An Automatical Parameterized Verification of FLASH Cache Coherence Protocols by Paraverifier

Abstract. The FALSH protocol is an industrial-scale cache coherence protocol, whose parameterized verification is a notoriously hard challenge to the field of formal methods. In this paper, we show how to verify its important properties by our tool paraverifier. Being distinguished from any other approach, our proof product are a formal proof with a set of inductive invariants. Among all the work, our work is the most automatical, and the the set of auxiliary invariants is the most complete. Both invariants are searched automatically and the formal proof is generated automatically. Besides, we make efforts to illustrate the semantical intuition behinds these invariants.

1 Introduction

Verification of parameterized concurrent systems is interesting in the area of formal methods, mainly due to the practical importance of such systems. Parameterized systems exist in many important application areas: cache coherence protocols, security systems, and network communication protocols, *etc.* In this work, we will focus on a cache coherence protocol -FLASH. The challenge posed by parameterized verification is that the desired properties should hold in any instance of the parameterized system.

Stanford FLASH cache coherence protocol [1] is a publicly-recognized challenging benchmark in the field of formal verification. In the pioneering work, Park and Dill applied the general purpose theorem prover PVS[2] to verifing the protocol for arbitrary N nodes[3]. This is a laborious process, since they introduce a simplified FLASH protocol with the so-called aggregated transactions, which is in fact an abstracted version of FLASH, and need prove the correspondence between the abstract and the original FLASH protocol, and then prove the correctness of the abstracted protocol, and subsequently derive the correctness of the original protocol by the correspondence. New auxiliary state variables like `fwdSrc` are introduced for verification. Deep human insight for FLASH is needed for both the construction of the abstracted protocol and introducing new state variables. Inductive invariants must also be provided by human, and the theorem prover must be manually guided to perform the induction proof to prove the correspondence between the aggregated protocol and the original one.

Mcmillan applies methods of compositional model checking [4] to the verification of FLASH by using Candence-SMV [5]. Safety and liveness of the FLASH protocol are both verified. Despite the fact SMV is designed as a model-checker to automatically checking properties, the core techniques for FLASH parameterized verification adopted by Mcmillan is SMV's advanced features for proof such as composition proof and temporal case splitting, and abstraction. Human interaction must be heavily relied on to guide SMV to work, depending on his deep insight to FLASH. Unlike a general theorem prover like Isabelle, SMV does not have a good logical foundation and mechanism to perform theorem proving. The above proofs are neither rigorous as a theorem prover, nor easily understood to a non-specialist of SMV. Because these proof techniques is only special for SMV, they are difficult to be generalized to other protocols and to be implemented in a general-purpose theorem prover.

The CMP method, which adopts parameter abstraction and guard strengthening, is proposed in [6] for verifying a safety property inv of a parameterized system. An abstract instance of the parameterized protocol, which consists of m + 1 nodes $\{P_1, \ldots, P_m, P^*\}$ with m normal nodes plus one abstract node P^*, is constructed iteratively. The abstract system is an abstraction for any protocol instance whose size is greater than m. Normally the initial abstract system does not satisfy the invariant inv. Nevertheless it is still submitted to a model checker for verification. When a counterexample is produced, people need to carefully analyze it and come up with an auxiliary invariant inv', then use it to strengthen the guards of some transition rules of the abstract node. The 'strengthened' system is then subject to model checking again. This process stops until the refined abstract system eventually satisfies the original invariant as well as all the auxiliary invariants supplied by people. However, this method's soundness is only argued in an informal way. To the best of our knowledge, no one has formally proved its correctness in a theorem prover. This situation may be not ideal because its application domain for cache coherence protocols which demands the highest assurance for correctness. Besides, the analysis of counter-example and generation of new auxiliary invariants usually depend on human's deep insightful understanding of the protocol. It is too laborious for people to do these analysis and some effective automatic tool is needed to help people.

In [7], an algorithm, called BRAB, computes over-approximations of backward reachable states that are checked to be unreachable in a finite instance of the system. These approximations (candidate invariants) are then model checked in together with the original safety properties. A Finite instance (even small) is regarded as an oracle for guiding the choice of candidate invariants. Auxiliary invariants are found automatically, but these auxiliary invariants are in concrete form, and are not generalized to the parameterized form. Thus there is no parameterized proof derived for parameterized verification. Until now, we still can not find a completely formal proof is constructed to verify the full-version of FLAH protocol by adopting the invariants computed by BRAB.

To sum up, FLASH is a hard benchmark with significance for any proposed method for parameterized verification. First it is a cache coherence protocol

in real-world, which is the most important landmark in this field. As Chou, Mannava, Park pointed out in their FMCAD 2004 paper [6], if the method works on FLASH, then there is a good chance that it will also work on many real-world cache coherence protocols. Second FLASH protocol is sufficiently hard that only two or three methods have fully verify the protocol parametrically. However, human guidance still plays a key role in these successful verification for FLASH. This fact reveals the weakness of automatical tool in the parameterized verification. Third, further efforts are still needed for clear mechanization. As argued in [6], "the first priority is clearly mechanization. Ideally, we want to formalize not only the reasoning steps but also the theory developed in a theorem prover, so that we can have a completely formal proof". Therefore it is preferable to have a completely formal proof for verification of FLASH in a well-known theorem prover. Previous work is too far away from giving a formal proof for FLASH. Even for a moderate case GERMAN protocol, which is much simpler than FLASH, no formal proof is not available yet. Let alone FLASH.

