An Automatic Parameterized Verification of the FLASH Cache Coherence Protocol

Yongjian Li and Kaiqiang Duan

State Key Laboratory of Computer Science, Chinese Academy of Sciences

Abstract. The FLASH protocol is an industrial-scale cache coherence protocol, whose parameterized verification is a notoriously hard challenge to the field of formal verification. In this paper, we show how to verify its important properties by our tool paraverifier. Being distinguished from any other approach, our verification product is a formal proof with a set of inductive invariants. On one hand, invariants are searched automatically and the formal proof is generated automatically in our work, thus tedious human labor can be avoided. On the other hand, the formal proof guarantees the most rigorous correctness of the parameterized verification. Besides, we use a flow chart of a READ-transaction of FLASH to illustrate the semantic intuition behind these invariants, thus our verification product is not only a certification of correctness, but also a comprehensive analysis report.

1 Introduction

Verification of parameterized concurrent systems is interesting in the area of formal verification, 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.* The challenge posed by parameterized verification is that the desired properties should hold in any instance of the parameterized system. In this work, we will focus on a real-world cache coherence protocol - Stanford FLASH protocol [1].

The FLASH protocol is a publicly-recognized challenging benchmark in the field of formal verification. In the previous work, Park and Dill applied the general purpose theorem prover PVS [2] to verify 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 the FLASH protocol, and need to 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. Inductive invariants are provided by hand, and the theorem prover must be manually guided to perform the induction proof to prove the correspondence between the simplified protocol and the original one.

McMillan applies methods of compositional model checking [4] to the verification of the FLASH protocol by using Cadence-SMV [5]. Safety and liveness of the FLASH protocol are both verified. Despite the fact that SMV is designed as a model-checker to check automatically properties, the core techniques for the

FLASH protocol parameterized verification adopted by McMillan is SMV's advanced features for proofs 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 insight into the FLASH protocol. Unlike a general theorem prover like Isabelle, SMV does not have a good logical foundation and mechanism to do theorem proving. Briefly speaking, the above proofs have the cons as follow: lacking of rigorousness as a theorem prover, hard understanding for non-specialists of SMV, as well as the limitation of generalization to other protocol case studies and implementation in general-purpose theorem provers.

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. The CMP method has been extended by [7] that the "message flow" can be used as invariants. These "flow" invariants is obtained manually by people who carefully analyze the design documents. However, the downside of the CMP method is that 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 this analysis and some effective automatic tool is needed.

In [8], an algorithm, called BRAB, is implemented in a SMT-based model checker Cubicle. It 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 that a completely formal proof is constructed to verify the full-version of the FLASH protocol by adopting the invariants computed by BRAB.

In [10], we do a successful experiment for parameterized verification of a simplified version of the FLASH protocol, which abstracts data-paths, but fails in the verification of the full version. In fact, an important safety property is on the consistency between cached data and stored data in main memory. This work continues developing our tool to improve its power to conquer this hard problem.

To sum up, the FLASH protocol is a litmus test 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 the FLASH protocol, then there is a good chance that it will also work on many real-world cache coherence protocols." Second, the FLASH protocol is sufficiently hard that only two or three methods have fully verified the protocol parametrically. However, human guidance still plays a key role in the above successful verification work for the FLASH protocol. This fact reveals the weakness of automatic methods in the field of parameterized verification. Third, further efforts are still needed for clear mechanization. As argued in [6], "the first priority is clear 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 the correctness of the FLASH protocol in a well-known theorem prover. Previous work is too far away from giving a formal proof for the FLASH protocol. Even for a moderate case GERMAN protocol [9], which is much simpler than the FLASH protocol, no formal proof is not available yet. Let alone the FLASH protocol.

The aim of this paper is to apply paraVerifier [10] to a parameterized verification of the full version of FLASH protocol in both an automatic and rigorous way. In detail,

- Interesting auxiliary invariants can be automatically found by paraVerifier. Our invariants are easily readable, which can characterize the semantic features of the FLASH protocol, and help people to precisely understand the design of the FLASH protocol.
- With the help of paraVerifier, a formal proof can be constructed automatically as a product of a parameterized verification of the FLASH protocol. 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 the FLASH protocol.

Novel features of our work, which can also be generalized and applied to the domain of verification, are as follows:

- With the novel actual-parameter instantiation policy for parameterized rules of a protocol like FLASH, we can sample interested concrete causal relations from information obtained from model checking; with the case generation strategy, a formal proof can be automatically derived for the verification of properties. In this sense, we can say that the protocol under verification can be made proof-carrying.
- In order to overrun the obstacle of constructing symbolic and reachable state set of a complex protocol like FLASH, we can derive most interesting invariants by model-checking the reachable state set of a simplified FLASH, which is obtained by data abstraction; Combing with explicit model checking techniques like Murphi, we can derive the other invariants on data by

model-checking them in the full version FLASH with some time-out. The abstraction and hybrid use of model checking techniques may derive a false invariant, but the ultimate correctness is guaranteed by the formal proof.
– The invariants searched by paraVerifier are refined and easily readable. These invariants have shed light on the semantics of the protocol. We relate a flow chart of a READ-transaction of FLASH to illustrate the semantic intuition behind these invariants. This illustration is valuable to the practical designer and engineer to implement the protocol.

