An Automatic Parameterized Verification of FLASH Cache Coherence Protocol by Paraverifier

Yongjian Li[1], Kaiqiang Duan[1], and Yi Lv[1]

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 proof product is a formal proof with a set of inductive invariants. On one hand, both invariants are searched automatically and the formal proof is generated automatically in our work, thus tedious human labor can be obviated. On the other hand, the formal proof guarantees the most rigorous correctness of the parameterized verification. Besides, we make efforts to illustrate the semantical intuition behinds these invariants, thus our proof product is not only a certification of correctness, but also a thoroughly 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.* In this work, we will focus on a real-world cache coherence protocol - Stanford FLASH protocol [**?**]. The challenge posed by parameterized verification is that the desired properties should hold in any instance of the parameterized system.

FLASH protocol 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 [**?**] to verify the protocol for arbitrary N nodes [**?**]. 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 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 aggregated protocol and the original one.

McMillan applies methods of compositional model checking [**?**] to the verification of FLASH protocol by using Cadence-SMV [**?**]. 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 FLASH protocol 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 insight into FLASH protocol. 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 are 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 [**?**] 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 [**?**] 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 this 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 to help people.

In [**?**], 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 FLASH protocol by adopting the invariants computed by BRAB.

To sum up, FLASH protocol 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 [**?**], "if the

method works on FLASH protocol, 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 verified the protocol parametrically. However, human guidance still plays a key role in the above successful verification work for FLASH protocol. This fact reveals the weakness of automatic tool in the parameterized verification. Third, further efforts are still needed for clear mechanization. As argued in [**?**], "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 verification of FLASH protocol in a well-known theorem prover. Previous work is too far away from giving a formal proof for FLASH protocol. Even for a moderate case GERMAN protocol [**?**], which is much simpler than FLASH protocol, no formal proof is not available yet. Let alone FLASH protocol.

The aim of this paper is to apply paraVerifier [**?**] to a parameterized verification 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 semantical features of FLASH protocol, and help people to precisely understand the design of FLASH protocol.
- With the help of paraVerifier, a formal proof can be constructed automatically as a product of a parameterized verification of 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 FLASH protocol.

The organization of this work is as follows: Section **??** introduces the background of paraVerifier; Section **??** introduces informally FLASH protocol; Section **??** introduces our FLASH protocol model in Murphi language [**?**]; Section **??** shows our verification experiments in detail. Especially we introduce some specific tactics used in our experiments: oracles of simplified version and hybrid oracles, and distributing oracles. Section **??** tries to explain meanings of inductive invariants with a flow. Section **??** 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 are 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. A novel feature of our work lies in that a so-called 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 $\mathsf{pre}(r)$ and $\mathsf{act}(r)$ be the guard and the action of the rule r, $\mathsf{preCond}(f, \mathsf{act}(r))$ be the weakest precondition to make f to be true after execution of $\mathsf{act}(r)$. The relation $\mathsf{invHoldRule}(s, f, r, fs)$ defines a causality relation between f, r, and fs, which guarantees that if each formula in fs holds before the execution of rule r, then f holds after the execution of rule r. This includes three cases. 1) $\mathsf{invHoldRule}_1(s, f, r)$ means that after rule r is executed, f becomes true immediately; 2) $\mathsf{invHoldRule}_2(s, f, r)$ states that $\mathsf{preCond}(S, f)$ is equivalent to f, which intuitively means that none of the state variables in f is changed, and the execution of statement S does not affect the evaluation of f; 3) $\mathsf{invHoldRule}_3(s, f, r, fs)$ states that there exist another invariant $f' \in fs$ such that the conjunction of the guard of r and f' implies the precondition $\mathsf{preCond}(S, f)$.

With the invHoldRule relation, we define a relation $\mathsf{consistent}(invs, inis, rs)$ between a protocol $(inis, rs)$ and a set of invariants $invs = \{inv_1, \ldots, inv_n\}$.

Definition 1. *We define a relation consistent* :: $formula\ set \times formula\ set \times rule\ set \Rightarrow bool$. $consistent(invs, inis, rs)$ *holds if the following conditions hold:*

1. *for all formulas* $inv \in invs$ *and* $ini \in inis$ *and all states* s, ini *holding at* s *implies* inv *holding at state* s;
2. *for all formulas* $inv \in invs$ *and rules* $r \in rs$ *and all states* s, $\mathsf{invHoldRule}(s, inv, r, invs)$

Then, a so-called consistency lemma is proposed, as shown below:

Lemma 1. *If* $P = (ini, rs)$, $\mathsf{consistent}(invs, ini, rs)$, *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., 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 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. **??**.