The aim of this paper is to apply paraVerifier to a parameterized verification of FLASH protocol in both an automatical and rigorous way. In detail,

- Interesting auxiliary invariants can be found by paraVerifier automatically. Our invariants are visible, which can characterize the semantical features of FLASH, and help people to precisely understand the design of FLASH.
- With the help of paraVerifier, a formal proof can be constructed automatically as a product of a parameterized verification of FLASH. The formal proof script not only models the protocol rigorously and specifies its properties without any ambiguity, but also proves them mechanically in the theorem prover. Therefore, it helps us to achieve the highest possible assurance for the correctness of FLASH.

The organization of this work is as follows: Section 2 introduces the background of paraVerifier; Section 3 introduces informally FLASH protocol; Section 4 introduces our FLASH model in Murphi[**?**]; Section 5 shows our verification experiments in detail. Especially we introduce some advanced techniques used in our experiments: oracles of simplified version and hybrid oracles, and distributing oracles. Section 6 tries to explain meanings of inductive invariants with flows. Section 7 concludes our work.

2 The Background of paraVerifier

paraVerifier is devoted to the parameterized verification of a protocol. Usually the tool is used to prove the correctness of the protocol after model checking or testing techniques have been used to verify that typical protocol instances of the protocol is bug-free. There is still a big gap between the bug-freeness in quite a small number of instances between the correctness in all instances. Ideally, a proof is preferable to be given to prove that the correctness holds for any instance. It is a formal proof in a theorem prover that paraVerifier generates to verify the protocol. In order to achieve this aim, paraVerifier is designed with the following features.

Theoretical foundation paraVerifier is based on a simple but elegant theory. Three kinds of causal relations among a rule and an invariant and a set of invariants are introduced, which are essentially special cases of the general induction rule. Then, a so-called consistency lemma is proposed, which is the cornerstone in our method. Especially, the theory foundation itself is verified as a formal theory in Isabelle, which is the formal library for verifying protocol case studies. The library provides basic types and constant definitions to model protocol cases and lemmas to prove properties. Therefore, the theoretical foundation itself is verified in Isabelle. Thus it is the most rigorous.

Tool paraVerifier is composed of two parts: an invariant finder `invFinder` and a proof generator `proofGen`. An overview of our method is illustrated in Fig. 2.

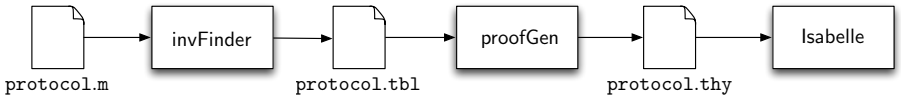

Fig. 1. The workflow of paraVerifier

invFinder Given a protocol $\mathcal{P}$ and a property *inv*, written in a Murphi file `prot.m`, is compiled into an internal form prot.ml, then fed into the `invFinder`. `invFinder` automatically transform the Murphi file into internal formal model in Ocaml, then tries to find useful auxiliary invariants and causal relations which are capable of proving *inv*. To construct auxiliary invariants and causal relations, we employ heuristics inspired by consistency relation. Also, when several candidate invariants are obtained using the heuristics, we use oracles such as NuSMV and Murphi model checker and an SMT-solver to check each of them under a small reference model of $\mathcal{P}$, and chooses the one that has been verified in the oracles. A table `prot.tbl` is worked out to store the set of ground invariants and causal relations.

proofGen For the auxiliary invariants and causal relations stored in table `prot.tbl`, `proofGen` generalizes them into a parameterized form, which are then used to construct a completely parameterized formal proof in a theorem prover (e.g., Isabelle) to model $\mathcal{P}$ and to prove the property *inv*. After the base theory is imported, the generated proof is checked automatically. Usually, a proof is done interactively. Special efforts in the design of the proof generation are made in order to make the proof checking automatically.

3 Informal Account of FLASH Protocol

3.1 A textual description

Firstly we give a classical textual description, which is directly cited from the material in [**?**] to informally introduce FLASH protocol. FLASH protocols is

a directory-based one which supports a large number of distributed processing nodes. Each cache line-sized block in memory is associated with directory header which keeps information about the line. Each memory address has a home node. Each cache line-sized block in memory is associated with a directory header which keeps information about the line. For a memory line, the node on which that piece of memory is physically located is called the home; the other nodes are called remote. The home maintains all the information about memory lines in its main memory in the corresponding directory headers. The system consists of a set of nodes, each of which contains a processor, caches, and a portion of the global memory. The distributed nodes communicate using asynchronous messages through a point-to-point network. The state of a cached copy is in either invalid, shared (readable), or exclusive (readable and writable). There are typical kinds of transactions as follows:

READ-Transaction If a read miss occurs in a processor where state of cache is invalid, the corresponding node sends out a GET request to the home (this step is not necessary if the requesting processor is in the home). Receiving the GET request, the home consults the directory corresponding to the memory line to decide what action the home should take. If the line is pending, meaning that another request is already being processed, the home sends a NAK (negative acknowledgment) to the requesting node. If the directory indicates there is a dirty copy in a remote node, then the home forwards the GET to that node. Otherwise, the home grants the request by sending a PUT to the requesting node and updates the directory properly. When the requesting node receives a PUT reply, which returns the requested memory line, the processor sets its cache state to shared and proceeds to read.