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 are bug-free. There is still a big gap between the bug-freeness in quite a small number of instances and 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.

Theoretical foundation paraVerifier is based on a simple but elegant theory. A novel feature of our work lies in that so-called three kinds of causal relation is exploited, which captures whether and how the execution of a particular protocol rule changes the protocol state variables appearing in an invariant. Here basic knowledge on protocol formalisation is assumed. Consider a rule r, a formula f, and a formula set fs, let $\mathsf{guard}(r)$ and $\mathsf{action}(r)$ be the guard and the action of the rule r, $\mathsf{WP}(f, \mathsf{action}(r))$ be the weakest precondition to make f to be true after execution of $\mathsf{action}(r)$.

In Hoare logic, a Hoare triple has the form of $\{f\}S\{f'\}$ where f and f' are assertions of formulas and S is a statement. f is called the precondition and f' the postcondition: when the precondition is met, executing S establishes the postcondition. We can interpret the intuition behind above three kinds of causality relation in Hoare triples as below:

1. $\mathsf{CR}_1(s, f, r)$ iff $\{\mathsf{guard}(r)\}\mathsf{action}(r)\{f\}$
2. $\mathsf{CR}_2(s, f, r)$ iff $\{\mathsf{guard}(r) \wedge f\}\mathsf{action}(r)\{f\}$
3. $\mathsf{CR}_3(s, f, r, F)$ iff $\exists f' \in F$ such that $\{\mathsf{guard}(r) \wedge f'\}\mathsf{action}(r)\{f\}$

$\mathsf{CRS}(s, f, r, F)$ is the disjunction of CR_{1-3}, and can be considered as a kind of general inductive tactics. Namely, a property f in F holds at a state s, and $\mathsf{CRS}(s, f, r, F)$, then f also holds at the post-state s' after a rule r is executed.

With these causal relations, we define a relation $\mathsf{consistent}(invs, inis, rs)$ between a protocol $(inis, rs)$ and a set of invariants $invs = \{inv_1, \ldots, inv_n\}$ if the following conditions hold: (1) for any formula $inv \in invs$ and $ini \in inis$ and any state s, ini holding at s implies inv holding at state s; (2) for any formula $inv \in invs$ and rule $r \in rs$ and any state s, $\mathsf{CR}_{1-3}(s, inv, r, invs)$. Then, a so-called consistency lemma is proposed, which is the corner-stone in our work:

for a protocol $P = (ini, rs)$, if $\mathsf{consistent}(invs, ini, rs)$, and $s \in \mathsf{reachableSet}(P)$, then for all inv s.t. $inv \in invs$, $s \models inv$.

In order to apply the consistency lemma to prove that a given property inv (e.g., the mutual exclusion property) holds for each reachable state of a protocol $P = (inis, rs)$ (e.g., the FLASH protocol), we need to solve two problems. First, we need to construct a set of auxiliary invariants $invs$ which contains inv and satisfies $\mathsf{consistent}(invs, inis, rs)$. By applying the consistency lemma, we decompose the original problem of invariant checking into that of checking the causal relation between some $f \in invs$ and $r \in rs$. The latter needs case analysis on the form of f and r. Only if a proof script contains *sufficient information on the case splitting and the kind of causal relation to be checked in each subcase*, Isabelle can help us to automatically check it. How to generate automatically such a proof within the ability of an automatical tactic provided by Isabelle is the second problem.

Our solutions to the two problems are as follows: Given a protocol, `invFinder` finds all the necessary ground auxiliary invariants from a small instance of the protocol in Murphi. This step solves the first problem. A table `protocol.tbl` is worked out to store the set of ground invariants and causal relations, which are then used by `proofGen` to create an Isabelle proof script which models and verifies the protocol in a parameterized form. In this step, ground invariants are generalized into a parameterized form, and accordingly ground causal relations are adopted to create parameterized proof commands which essentially proves the existence of the parameterized causal relations. This solves the second problem. At last, the Isabelle proof script is fed into Isabelle to check the correctness of the protocol. An overview of our method is illustrated in Fig. 1.