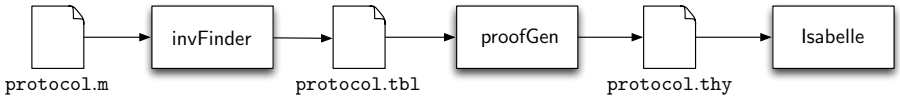

Fig. 1. The workflow of paraVerifier

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 transforms the Murphi file into the 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 parameterized forms, 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

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 cache 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 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. 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 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, only an (in)formal account of FLASH protocol, as shown above, is provided in most previous work. It is still not very clear to most readers. 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. 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 structure, 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 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 of a

set of sub-formulas f_i, denoted as $decompose(f)$, 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 an READ-transaction including a sharing-write back is shown as follows:

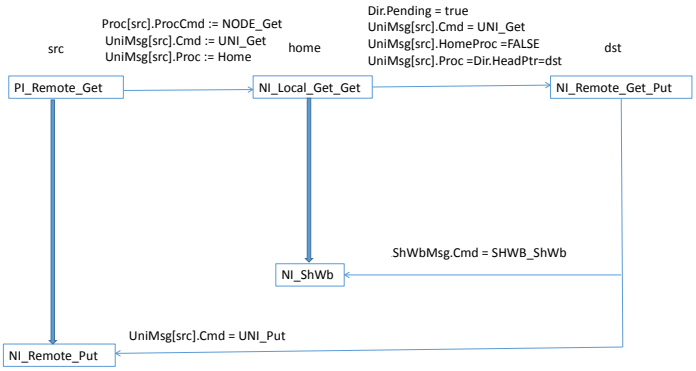

Fig. 2. The flow of READ-transaction including a sharing-write back

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 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 agents' executing rules are illustrated clearly; (2) communications between agents are also illustrated. In section **??** and **??** we combine steps in a flow with auxiliary inductive invariants, by which we can make it clear that how FLASH protocol guarantee the synchronization between different clients. 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 [**?**]. In [**?**], 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 to help user better understanding the system with the generated invariants.

4 Formal Description of FLASH protocol in Murphi

A formal protocol model includes a set of parameterized transition rules and properties. Here we adopt a version of FLASH protocol similar to that in [**?**], which is different from the version in [**?**]. The difference between them lies in the modeling 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 are identical to those of the other non-Home node. In the model of work in [**?**], state variables of the Home node and non-Home nodes are both array-variables. This modeling has a shortage of destroying the symmetry between indices which represents identifiers for agents.

In our modeling, the symmetry between indices should be preserved and will be used in paraVerifier. Namely, some parameterized variable $a[i]$ records some state a of node i. Home is not an explicit index to reference a node in our model. State variable a of the Home node is recorded as a global (or non-parameterized) variable $Homea$. Being consistent with the above philosophy, our FLASH protocol in Murphi is changed accordingly. For instance, we compare the original version in [**?**], 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 one. $HomeHeadPtr$ is added similarly in our model. $HomeShrSet$ and $HomeInvSet$ are added in order to model $ShrSet[Home]$ and $InvSet[Home]$ to be true in the original version. We use the assignment $Sta.UniMsg[src].HomeProc := true$ to model that $Sta.UniMsg[src].Proc$ is the $Home$ node. There are three properties under verification. Our experiment data includes the paraVerifier instance, invariant sets, Isabelle proof scripts [**?**].

```
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 data property.

5 Verifying FLASH protocol by paraVerifer

An obstacle of generating an SMV-oracle and its solution For cache coherence protocols like MESI or German protocols with small scales, paraVerifier is easy to use to verify automatically them. A model written in Murphi language is provided as a formal model, which contains both the protocol and properties under verification. 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 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 protocol with data path is a real world protocol with industry scale. For a FLASH protocol instance configuration size NODENUM=3 and 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 32 Inter Xeon processors, 384 GB memory. In fact, this problem of state explosion is the main obstacle to the applying previous automatic solutions such as [**?**,**?**] to the parameterized verification of FLASH protocol. For instance, a reachable state set of a protocol instance of FLASH protocol is needed in both the work in [**?**,**?**]. 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 a 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 protocol, the above two approaches 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 protocol with data paths, we construct a SMV -model of a simplified version of the FLASH protocol without data paths, 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 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))) can't be judged via the SMV -model of a simplified version of the FLASH protocol 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 is built, and Murphi tool is run to judge the formula containing data variables. Here we adopt an heuristic strategy: if Murphi tool 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 for 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.

Verifying procedure In order to verify FLASH protocol by paraVerifier under Linux system, some simple steps are needed.

1. The Murphi model of FLASH protocol is converted to its corresponding internal model by compiler murphi2ocaml.
2. The specific verifier for FLASH protocol with OCaml is built and the generated executable file supposing that address of the NuSMV oracle was vserv and address of the Murphi oracle was mserv is executed. Afterwards, a directory named n_flash where proof scripts were put will be generated.
3. The proof scripts in Isabelle is then executed.

We run paraVerifier on a client with 4 Intel Xeon processors, 8 GB memory and 64 bit Linux 3.15.10. 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 **??**.

Table 1. Verification Result on FLASH protocol

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

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.