WRITE-Transaction For a write miss, the corresponding node sends out a GETX request to the home. Receiving the GETX request, the home consults the directory. If the line is pending, the home sends a NAK to the requesting node. If the directory indicates there is a dirty copy in a third node, then the home forwards the GETX to that node. If the directory indicates there are shared copies of the memory line in other nodes, the home sends INVs (invalidations) to those nodes. At this point, the protocol depends on which of two modes the multiprocessor is running in: EAGER or DELAYED. In EAGER mode, the home grants the request by sending a PUTX to the requesting node; in DELAYED mode, this grant is deferred until all the invalidation acknowledgments are received by the home. If there are no shared copies, the home sends a PUTX to the requesting node and updates the directory properly. When the requesting node receives a PUTX reply which returns an exclusive copy of the requested memory line, the processor sets its cache state to exclusive and proceeds to write.

Usually, only an informal account of FLASH is provided in most previous work as shown in the above paragraph. It is still not very clear to most readers, especially to the novel who are not very familiar with FLASH. Its is very desirable for people to have some flow-charts each of which is a series of rules

executed to complete a READ/WRITE transaction. From such a flow, who execute transition rules and exchange messages to implement the designed protocol goals. How rules are executed in some special causal order to finish a transaction? How do transactions of different nodes are synchronized with some mechanism to guarantee important properties? Here we adopt a novel data flow chart to illustrate a case of READ (or WRITE) transaction, as shown below:

3.2 Flow

Formally, a transaction flow is a structure, which can be formalized by pair $< R \times N, \rightarrow \cup \Rightarrow >$, where $(r, i) \in R \times N$ is a node, indicating an client i performs an action of executing a rule r. Relations $\rightarrow$ and $\Rightarrow$ are two orders over $R \times N$. $(r_1, i_1) \rightarrow (r_2, i_2) \equiv \exists v\ e.(v := e) \in \mathsf{act}(r_1) \wedge (v = e) \in \mathit{decompose}(\mathit{guard}(r_2))$. Here $v := e$ is an assignment; act is the set of assignments of an action of a rule; A formula f in conjunction form can be composed into a set of sub-formulas f_i, denoted as $\mathit{decompose}(f)$, such that each f_i is not of a conjunction form and f is semantically equivalent to $f_1 \wedge f_2 \wedge ... \wedge f_N$. Intuitively $(r_1, i_1) \rightarrow (r_2, i_2)$ means that the triggering of agent i_2 executing r_2 needs the completion of agent i_1 executing r_1; $(r_1, i_1) \Rightarrow (r_2, i_1)$ means that agent i_1's action of executing r_2 follows the completion of executing r_1. A classical three-hop flow case of a READ-transaction is shown as follows:

This flow is finished by the following steps:

1. An idle remote client *src* needs a shared copy of a memory line, executes rule PI_Remote_Get, and sends the GET request to home.
2. When the GET request from *src* arrives at the home, the home consults the directory to find that the line is dirty in *dst*, executes rule NI_Lcal_Get_Get: forwards the GET to *dst* by telling that *src* needs the data, and sets Dir.Pending TRUE to make no any other node's request to be processed during the sharing write-back procedure.
3. When *dst* receives the forwarded GET, the processor *dst* executes rule NI_Remote_Get_Put, sets its copy to shared state and issues a PUT to *src* conveying the dirty data, and issues a shwb_shwb (sharing write-back) conveying the dirty data at the same time.
4. When *home* receives this SHWB message, it executes the rule NI_ShWb, and writes the received data back to main memory and puts *src* on the sharer list.
5. When *dst* receives this PUT message, it executes the rule NI_Remote_Put. If there is not another INV message, then it uses the received data to update its cache line, and sets its cache to shared state.

Compared with a textual description, a flow description has the advantage that (1) causal order relations between agents' executing rules are illustrated clearly; (2) communications between agents are also illustrated. In section 5 and 6 we combine steps in a flow with auxiliary inductive invariants, by which we can make it clear that how FLASH guarantee the synchronization between different clients. Formally, auxiliary inductive invariants will show the synchronization mechanism in logical formulas.

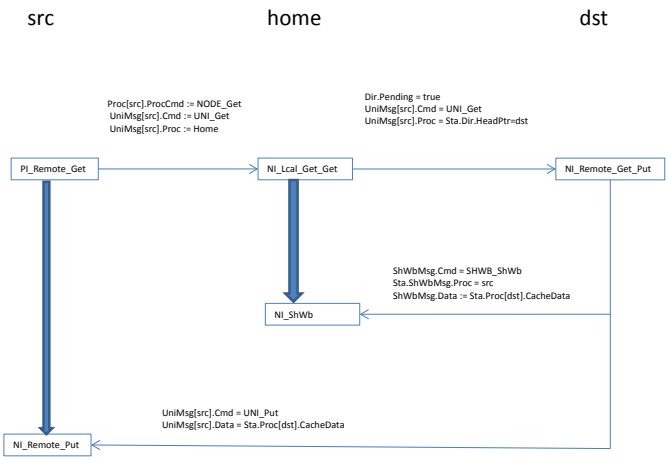

Fig. 2. The flow of READ-transaction involving sharing-write back

4 Formal Description of FLASH in Murphi

A formal protocol model includes a set of parameterized transition rules and properties. Here we adopt a version of FLASH similar to that in [7], which is different from the version in [6]. The difference between them lies in the modelling of behaviors of a Home node. Behaviours of the Home node is not identical to those of the other nodes, while behaviors of a non-Home node is identical to that of the other non-Home node. In the model of work in [6], state variables of the Home node and non-Home nodes are both array-variables. This modelling has a shortage of destroying the symmetry between indices which represents identifiers for agents.