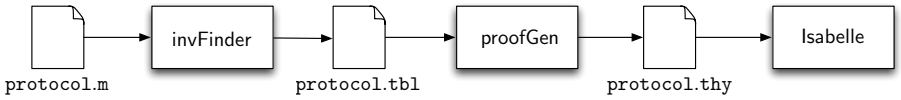

Fig. 1. The workflow of paraVerifier

3 Informal Account of the FLASH Protocol

A textual description Firstly we give a classical textual description, which is directly cited from the material in [13] to informally introduce the FLASH protocol. The FLASH protocol is a directory-based one which supports a large number of distributed processing nodes. Each node 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). Each cache line-sized block in memory is associated with directory header which keeps information about the cache line. For a memory address, 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 in its main memory in the corresponding directory headers.

There are typical kinds of transactions as follows:

READ-Transaction If a read miss occurs in a processor where the state of its

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 an 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 address, 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 cache line is pending, the home sends an 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 address 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 address, the processor sets its cache state to exclusive and proceeds to write.

Usually a flow-chart, which is a series of rules executed to complete a transaction, is a desirable way to illustrate a protocol. Such a flow shows: Who do execute transition rules? How are rules executed in some special causal order to finish a transaction? How are transactions of different nodes synchronized with some mechanism to guarantee important properties?

A Flow Chart Formally, a transaction flow is a well-founded graph, which can be formalized by a pair $< R \times N, \rightarrow \cup \Rightarrow >$, where $(r, i) \in R \times N$ is a node, indicating that a remote node i performs an action of executing a rule r. Relations $\rightarrow$ and $\Rightarrow$ are two kinds of edges over $R \times N$. This kind of well-founded graph is similar to the bundle which is used to formalise a protocol execution in security protocol [11]. $(r_1, i_1) \rightarrow (r_2, i_2) \equiv \exists v\ e.(v := e) \in \mathsf{action}(r_1) \wedge (v = e) \in \mathsf{decompose}(\mathsf{guard}(r_2))$. Here $v := e$ is an assignment; action is the set of assignments of an action of a rule; A formula f in conjunction form can be composed of a set of sub-formulas f_i, denoted as $\mathsf{decompose}(f)$, 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.

The three-hop flows is the most novel feature of the FLASH protocol. Different from other cache protocols that all messages of remote nodes must be

transferred by their home node, two remote nodes of the FLASH protocol can communicate directly. A classical three-hop flow case of a READ-transaction including a sharing-write back is shown in Fig. 2:

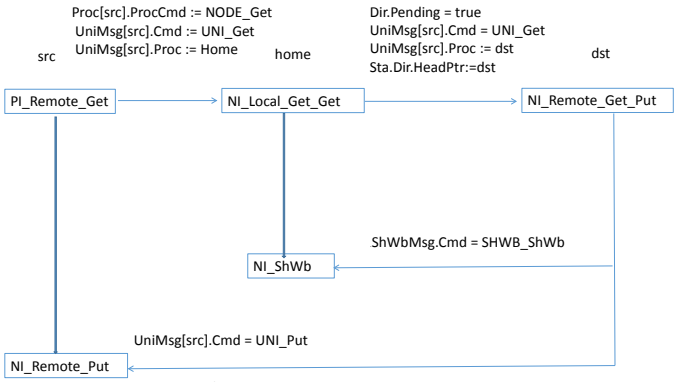

Fig. 2. The flow of READ-transaction including a sharing-write back

This flow consists of the following steps:

1. An idle remote node *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 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 data back to main memory and puts *src* on the sharers' 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 nodes' executing rules are illustrated clearly; (2) communications between nodes 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 how the FLASH protocol guarantees the synchronization between different nodes. Formally, auxiliary inductive invariants will show the synchronization mechanism in logical formulas. It is also noteworthy that the purpose about the flow in our paper is completely different with [7]. In [7], the flow message is recognized by people manually and is used as flow invariants to assist the verification. As comparison, we extract the flow knowledge, as well as with the generated invariants, to help user better understanding the system.

