) for the protocol. All the abstraction and refinement operations take place on the AST. We specify the number of agents, usually 2 or 3, to be retained concretely in the abstract model depending on the protocol to be verified. The flows are specified in a separate file. Abster automatically constructs flow lemmas and adds them to the guards of the appropriate rules (a rule whose guard appears in the flow invariant gets modified by that invariant).

[3]This had to be done only for some of the rules. Similarly, not all branch conditions had to be lifted.

To carry out the proof we used 15 flows from design documents. These led to 36 flow lemmas, with 25 of them coming from precedence constraints and the rest from conflict constraints. At first we used only the precedence constraints and proceeded to refine the abstract model with hand supplied lemmas. But we soon realized that several counterexamples were caused by two conflicting flows starting at the same time. More precisely, suppose one of the concrete agents, say 1, is involved in a *Request Shared* transaction. Since the $Other$ process is not fully constrained, it might start sending conflicting messages corresponding to *Request Exclusive* transaction to the directory. One way to prevent this it to write lemmas by hand and refine the $Other$ process. The simpler option is to use the conflict constraints that prevent a *Request Exclusive* transaction from starting while a *Request Shared* transaction is alive. Together with precedence constraints this ensures that no rule in a conflicting flow can fire.

Another cause of spurious counterexamples was the $Other$ process repeatedly firing the same rule from a flow. Suppose (rl', id') precedes (rl, id) and some other event. That is, the out degree of (rl', id') is 2. Since we are only tracking the number of successors of (rl', id') and not the precise names, (rl, id) can get fired twice by the $Other$ process. This does not happen for the concrete agents because they have state variables controlling their execution which is not the case with $Other$ process. Adding conflict constraints fixes this problem as well.

To complete the proof it was necessary to add 5 lemmas by hand. Since we had in- and out-message buffers, characterizing when a cache had access to a given item was hard. A grant shared message might be sitting in the in-message buffer, for instance, though the L2 state may not reflect it. We used the CMP method (without the flow invariants) earlier to verify a protocol of similar size though less intricate because of simpler internal structure of each cache [12]. There we had to add 25 lemmas by hand. Compared to that effort the reduction obtained using flows is dramatic and clearly makes the CMP method much more usable. This experience once again confirms that flows do yield powerful invariants that get to the heart of protocol correctness. The running time for the final abstract model was around 5.5 hours.

D. *Flows and State Explosion*

A surprising discovery during this proof process was that, even after we have chosen important flows involving the directory, using all the flows is not the right thing to do as it leads to state explosion in the abstract model. Apart from various flows for requesting shared and exclusive accesses we had a collection of flows for write backs and invalidates. Unlike the former flows the latter flows were not incompatible with themselves, that is, there could well be many of these flows alive at the same time. Thus, in the abstract model, the $Other$ process can fire rules from these flows multiple times and saturate the auxiliary variables leading to an explosion in the number of states. It took us some time to understand this phenomenon, especially since augmenting the concrete model with auxiliary variables increased the state space by at most a factor of 2. That is, the auxiliary variables don't increase the number of states in the concrete model by much, they only widen the state. But this is not so for the abstract model, especially if we have flows that don't have too many conflict constraints. This experience indicates that flows that appear in their own conflict sets might be best ones to use.

E. Flows as Validation Collateral

Apart from helping us prove the safety properties of interest by yielding invariants, the flows themselves are valuable validation collateral. The CMP method not only uses lemmas but also validates them in the process. Thus, we are not only using flows but also proving them correct. To see the crucial flows of the protocol exercised and to have an assurance that important actions of the protocol, like the directory sending grant exclusive/shared access messages, happen precisely as specified by flows, constitutes a much stronger validation of the protocol than just verifying a final global property. In fact, the architects who saw our proofs were impressed more with the fact that we validated the flows than they were with the global safety properties that we verified!

VIII. Conclusion

Finding invariants is one of the central problems of formal verification and extensive research is being carried out to find invariants automatically or via user provided annotations. While the flow based technique falls into the latter category, it has the advantage that flows arise naturally and are readily available. Ideally, annotations (or user supplied information) should be 1) easy to find, 2) provide relevant information precisely and in an easy to understand manner and 3) stay stable as the design details change. Flows have all three properties which makes them so attractive to use.

Apart from message passing distributed systems handled here, flows or partial orders on system events can also be applied to other types of distributed system. Lamport [27] has used partial orders analogous to flows in reasoning about mutual exclusion algorithms. In [27] the partial order is defined over *operations* which consist of sequences of atomic events. In addition to the precedence relation over events used in this paper, Lamport also defines a *can influence* relation over operations. While we used flows to derive invariants, Lamport (manually) reasons directly in terms of the partial order to prove the mutual exclusion property (which is also defined in terms of the precedence relation). A natural extension to our work is to generalize the notion of flows along the lines suggested by [27] to address other types of distributed systems.