In our modelling, the symmetry between indices should be preserved and will be used in paraVerifier. Namely A state variable a on non-Home nodes are stored in array-variables. Some parameterized variable $a[i]$ records some state on a of node i. Home is not an explicitly index to reference a node in our model. State of the Home node on a is recorded as global (or non-parameterized variable) $Homea$. Being consistent with the above philosphy, our FLASH in Murphi is changed accordingly. For instance, we compare the original version in [6], which is shown in (a), with our version, which is shown in (b):

```
UNI_MSG : record
  Cmd : UNI_CMD;
  Proc : NODE;
  Data : DATA;
end;
DIR_STATE : record
  Pending : boolean;
  Local : boolean;
  Dirty : boolean;
  HeadVld : boolean;
  HeadPtr : NODE;
  ShrVld : boolean;
  ShrSet : array [NODE] of boolean;
  InvSet : array [NODE] of boolean;
end;
ruleset src : NODE do
rule "PI_Remote_Get"
src != Home &
Sta.Proc[src].ProcCmd = NODE_None &
Sta.Proc[src].CacheState = CACHE_I
==>
begin
  Sta.Proc[src].ProcCmd := NODE_Get;
  Sta.UniMsg[src].Cmd := UNI_Get;
  Sta.UniMsg[src].Proc := Home;

endrule; endruleset;
```

(a)

```
UNI_MSG : record
  Cmd : UNI_CMD;
  Proc : NODE;
  HomeProc : boolean;
  Data : DATA;
end;
DIR_STATE : record
  Pending : boolean;
  Local : boolean;
  Dirty : boolean;
  HeadVld : boolean;
  HeadPtr : NODE;
  HomeHeadPtr : boolean;
  ShrVld : boolean;
  ShrSet : array [NODE] of boolean;
  HomeShrSet : boolean;
  InvSet : array [NODE] of boolean;
  HomeInvSet : boolean;
end;
ruleset src : NODE do
rule "PI_Remote_Get"
  Sta.Proc[src].ProcCmd = NODE_None &
  Sta.Proc[src].CacheState = CACHE_I
==>
begin   Sta.Proc[src].ProcCmd := NODE_Get;
  Sta.UniMsg[src].Cmd := UNI_Get;
  Sta.UniMsg[src].HomeProc := true;
endrule; endruleset;
```

(b)

In our version, we add a field $HomeProc$. If $HomeProc$ is true, then this is according to the case $Proc = Home$ in the original version, else the case where

$Proc$ is a non-Home node. $HomeHeadPtr$ is added similiarly in our model. $HomeShrSet$ and $HomeInvSet$ are added in our model to model $ShrSet[Home]$ and $InvSet[Home]$ to be true in the original version. We use the assignment $Sta.UniMsg[src].HomeProc := true$ to implicitly model the fact that $Sta.UniMsg[src].Proc$ is the $Home$ node.

There are three properties under verification.

```
invariant "CacheStateProp"
forall p : NODE do forall q : NODE do p != q ->
!(Sta.Proc[p].CacheState = CACHE_E & Sta.Proc[q].CacheState = CACHE_E)
end end;
invariant "CacheStatePropHome"
forall p : NODE do
!(Sta.Proc[p].CacheState = CACHE_E & Sta.HomeProc.CacheState = CACHE_E)
end;
invariant "MemDataProp"
!((Sta.Dir.Dirty = FALSE) & (!(Sta.MemData = Sta.CurrData)));
```

The former two properties are the mutual-exclusion properties between cache state status of two nodes. The last is the consistency property on data.

5 Verifying FLASH by paraVerifer

For cache coherence protocols like MESI or German protocols with small scales, paraVerifier is easy to use to verify automatically them. A Murphi file is only provided as a formal model, which contain both the protocol and properties under verification. The Murphi model will be automatically transformed into not only an internal model, but also a SMV-smodel. NuSMV will be called to compute the reachable set of the SMV-model. The internal model will be used by the invFinder to generate auxiliary invariants. During this generation procedure, a set of candidate invariant formulas are generated in one step, and only one is chosen if it is an invariant by checking the reachable set of SMV-model. The oracle of the SMV-model is automatically generated inside the inVfinder without human intervention.

However, FLASH with data path is a real world protocol with industry scale. For a FLASH instance configuration size NODENUM=3 ana DATANUM=2, the protocol is so complex that NuSMV can't compute the reachable state set of the SMV-model which is generated by invFinder in a computing server with ?. In fact, this problem of state explosion is the main obstacle to the applying previous automatical solutions such as [8,9] to the parameterized verification of FLASH. For instance, a reachable state set of a protocol instance of FLASH is need in both the work in [8,9]. The former needs the reachable state set to compute the so-called "invisible inductive invariants" for deductive theorem proving. The latter needs it to strengthen the guards of rule for an counter-example guided refinement of an abstract protocol. Due to the non-availability of the necessary the reachable state set of a proper instance of FLASH, the above two approach both fail.