4 Formal Description of the FLASH protocol in Murphi

A formal protocol model includes a set of parameterized transition rules and properties. Here we adopt a version of the FLASH protocol similar to that in [8], which is different from the version in [6]. The only difference between [8] and [6] is that we abstracted away the *Home* processor and for all arrays indexed by *Home*, each occurrence of $A[Home]$ was replaced by a global variable A_home. Following the modelling way in [8], we can preserve the full symmetry between indices of array variables, which plays a key role in our generalization strategy.

```
UNI_MSG : record
  Cmd : UNI_CMD;
  Proc : NODE;
  HomeProc : boolean;
  Data : DATA;
end;
ruleset src : NODE; pp : NODE do
rule "NI_Local_GetX_PutX_8"
  Sta.UniMsg[src].Cmd = UNI_GetX &
  Sta.UniMsg[src].HomeProc &
  !Sta.Dir.Pending &
  !Sta.Dir.Dirty & Sta.Dir.HeadVld &
Sta.Dir.HeadPtr = src & !Sta.Dir.HomeHeadPtr &
  Sta.Dir.ShrSet[pp] &
  Sta.Dir.Local & Sta.HomeProc.ProcCmd !=
NODE_Get
==>
begin

  for p : NODE do
    Sta.Dir.ShrSet[p] := false;
    if ( p != src &
    ( Sta.Dir.ShrVld & Sta.Dir.ShrSet[p] |
    Sta.Dir.HeadVld & Sta.Dir.HeadPtr = p &
!Sta.Dir.HomeHeadPtr) ) then
      Sta.Dir.InvSet[p] := true;
      Sta.InvMsg[p].Cmd := INV_Inv;
    else
      Sta.Dir.InvSet[p] := false;
      Sta.InvMsg[p].Cmd := INV_None;
    end;
  end;

  Sta.Dir.Pending := true;
  Sta.Dir.Local := false;
  Sta.Dir.Dirty := true;
  Sta.Dir.HeadVld := true;
  Sta.Dir.HeadPtr := src;
  Sta.Dir.HomeHeadPtr := false;
  Sta.Dir.ShrVld := false;
  Sta.Dir.HomeShrSet := false;
  Sta.Dir.HomeInvSet := false;
  Sta.HomeInvMsg.Cmd := INV_None;

  Sta.UniMsg[src].Cmd := UNI_PutX;

  Sta.HomeProc.CacheState := CACHE_I;
  Sta.UniMsg[src].Data := Sta.MemData;
endrule;
endruleset;

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)));
```

We list a rule `NI_Local_GetX_PutX_8` and three properties under verification. This rule is done by the *Home* node to deal with a `PUTX` request to invalidate all shared copies. There are three properties under verification. The former two properties are the mutual-exclusion properties between cache state status of two nodes. The last is the data property. Our experiment data includes the paraVerifier instance, invariant sets, Isabelle proof scripts [14].

5 Verifying the FLASH protocol by paraVerifer

Verifying procedure In order to verify the FLASH protocol by paraVerifier under Linux system, some simple steps are needed. (1) The Murphi model of the FLASH protocol is converted to its corresponding internal model by compiler murphi2ocaml. (2) The specific verifier for FLASH protocol with OCaml is built. Furthermore, the generated executable file is executed on a computing

server while a model checking engine for NuSMV oracle and a model checking engine for Murphi oracle are executed on other computing servers respectively. Afterwards, a directory containing proof scripts will be generated. (3) The proof scripts in Isabelle is then executed.

We run paraVerifier on a server with 4 Intel Xeon processors, 8 GB memory and 64 bit Linux 3.15.10. At the same time, the NuSMV oracle and Murphi oracle were set on a server with 32 Inter Xeon processors, 384 GB memory and 64 bit Linux 2.6.32. Result of our experiments is as Table 1.

invFinder Starting from a given set of initially given concrete invariants, invFinder works iteratively in a semi-proving and semi-searching way, repeatedly instantiate a rule r into some concrete rule according to a novel parameter instantiation policy, tries to find new invariants, in the form of concrete formulas, by constructing the causal relation between the invariants and these concrete rules. It uses oracles that checks whether a concrete formula is a tautology or an invariant from a set of candidates of formulas in the small reference model of the protocol invFinder stops until no new invariants can be found. The output of invFinder is stored in file `flash.tbl`. The column `ruleParas` lists groups of actual parameters which are used to instantiate r, column `CR` the causal relation between the concrete rule and the concerete invariant, and `f'` the formula to make the CR_3 relation hold. Now oracles used in paraVerifier are Murphi, NuSMV, and SMT solver Z3. As we mention, a model written in Murphi is provided as a formal model, which contains both the protocol and properties under verification. In this the model, the model sizes (namely the number of $NODE$ and $DATA$) are set. The Murphi model will be automatically transformed into not only an internal model but also an SMV-model. NuSMV will be used as the model checking engine and 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 formulas are generated in one step, and only one is chosen if it is an invariant by checking the reachable set of SMV-model. NuSMV-oracle is preferred if the reachable state set is available and the query will be answered immediately.