In this paper we used abstraction to verify properties of a given model. An important related problem is bridging the abstraction gap between two existing models, such as the logical specification of a coherence protocol and its RTL implementation. Chen and others [28], [29] have developed techniques that address this problem and another natural extension to our work is to investigate the applicability of these techniques to our protocols.

Acknowledgments We thank Ching-Tsun Chou for building and validating the original LCP Murphi model that we validated parametrically in this paper.

References

[1] M. Talupur and M. R. Tuttle, "Going with the flow: Parameterized verification using message flows," in *Proc. FMCAD*, 2008.
[2] K. L. McMillan, "Parameterized verification of the FLASH cache coherence protocol by compositional model checking," in *Proc. CHARME*, 2001.
[3] C.-T. Chou, P. K. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," in *Proc. FMCAD*, 2004.
[4] S. Krstic, "Parameterized system verification with guard strengthening and parameter abstraction," in *Automated Verification of Infinite State Systems*, 2005.
[5] A. Pnueli, J. Xu, and L. Zuck, "Liveness with $(0, 1, \infty)$ counter abstraction," in *Proc. CAV*, 2002.
[6] S. K. Lahiri and R. Bryant, "Constructing quantified invariants," in *Proc. TACAS*, 2004.
[7] A. E. Emerson and V. Kahlon, "Reducing model checking of the many to the few," in *Proc. CADE*, 2000.
[8] J. Bingham, "Automatic non-interference lemmas for parameterized model checking," in *Proc. FMCAD*, 2008.
[9] Y. Lv, H. Liu, and H. Pan, "Computing invariants for parameter abstraction," in *Proc. MEMOCODE*, 2007.
[10] S. Das, D. L. Dill, and S. Park, "Experience with predicate abstraction," in *Proc. CAV*, 1999.
[11] S. Park and D. L. Dill, "Verification of flash cache coherence protocol by aggregation of distributed transactions," in *Proc. SPAA*, 1996.
[12] M. Talupur, S. Krstic, J. O'Leary, and M. R. Tuttle, "Parameteric verification of industrial strength cache coherence protocols," in *Proc. Workshop on Designing Correct Circuits*, 2008.
[13] L. Seiler *et al.*, "Larrabee: A many-core x86 architecture for visual computing," *ACM Transactions on Graphics*, vol. 27(3), Aug. 2008.
[14] S. K. Lahiri and R. Bryant, "Indexed predicate discovery for unbounded system verification," in *Proc. CAV*, 2004.
[15] G. Delzanno, "Automated verification of cache coherence protocols," in *Proc. CAV*, 2000.
[16] P. Abdullah, A. Buojjani, B. Jonsson, and M. Nilsson, "Handling global conditions in parameterized verification," in *Proc. CAV*, 1999.
[17] P. Abdullah and B. Jonsson, "On the existence of network invariants for verifying parameterized systems," in *Correct Systems Design-Recent Insights and Advances*, 1999.
[18] A. Cohen and K. Namjoshi, "Local proofs for global safety properties," in *Proc. CAV*, 2007.
[19] E. Clarke, M. Talupur, and H. Veith, "Environment abstraction for parmeterized verification," in *Proc. VMCAI*, 2006.
[20] ——, "Proving Ptolemy right: Environment abstraction principle for parameterized verification," in *Proc. TACAS*, 2008.
[21] A. E. Emerson and V. Kahlon, "Model checking large-scale and parameterized resource allocation systems," in *Proc. TACAS*, 2002.
[22] E. A. Emerson and K. S. Namjoshi, "Reasoning about rings," in *Proc. POPL*, 1995.
[23] E. Clarke, M. Talupur, T. Touilli, and H. Veith, "Verification by network decomposition," in *Proc. CONCUR*, 2004.
[24] M. Talupur, "Abstraction Techniques for Infinite State Verification," Ph.D. dissertation, SCS, CMU, 2006.
[25] J. Rushby, "Verification diagrams revisited: Disjunctive invariants for easy verification," in *Proc. CAV*, 2000.
[26] K. L. McMillan, "Verification of infinite state systems by compositional model checking," in *Proc. CHARME*, 1999.
[27] L. Lamport, "A new approach to proving correctness of multiprocess programs," *ACM TOPLAS*, vol. 1(1), pp. 84–97, 1979.
[28] X. Chen, S. German, and G. Gopalakrishnan, "Transaction based modeling and verification of hardware protocols," in *Proc. FMCAD*, 2007.
[29] 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 *Proc. HLDVT*, 2007.