In order to overrun the obstacle of oracles, invFinder provides techniques of simplifying protocols and hybrid oracles. For a complex protocol, a smv-model of a simplified version of this protocol can be used as an external oracle to verify the guessed candidate formulas. For FLASH with data paths, we construct a SMV -model of a simplified version of the FLASH without data paths, and use NuSMV to enumerate its reachable state set to judge a candidate invariant formula. Here we emphaszie that the reachable state set of the simplified FLASH instance with configuration size NODENUM=3 can be enumerated by the NuSMV in the aforementioned computing server. This oracle is enough for the invFinder to find all the invariant formulas on control variables (or without data properties).

However, if a candidate invariant formula in which data variables occurring, how does invFinder deal with it? For instance, ((Sta.Dir.Dirty = FALSE) & (!(Sta.MemData = Sta.CurrData))) cann't be judged via the SMV -model of a simplified version of the FLASH without data paths. At this time, a hybrid oracle will be used. A full protocol instance in Murphi with configuration size NODENUM=3 ana DATANUM=2, and Murphi is run to judge the formula containing data variables. Here we adopt an approximation strategy: if Murphi has been run to check the property for a time-out, and no counter-example is found, then the checked formula will be regarded for an invariant.

The approximation may introduce a false invariant, which leads to a doubt of the soundness of our method. Intuitively, a false invariant holds at any state which either occurs in the reachable state set of the smv-model or in a state traversed by Murphi. But it may not hold at a state of a protocol instance which is not in the aforementioned reachable state sets. Here we emphasize that the ultimate correctness is guaranteed by the theorem proving process of the generated proof script in Isabelle. If such a false invariant is generalized and occurs in the parameterized invariant set, the generated proof script can't be passed in Isabelle.

The verification command,

memory and time cost

5.1 Auxiliary Invariants and Causal Relations

Initially, paraVerifier sets the auxilary invariant set to be {`!(Sta.Proc[1].CacheState = CACHE_E & Sta.Proc[2].CacheState = CACHE_E), !(Sta.Proc[1].CacheState = CACHE_E & Sta.HomeProc.CacheState = CACHE_E), !((Sta.Dir.Dirty = FALSE) & (!(Sta.MemData = Sta.CurrData)))`}.

We also need tell the tool two protocol instances. One is with size 2 but without data paths (i.e., the variables such as Sta.MemData and Sta.CurrData can be omitted), and the second is with size 2 but with data paths. The reachable state set of the former can be enumerated by SMV, but that of the latter cann't be. paraVerifier need the two instances to verify a candidate formula which is guessed by heuristics. If all the variables occurring in the candidate formula is in the former instance, then paraVerifier will query the reachable state set to verify the formula; otherwise paraVerifier will call MURPHI to verify the formula. In the second kind of checking, a time limit will beset, if MURPHI can check the falsity

of the formula within the limit, then the candidate will be dropped; otherwise the candidate will be put it into the auxiliary invariant set.

After searching, the set of auxiliary invariants contains 162 formulas. This is fully AUTOMATICAL work done by paraVerifier. Here we select and analyze all the invariants on Sta.Dir.Pending, which help us to understand the function of the control variable.

```
inv__44!((Sta.HomeUniMsg.Cmd = uni_get) & (Sta.Dir.Pending = FALSE))
inv__52!((Sta.HomeUniMsg.Cmd = uni_getx) & (!(Sta.Dir.Pending = TRUE)))
inv__53!((Sta.HomeUniMsg.Cmd = uni_put) & (!(Sta.Dir.Pending = TRUE)))
inv__57!((Sta.HomeUniMsg.Cmd = uni_putx) & (!(Sta.Dir.Pending = TRUE)))
inv__59!((Sta.UniMsg[1].Cmd = uni_get) & (Sta.UniMsg[1].HomeProc = FALSE) & (Sta.Dir.Pending = FALSE))
inv__74!((Sta.UniMsg[1].Cmd = uni_getx) & (Sta.UniMsg[1].HomeProc = FALSE) &(!(Sta.Dir.Pending = TRUE)) )
inv__88!((Sta.Dir.InvSet[1] = TRUE) & (Sta.UniMsg[2].Cmd = uni_putx)& (Sta.Dir.Pending = FALSE))
inv__91!((Sta.ShWbMsg.Cmd = shwb_shwb) & (!(Sta.Dir.Pending = TRUE)))
inv__92!!((Sta.ShWbMsg.Cmd = shwb_fack) & (Sta.Dir.Pending = FALSE))
inv__111!((Sta.NakcMsg.Cmd = nakc_nakc) & (Sta.Dir.Pending = FALSE))
inv__114!((Sta.Dir.InvSet[1] = TRUE) & (Sta.Dir.Dirty = TRUE) & (Sta.Dir.Pending = FALSE))
inv__117!((Sta.Dir.InvSet[1] = TRUE) & (Sta.Dir.HeadVld = FALSE)& (Sta.Dir.Pending = FALSE))
inv__134!((Sta.Dir.InvSet[1] = TRUE) & (Sta.Proc[2].CacheState = cache_e)& (Sta.Dir.Pending = FALSE))
inv__153!((Sta.Dir.InvSet[1] = TRUE) & (Sta.WbMsg.Cmd = wb_wb) & (Sta.Dir.Pending = FALSE))
inv__161!((Sta.HomeProc.CacheState = cache_s) & (Sta.Dir.Pending = TRUE))
```

From these invariants, we can know when the control state variable Sta.Dir.Pending is set or not.