Table 1. Verification Result

#rules	62
#invariants	162
Time of paraVerifier (minutes)	9.82
Memory of paraVerifier (MB)	178
Time of proof (hours)	10.76

Data abstraction and hybrid oracles However, the FLASH protocol with data path is a real world protocol with industry scale. Here data path means that all the variable of type `DATA` and operations on these variables, which models the cached data in a processor node or a data stored in the main memory in FLASH. For a FLASH protocol instance configuration size NODENUM=3 and DATANUM=2, the protocol is so complex that NuSMV cannot compute the reachable state set of the SMV-model which is generated by invFinder in a computing server with 32 Intel Xeon processors, 384 GB memory. In fact, this problem of state explosion is the main obstacle to the application of previous automatic solutions such as [9,15] to the parameterized verification of the

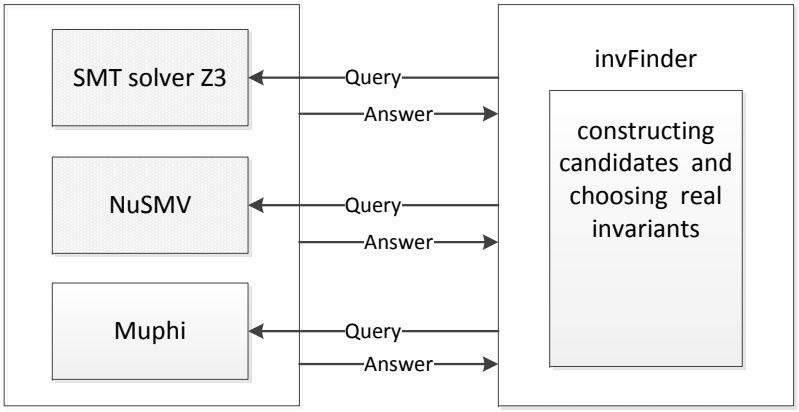

Fig. 3. Oracles Used in the invFinder

FLASH protocol. For instance, a reachable state set of a protocol instance of the FLASH protocol is needed in both the work in [9,15]. 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 rules for a counter-example guided refinement of an abstract protocol. Due to the non-availability of the necessary reachable state set of a proper instance of the FLASH protocol, the above two approaches both fail.

In order to overrun the obstacle of constructing oracles, invFinder provides techniques of simplifying protocols by data abstraction and hybrid oracles. For the FLASH protocol with data paths, we construct a SMV -model of a simplified version of the FLASH protocol by data abstraction. Here data abstraction means that we abstract (or remove) all the variables and operations on data. For instance, in the simplified FLASH with data abstraction, the UNI_MSG type will be simplified as `UNI_MSG : record Cmd : UNI_CMD; Proc : NODE; HomeProc : boolean; end`, and the statement `Sta.UniMsg[src].Data := Sta.MemData` is removed in the rule `NI_Local_GetX_PutX_8`. If a rule in FLASH protocol like `Store` only deals with data, it can be omitted in the simplified FLASH.

We can explicitly tell invFinder to use the simplified FLASH NuSMV reference model, and use NuSMV to enumerate its reachable state set to judge a candidate invariant formula. Here we emphasize that the reachable state set of the simplified FLASH protocol 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 without data properties.

However, if variables on data occurs in a candidate formula, how does invFinder deal with it? For instance, ((Sta.Dir.Dirty = FALSE) & (!(Sta.MemData =

Sta.CurrData))) can't be judged via the SMV -model of a simplified version of the FLASH protocol. At this time, a Murphi oracle will be used. A full protocol instance in Murphi with configuration size NODENUM = 3 and DATANUM = 2 is built, and Murphi is run to judge the formula containing data variables. Here we adopt a heuristic strategy: if Murphi has been running to check the property up to a time-out, and no counter-example is found, then the checked formula will be regarded as an invariant.

The heuristic 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.