- Invariant 44 means that the Home node is fetching a data from some node to read, thus `(Sta.HomeUniMsg.Cmd = uni_get)`, so the pending flag is TRUE to block another new READ-WRITE requests.
- Invariants 52, 53, and 57 can be analyzed similarly.
- In invariant 59, `(Sta.UniMsg[1].Cmd = uni_get) & (Sta.UniMsg[1].HomeProc = FALSE)` means that node 1 has request a shared copy and been granted to fetch the copy from some node, thus Sta.UniMsg[1].HomeProc = FALSE, the pending flag is set TRUE to block another new READ-WRITE requests.
- Invariant 74 has a similar meaning while the request is for WRITE operation.
- Invariant 88 specifies that node 2 performs a write-request, thus `Sta.UniMsg[2].Cmd = uni_putx`, and the data is shared by node 1, then the shared copy in node 1 is invalidated, thus (Sta.Dir.InvSet[1] = TRUE), at this time, the pending flag is set TRUE to block another new READ-WRITE requests.
- Invariants 91 and 92 says that this flag is set during sharing write-back procedure. Invariants 111 that this flag is set during the nakc procedure when Sta.NakcMsg.Cmd = NAKC_Nakc.
- Invariants 114 that the flag is set during the invalidating procedure to an old shared-copy store in node 1. The invalidating procedure is a sub-procedure of a WRITE request from any other node. In 134, `Sta.Proc[2].CacheState = cache_e` shows that the WRITE request is from node 2, and CacheState of node 2 changes to exclusive even before node 1 has not been invalidated. From this, we can see that the version we verify is an eager-mode.
- Invarian 153 that Sta.Dir.Pending is set during a write-back procedure.
- The last one says that if there is a local shared-copy in the Home node, Sta.Dir.Pending is FALSE because the requests can be processed at once, thus the system need not be pended.

Not only auxiliary invariants are searched, but also causal relations between invariants and rules are searched. For instance, we select a fragment to show as follows:

```
ruleset dst : NODE do rule "NI_Remote_PutX"
Sta.UniMsg[dst].Cmd = UNI_PutX & Sta.Proc[dst].ProcCmd = NODE_GetX
==>
begin Sta.UniMsg[dst].Cmd := UNI_None; Sta.Proc[dst].ProcCmd := NODE_None;
Sta.Proc[dst].InvMarked := false; Sta.Proc[dst].CacheState := CACHE_E;
Sta.Proc[dst].CacheData := Sta.UniMsg[dst].Data;
endrule; endruleset;

rule: n_NI_Remote_PutX[1]; inv: !((Sta.Proc[2].CacheState = cache_e) & (Sta.Proc[1].CacheState
= cache_e)); g: TRUE;
rel: invHoldForRule3-inv4:!((Sta.Proc[2].CacheState = cache_e) & (Sta.UniMsg[1].Cmd = uni_putx))
rule: n_NI_Remote_PutX[2]; inv: !((Sta.Proc[2].CacheState = cache_e) & (Sta.Proc[1].CacheState
= cache_e)); g: TRUE;
rel: 3invHoldForRule3-inv4:!((Sta.Proc[1].CacheState = cache_e) & (Sta.UniMsg[2].Cmd = uni_putx))
rule: n_NI_Remote_PutX[3]; inv: !((Sta.Proc[2].CacheState = cache_e) & (Sta.Proc[1].CacheState
= cache_e)); g: TRUE;
rel: invHoldForRule2
```

where `n_NI_Remote_PutX[i]` is the rule instance by instantiating `NI_Remote_PutX` with actual parameter i$\in\{1,2,3\}$.

- Line 1 specifies that if a state s satisfies the guard of the rule, and the formula `inv4 2 1=!((Sta.Proc[2].CacheState = cache_e) & (Sta.UniMsg[1].Cmd = uni_putx))`, and s' is the post state after the execution of the rule, then the invariant `!((Sta.Proc[2].CacheState = cache_e) & (Sta.Proc[1].CacheState = cache_e))` holds at state s'. Reader can verify this easily. This expresses the intuition behind the causal relation invHoldForRule3. Line 2 can be analyzed similarly.
- Line 3 that `n_NI_Remote_PutX[3]` has nothing to do with all the variables in the invariant formula, thus s satisfies the formula if and only the post state s' s satisfies the formula. This expresses the intuition behind the causal relation invHoldForRule2.

5.2 Parameterized Proof Script

There are concrete invariants and causal relations, where each index is a concrete one such as 1 and 2, etc. `proofGen` generalizes these into a N-parameterized form, where each index is symbolic such as $i1$ and $i2$ and N is also symbolic. An Isabelle proof script is automatically generated by `proofGen`, where an N-prameterized instance of FLASH is modelled and the properties are formally proved. in detail, the proof script is divided into parts as follows:

1 Definitions of formally parameterized invariant formulas, which are generalized from concrete invariants. There are 161 such invariants. An actual N-parameterized invariant can be obtained by instantiating a formal invariant formula with symbolic indexes. All actual invariant formulas in the N-parameterized are defined by a set invariants N;

2 Definitions of formally parameterized rules, which can be directly transformed from the Murphi rules of FLASH. There are ? rules. Actual parameterized rules can be defined similarly. All actual invariant formulas in the N-parameterized are defined by a set rules N;

3 Definitions of specification of the initial state, which can be directly transformed from the `startstate` part of Murphi's code;

4 A lemma such as `ruleName_Vs_invName` on a causal relation of a rule and a parameterized invariant, which is proved by a formal proof automatically generated by `proofGen`. There are ? such lemmas , which is the product of the numbers of rules and invariants.