Auxiliary Invariants The set of auxiliary invariants, which is found by paraVerifier, contains 162 formulas. This work is fully AUTOMATICALLY 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.

```
inv44:!((Sta.HomeUniMsg.Cmd = uni_get) & (Sta.Dir.Pending = FALSE))
inv52:!((Sta.HomeUniMsg.Cmd = uni_getx) & (!(Sta.Dir.Pending = TRUE)))
inv53:!((Sta.HomeUniMsg.Cmd = uni_put) & (!(Sta.Dir.Pending = TRUE)))
inv57:!((Sta.HomeUniMsg.Cmd = uni_putx) & (!(Sta.Dir.Pending = TRUE)))
inv59:!((Sta.UniMsg[1].Cmd = uni_get) & (Sta.UniMsg[1].HomeProc = FALSE) & (Sta.Dir.Pending = FALSE))
inv74:!((Sta.UniMsg[1].Cmd = uni_getx) & (Sta.UniMsg[1].HomeProc = FALSE) &(!(Sta.Dir.Pending = TRUE)))
inv88:!((Sta.Dir.InvSet[1] = TRUE) & (Sta.UniMsg[2].Cmd = uni_putx)& (Sta.Dir.Pending = FALSE))
inv91:!((Sta.ShWbMsg.Cmd = shwb_shwb) & (!(Sta.Dir.Pending = TRUE)))
inv92:!((Sta.ShWbMsg.Cmd = shwb_fack) & (Sta.Dir.Pending = FALSE))
inv111:!((Sta.NakcMsg.Cmd = nakc_nakc) & (Sta.Dir.Pending = FALSE))
inv114:!((Sta.Dir.InvSet[1] = TRUE) & (Sta.Dir.Dirty = TRUE) & (Sta.Dir.Pending = FALSE))
inv117:!((Sta.Dir.InvSet[1] = TRUE) & (Sta.Dir.HeadVld = FALSE)& (Sta.Dir.Pending = FALSE))
inv134:!((Sta.Dir.InvSet[1] = TRUE) & (Sta.Proc[2].CacheState = cache_e)& (Sta.Dir.Pending = FALSE))
inv153:!((Sta.Dir.InvSet[1] = TRUE) & (Sta.WbMsg.Cmd = wb_wb) & (Sta.Dir.Pending = FALSE))
inv161:!((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. In the following analysis, an index 1 can be generalized into any index in one invariant, two indices 1 and 2 can be generalized into any two indices i_1 and i_2 s.t. $i_1 \neq i_2$.

- 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 request.
- Invariants 52, 53, and 57 can be analyzed similarly.
- In invariant 59, `(Sta.UniMsg[1].Cmd = uni_get) & (Sta.UniMsg[1].Home Proc = FALSE)` means that node 1 has requested a shared copy and been granted to fetch the copy from some node, thus Sta.UniMsg[1].HomeProc = FALSE, the pending flag is set to 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 request.
– 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 states 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.
– Invariant 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 pending.

Not only auxiliary invariants are searched, but also causal relations between invariants and rules are searched. A fragment on the rule `NI_Local_GetX_PutX_8` and invariant `inv16(1,2)` is shown as (a) in Fig. 4.

(a) a fragment of concrete causal relations

rule	ruleParas	inv	CR	f'
r	[1, 2]	cf	CR_3	$inv_{17}(2)$
r	[2, 1]	cf	CR_3	$inv_{17}(1)$
r	[1, 3]	cf	CR_2	$inv_{17}(2)$
r	[3, 1]	cf	CR_2	
r	[2, 3]	cf	CR_2	$inv_{17}(1)$
r	[3, 2]	cf	CR_2	
r	[3, 4]	cf	CR_2	

(b) The generalized causal relations from (a)

rule	cases	inv	CR	f'
r	$iR_1 = i_1 \wedge iR_2 = i_2$	f	CR_3	$inv_{17}(i_2)$
r	$iR_1 = i_2 \wedge iR_2 = i_1$	f	CR_3	$inv_{17}(i_1)$
r	$iR_1 = i_1 \wedge \bigwedge^2_{j=1} iR_2 \neq i_j$	f	CR_3	$inv_{17}(i_2)$
r	$\bigwedge^2_{j=1} iR_1 \neq i_j \wedge iR_2 = i_1$	f	CR_2	
r	$iR_1 = i_2 \wedge \bigwedge^2_{j=1} iR_2 \neq i_j$	f	CR_3	$inv_{17}(i_1)$
r	$\bigwedge^2_{j=1} iR_1 \neq i_j \wedge iR_2 = i_2$	f	CR_2	
r	$\bigwedge^2_{j=1} iR_1 \neq i_j \wedge \bigwedge^2_{j=1} iR_2 \neq i_j$	f	CR2	

Fig. 4. Comparison of concrete causal table and symbolic causal table

,where $r = NI\_Local\_GetX\_PutX\_8(iR_1, iR_2)$, $f = inv_{16}(i_1, i_2) = \neg((Sta.UniMsg[i_1].Cmd = uni\_putx) \wedge (Sta.UniMsg[i_2].Cmd = uni\_putx))$, $inv_{17}(i_1) = \neg(((Sta.UniMsg[i_1].Cmd = uni\_putx) \wedge (Sta.Dir.Dirty = FALSE))$, $cf = inv_{16}(1, 2)$.

Causal Relation Generalization and Parameterized Proof Script There are concrete invariants and causal relations in Table (a) in Fig. 4, where each index is a concrete one such as 1 and 2, etc. `proofGen` generalizes these into an N-parameterized form, where each index is symbolic such as i_1 and i_2 and N is also symbolic. Basing on a novel *case generation* strategy by comparing actual rule parameters and actual parameters occurring in cf, `proofGen` generalizes the concrete causal relation into a symbolic one, as shown in (b) in Fig. 4.

Basing on the symbolic causal relation table, the proof of lemma `NI_Local_GetX_PutX8_Vsinv_16`, which proves the existence of causal relation between the rule

NI_Local_GetX_PutX8 and invariant inv_16 can be easily be constructed because the information of case splitting in a proof doing case analysis. Namely, the assumption of each subcase and the tactic among CR_{1-3} in each case are indicated explicitly. The transformation strategy from the symbolic causal table to a proof of a subcase is shown in Fig. 5.

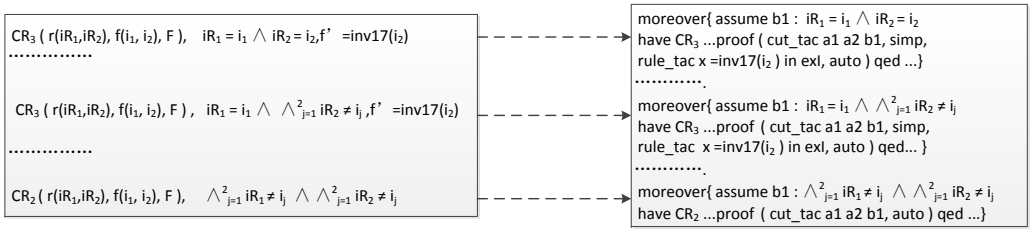

Fig. 5. From symbolic causal table to a proof

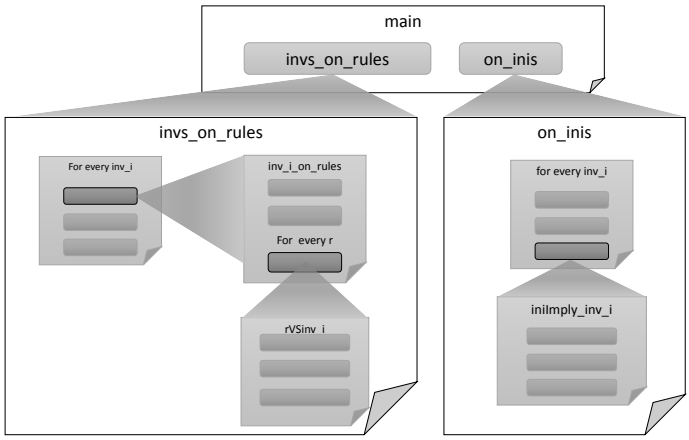

Fig. 6. The Hierachy of the lemmas

The Hierarchy of the lemmas in a generated proof script is illustrated in Fig. 6. In detail, the proof script is divided into parts as follows:

1 Definitions of formally parameterized invariant formulas.There are 162 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. There are 62 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.
4 A lemma such as `ruleName_Vs_inv_i` on a causal relation of a rule and a parameterized invariant.
5 A Lemma such as `rules_inv_i` on causal relation for all rules and an invariant *inv_i*.

6 A lemma `rules_invs` on a causal relation for all rules and all invariants.
7 A Lemma such as `iniImply_inv_i` on a fact that an invariant hold at the initial state.
8 A lemma `on_inits` proves that all invariants hold at the initial state.
9 Main theorem applying the consistency lemma to prove that any invariant holds at any reachable state of a parameterized FLASH protocol instance.

6 Illustrating Invariants With a Typical Flow