5 A Lemma such as `rules_invName` on causal relation for all rule and an invariant, which is proved by a formal proof automatically generated by `proofGen`. There are 161 such lemmas , which is the same as the numbers of invariants.

6 A lemma `rules_invs` on a causal relation for all rules and all invariants, which is proved by a formal proof automatically generated by `proofGen`.

7 A Lemma such as `iniImply_inv_i` on a fact that an invariant hold at the initial state defined by the specification of the initial state. There are 161 such lemmas, each of which can be proved by an `auto` command.

8 A lemma `on_inits` proves that for all invariants they hold at the initial state of the protocol.

9 Main theorem proving that any invariant formula hods at any reachable state of the N-parameterized FLASH protocol instance.

The Hierachy of the lemmas is illustrated as Fig. **??**. $A \rightarrow B$ means that the proof of lemma A needs applying lemma B.

```
definition inv1::"nat ⇒ nat ⇒ formula" where [simp]:
"inv1 i0 i1 ≡
(neg (andForm (eqn (IVar (Field (Para (Field (Ident ''Sta'') ''Proc'') i1) ''CacheState''))
(Const CACHE_E)) (eqn (IVar (Field (Para (Field (Ident ''Sta'') ''Proc'') i0) ''CacheState'')) (Const CACHE_E))))"
definition invariants::"nat ⇒ formula set" where [simp]:
"invariants N ≡ {f.
(∃ pInv0 pInv1. pInv0≤N∧pInv1≤N∧pInv0 =pInv1∧f=inv1 pInv0 pInv1) ∨
(∃ pInv0. pInv0 ≤N∧f=inv2 pInv0) ∨
(f=inv3 ) ∨ ......
(∃ pInv0. pInv0≤N∧f=inv162 pInv0)
}"
```

```
lemma n_NI_Remote_PutXVsinv1:
assumes a1: "(∃ dst. dst≤N∧r=n_NI_Remote_PutX dst)" and
 a2: "(∃ pInv0 pInv1. pInv0≤N∧pInv1≤N∧pInv0 =pInv1∧f=inv1 pInv0 pInv1)"
shows "invHoldForRule' s f r (invariants N)" (is "?P1 s ∨ ?P2 s ∨ ?P3 s")
proof -
from a1 obtain dst where a1:"dst≤N∧r=n_NI_Remote_PutX dst" apply fastforce done
from a2 obtain pInv0 pInv1 where a2:"pInv0≤N∧pInv1≤N∧pInv0 =pInv1∧f=inv1 pInv0 pInv1"
  apply fastforce done
have "(dst=pInv0)∨(dst=pInv1)∨(dst =pInv0∧dst =pInv1)" apply (cut_tac a1 a2, auto) done
moreover {
assume b1: "(dst=pInv0)"
have "?P3 s"
apply (cut_tac a1 a2 b1, simp, rule_tac x="(neg (andForm (eqn (IVar (Field (Para (Field (Ident
''Sta'') ''Proc'') pInv1) ''CacheState'')) (Const CACHE_E))
(eqn (IVar (Field (Para (Field (Ident ''Sta'') ''UniMsg'') pInv0) ''Cmd'')) (Const UNI_PutX ))))"
in exI, auto) done
then have "invHoldForRule' s f r (invariants N)" by auto }
moreover {
assume b1: "(dst=pInv1)"
have "?P3 s"
apply (cut_tac a1 a2 b1, simp, rule_tac x="(neg (andForm (eqn (IVar (Field (Para (Field (Ident
''Sta'') ''Proc'') pInv0) ''CacheState'')) (Const CACHE_E))
(eqn (IVar (Field (Para (Field (Ident ''Sta'') ''UniMsg'') pInv1) ''Cmd'')) (Const UNI_PutX))))"
in exI, auto) done
then have "invHoldForRule' s f r (invariants N)" by auto }
moreover {
assume b1: "(dst =pInv0∧dst =pInv1)"
have "?P2 s"
proof(cut_tac a1 a2 b1, auto) qed
then have "invHoldForRule' s f r (invariants N)" by auto }
ultimately show "invHoldForRule' s f r (invariants N)" by auto
qed
```

In order to understand the above proof, we can relate the proof with the three causal lines in Sect 5.1?. If we regard the three lines as a three test points of the causal relation between the parameterized rule `n_NI_Remote_PutXVsinv1` and invariant `inv1`, the proof is a natural generalization of the three tests. For any actual invariant parameters $i1$ and $i2$, any actual rule parameter ir which satisfy the assumption of the lemma, the causal relation between must be symmtric to the concrete relations listed in the three lines. Therefore each proof is an abstraction of a kind of concrete causal relations.