```
inv13: !((!(Sta.ShWbMsg.Data = Sta.CurrData)) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv21: !((Sta.Proc[1].CacheState = cache_e) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv26: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.HomeProc.CacheState = cache_e))
inv29: !((Sta.ShWbMsg.HomeProc = TRUE) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv34: !((Sta.UniMsg[1].Cmd = uni_putx) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv42: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.Dir.Dirty = FALSE))
inv43: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.HomeUniMsg.Cmd = uni_putx))
inv49: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.Dir.Local = TRUE))
inv55: !((Sta.HomeUniMsg.Cmd = uni_put) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv56: !((Sta.WbMsg.Cmd = wb_wb) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv57: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.Dir.Pending = FALSE))
inv58: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.HomeUniMsg.Cmd = uni_getx))
inv70: !((Sta.HomeUniMsg.Cmd = uni_get) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv75: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.Dir.InvSet[1] = TRUE))
inv76: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.NakcMsg.Cmd = nakc_nakc))
inv101: !((Sta.UniMsg[1].Cmd = uni_get) & (Sta.UniMsg[1].HomeProc = FALSE) &
    (Sta.ShWbMsg.Cmd = shwb_shwb))
inv102: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.Dir.ShrSet[1] = TRUE))
inv104: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.Dir.ShrVld = TRUE))
inv105: !((Sta.ShWbMsg.Cmd = shwb_shwb) & (Sta.UniMsg[1].Cmd = uni_getx) &
    (Sta.UniMsg[1].HomeProc = FALSE))
inv135: !((Sta.Dir.HeadVld = FALSE) & (Sta.ShWbMsg.Cmd = shwb_shwb))
inv146: !((Sta.Dir.HomeHeadPtr = TRUE) & (Sta.ShWbMsg.Cmd = shwb_shwb))
```

In order to understand state transitions of the flow in Fig. 2 after shwb_shwb is issued, we may observe the invariants containing the condition Sta.ShWbMsg.Cmd = shwb_shwb.

- inv13 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 inv21 and inv26.
- Because *src* is a remote node, Sta.ShWbMsg.HomeProc is set FALSE. This is formalized by inv29.
- 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 inv34 and inv43.
- 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 inv56, inv76, and inv102.
- 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 inv42 and inv49.

– 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 inv75.
– Meanwhile, Sta.Dir.HeadVld has been set TRUE and Sta.Dir.HomeHeadPtr has been set to *dst*. This formalized by inv135 and inv146 respectively.
– Sta.UniMsg[src].Cmd is set uni_put after the rule NI_Remote_Get_Put. When (Sta.UniMsg[src].HomeProc = FALSE &Sta.ShWbMsg.Cmd = shwb_shwb), Sta.UniMsg[src].Cmd can not be uni_get, thus inv101 holds for src.
– For other node src' than src, its UniMsg[src'].HomeProc flag is initialized by TRUE, thus inv101 holds for src'.

7 Conclusion and Future Work

Safety or Security critical computer system, such as nuclear plant control system or trusted operating system, usually have the requirements of formal proof, rather than verification. These systems are often parameterized systems. Our case study on the FLASH protocol is a typically successful parameterized verification of paraVerifier which combines model checking and theorem proving. The consistency lemma basing on the induction approach is the core of our work, which gives the heuristics to guide the tool to construct candidate invariants. By using NuSMV or Murphi to model check candidate invariants, true invariants are chosen. By generalizing the invariants into a parameterized form, paraVerifier generates automatically a proof script to prove the protocol. Again, the consistency lemma is used as a main theorem to prove all the invariants. Searching invariants and proof generation are both automatic. The ultimate correctness is guaranteed by a mechanical proof in a theorem prover. Therefore, we achieve the two goals set up in Section 1: verifying the FLASH protocol in both an automatic and rigorous way.

Compared with previous work, these easily readable invariants in our work establish "a chain of evidence" for the correctness proof. As we have discussed in Section 5 and 6, these invariants have shed light on the semantics of the FLASH protocol. Both functions of control variables and synchronization of executing rules of different READ-WRITE transactions. In this sense, our understanding is the most profound by using paraVerifier to verify it. In fact, these invariants can be regarded as axiom semantics of FLASH protocol.

In the future, we want to extend our work in the following two directions: (1) We want to automate the CMP-method in [6] by adopting our invariants. We believe that our invariant should be enough to strengthen guards of the rules of the abstract node; (2) We want to relate the predicate-abstraction with our work. By assigning proper predicates, we can construct the abstract version of the protocol, which can be regarded as a specification of the protocol. The original FLASH protocol is regarded as the implementation of the specification. Our invariants should be very useful in proving the refinement between the implementation and specification.

References

1. Kuskin, J., et al.: The Stanford FLASH multiprocessor. In: ISCA'94, pp.302–313. IEEE (1994)

2. Owre, S., Rushby, J.M., , Shankar, N.: PVS: A prototype verification system. In: CADE'92, pp.748–752. Springer (1992)
3. Park, S., Dill, D.L.: Verification of flash cache coherence protocol by aggregation of distributed transactions. In: SPAA'96, pp.288–296. ACM (1996)
4. Mcmillan, K.L.: Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In: CHARME'01, pp.179–195. Springer (2001)
5. McMillan, K.L.: 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: FMCAD'04, pp.382–398. Springer (2004)
7. Talupur, M., Tuttle, M.: Going with the flow: Parameterized verification using message flows. In: FMCAD'08, pp.1–8. IEEE (2008)
8. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaidi, F.: Invariants for finite instances and beyond. In: FMCAD'13, pp.61–68. IEEE (2013)
9. Arons, T., Pnueli, A., Ruah, S., Xu, Y., Zuck, L.: Parameterized verification with automatically computed inductive assertions. In: CAV'01, pp.221–234. (2001)
10. Li, Y., Pang, J., Lv, Y., Fan, D., Cao, S., Duan, K.: Paraverifier: An automatic framework for proving parameterized cache coherence protocols. In: ATVA'15, pp.207–213. Springer (2015)
11. Li, Y., Pang, J: An inductive approach to strand spaces. Formal Asp. Comput. 25(4): 465-501 (2013)
12. Dill, D.: The murphi verification system. In: CAV'96, pp.390–393 (1996)
13. Park, S., Das, S., Dill, D.: Automatic checking of aggregation abstractions through state enumeration. IEEE TCAD **19**(10) 1202–1210 (2000)
14. Li, Y., Duan, K.: Experiments on FLASH protocol. (2016) `http://lcs.ios.ac.cn/~lyj238/flash.html`.
15. Lv, Y., Lin, H., Pan, H.: Computing invariants for parameter abstraction. In: MEMOCODE'07, pp.29–38. IEEE (2007)