6 Illustrating a Typical Flow by With Invariants

FLASH has ? rules, each of which can be scheduled and executed if its guard are satifisfied. However, these transactions can be categoried into several groups. In a group, rules are organized by some partial order to implement a READ/WRITE transaction. In a transaction, there should be a commit step,

In order to understand state transitions of this flow after shwb_shwb is issued, we may observe the invariants containing the condition Sta.ShWbMsg.Cmd = shwb_shwb.

```
inv__13: !((!(Sta.ShWbMsg.Data = Sta.CurrData)) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv__21: !((Sta.Proc[1].CacheState = cache_e) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv__26: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.HomeProc.CacheState = cache_e))
inv__29: !((Sta.ShWbMsg.HomeProc = TRUE) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv__34: !((Sta.UniMsg[1].Cmd = uni_putx) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv__42: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.Dir.Dirty = FALSE))
inv__43: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.HomeUniMsg.Cmd = uni_putx))
inv__49: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.Dir.Local = TRUE))
inv__55: !((Sta.HomeUniMsg.Cmd = uni_put) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv__56: !((Sta.WbMsg.Cmd = wb_wb) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv__57: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.Dir.Pending = FALSE))
inv__58: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.HomeUniMsg.Cmd = uni_getx))
inv__70: !((Sta.HomeUniMsg.Cmd = uni_get) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv__75: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.Dir.InvSet[1] = TRUE))
inv__76: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.NakcMsg.Cmd = nakc_nakc))
inv__101: !((Sta.UniMsg[1].Cmd = uni_get) & (Sta.UniMsg[1].HomeProc = FALSE) &
    (Sta.ShWbMsg.Cmd = shwb_shwb))
inv__102: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.Dir.ShrSet[1] = TRUE))
inv__104: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.Dir.ShrVld = TRUE))
inv__105: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.UniMsg[1].Cmd = uni_getx) &
    (Sta.UniMsg[1].HomeProc = FALSE))
inv__135: !((Sta.Dir.HeadVld = FALSE) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv__146: !((Sta.Dir.HomeHeadPtr = TRUE) & (Sta.ShWbMsg.Cmd = shwb_shwb))
```

inv__13 says that Sta.ShWbMsg.Data must be the same as Sta.CurrData because *dst* conveys the dirty data when it issues the shwb_shwb command in executing rule NI_Remote_Get_Put. At the same time, *dst* also sets its copy to shared state, thus there is no an exclusive copy. This specified by inv__21 and inv__26. Because *src* is a remote node, Sta.ShWbMsg.HomeProc is set FALSE. This is formalized by inv__29. Because a sharing write-back occurs in a READ-transaction, and others' requests are pended to be processed, there will be no any uni_putx command to grant a node's WRITE-request. This formalized by inv__34 and inv__43. Home node itself will not grant its local WRITE-request, thus Sta.HomeUniMsg.Cmd can't be uni_getx. Due to the pending status, thus replacement to the exclusive copy and invalidating operation to shared nodes can't occur, thus Sta.WbMsg.Cmd can't be wb_wb, Sta.NakcMsg.Cmd can't be nakc_nakc, and Sta.Dir.InvSet tag to any node can't be TRUE. These are formalized by inv__56, inv__76, and inv__102. A sharing write-back occurs when Home node find that there is a remote and dirty copy, therefore Sta.Dir.Dirty must be TRUE and Sta.Dir.Local must be FALSE. This is formalized by inv__42 and inv__49. A dirty copy also means that no any other node shares this data, thus Sta.Dir.ShrSet of any node is set FALSE. This is formalized by inv__75. At the same time, Sta.Dir.HeadVld has been set TRUE and Sta.Dir.HomeHeadPtr has been set to *dst*. This formalized by inv__135 and inv__146 respectively. At the same time,

7 Conclusion

Our case studies on cache coherence protocols are typical examples to illustrate the guiding principle of paraVerifier. The consistency lemma based on the induction approach, is the core of our work, which gives the heuristics to guide the tool to search invariants. Instead of "invisible invariants" in previous work

(see e.g, [10], our invariants are visible, which can be further refined to precisely analyze the correctness of the protocol both in theoretical and practical aspects.

References

1. 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: Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on. (Apr 1994) 302–313
2. Owre, S., Rushby, J.M., , Shankar, N.: PVS: A prototype verification system. In Kapur, D., ed.: 11th International Conference on Automated Deduction (CADE). Volume 607 of Lecture Notes in Artificial Intelligence., Saratoga, NY, Springer-Verlag (jun 1992) 748–752
3. Park, S., Dill, D.L.: Verification of flash cache coherence protocol by aggregation of distributed transactions. In: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures, Padua, Italy, ACM (1996) 288–296
4. Mcmillan, K.L., Labs, C.B.: Parameterized verification of the flash cache coherence protocol by compositional model checking. In: In CHARME 01: IFIP Working Conference on Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science 2144, Springer (2001) 179–195
5. McMillan, K.: The cadence smv model checker `http://www.kenmcmil.com/smv.html`.
6. Chou, C.T., Mannava, P., Park, S.: A simple method for parameterized verification of cache coherence protocols. In: Proc. 5th International Conference on Formal Methods in Computer-Aided Design. Volume 3312 of LNCS. Springer (2004) 382–398
7. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaidi, F.: Invariants for finite instances and beyond. In: Formal Methods in Computer-Aided Design (FMCAD), 2013. (Oct 2013) 61–68
8. Arons, T., Pnueli, A., Ruah, S., Xu, Y., Zuck, L.: Parameterized verification with automatically computed inductive assertions? In: Proc. 13th International Conference on Computer Aided Verification. Volume 2102 of LNCS. Springer (2001) 221–234
9. Lv, Y., Lin, H., Pan, H.: Computing invariants for parameter abstraction. In: Proc. the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign, IEEE CS (2007) 29–38
10. Pnueli, A., Ruah, S., Zuck, L.: Automatic deductive verification with invisible invariants. In: 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Volume 2031 of LNCS. Springer (2001) 82–97