```
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. 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 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 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 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 pend.

Not only auxiliary invariants are searched, but also causal relations between invariants and rules are searched. A fragment on the rule `NI_Remote_PutX` and invariant `inv1= !((Sta.Proc[2].CacheState = cache_e) & (Sta.Proc[1].CacheState = cache_e))` is shown as below:

```
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;

1 rule: 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))
2 rule: 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))
3 rule: NI_Remote_PutX[3]; inv: !((Sta.Proc[2].CacheState = cache_e) & (Sta.Proc[1].CacheState
= cache_e)); g: TRUE; rel: invHoldForRule2
```

- Line 1 specifies that if a state s satisfies the guard of the rule, and `inv4= !((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 `inv1` holds at state s'. Reader can verify this easily. This expresses the intuition behind the relation $invHoldForRule_3$. Line 2 can be analyzed similarly.
- Line 3 that `NI_Remote_PutX[3]` has nothing to do with all the variables in `inv1`, thus s satisfies the formula if and only the post state s' satisfies the formula. This expresses the intuition behind the relation $invHoldForRule_2$.

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 an N-parameterized form, where each index is symbolic such as $i1$ and $i2$ and N is also symbolic. An Isabelle proof script is generated by `proofGen`, where an N-parameterized instance of FLASH protocol is modeled and the properties are formally proved. The Hierarchy of the lemmas is illustrated in Fig. **??**.

```
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) }"
```

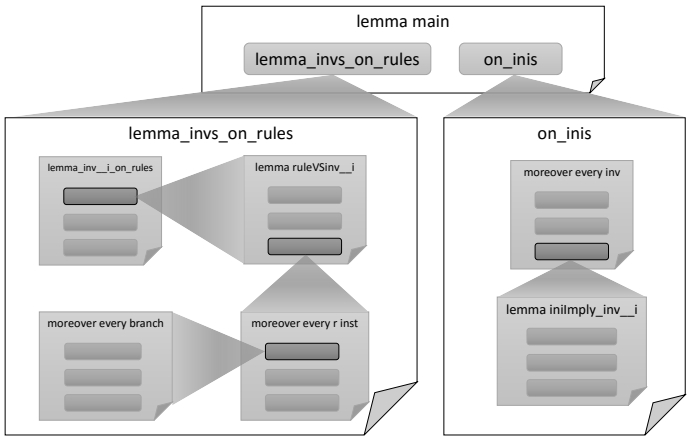

Fig. 3. The Hierachy of the lemmas

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_invName` on a causal relation of a rule and a parameterized invariant.
5 A Lemma such as `rules_invName` on causal relation for all rule and an invariant.
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 formula hods at any reachable state of the N-parameterized FLASH protocol instance.

In order to understand the above proof, we can relate the proof with the aforementioned three causal lines. If we regard the three lines as 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 satisfies the assumption of the lemma, the causal relation between must be sym-

metric to the concrete relations listed in the three lines. Therefore, each proof is an abstraction of a kind of concrete causal relations.

```
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
```

6 Illustrating a Typical Flow by With Invariants

In order to understand state transitions of the flow in Fig. **??** after shwb_shwb is issued, we may observe the invariants containing the condition 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 an 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.
– Meanwhile, 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.
– 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 inv__101 holds for src.
– For other node src' than src, its UniMsg[src'].HomeProc flag is initialized by TRUE, thus inv__101 holds for src'.

```
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))
```

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 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 automatical. The ultimate correctness is gurantteed by a mechanical proof in a theorem prover. Therefore, we achieve

the two goals set up in Section **??**: verifying FLASH protocol in both an automatical 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 **??** and **??**, these invariants have shed light on the semantics of 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 [**?**] by adopting our invariants. We believe that our invariant should be enough to strenthen 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 (1994)
2. Owre, S., Rushby, J.M., , Shankar, N.: PVS: A prototype verification system. In: CADE'92, pp.748–752 (1992)
3. Park, S., Dill, D.L.: Verification of flash cache coherence protocol by aggregation of distributed transactions. In: SPAA'96, pp.288–296 (1996)
4. Mcmillan, K.L.: Parameterized verification of the FLASH cache coherence protocol by compositional model checking. In: CHARME'01, pp.179–195 (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 (2004)
7. Talupur, M., Tuttle, M.: Going with the flow: Parameterized verification using message flows. In: FMCAD'08, pp.1–8 (2008)
8. Conchon, S., Goel, A., Krstic, S., Mebsout, A., Zaidi, F.: Invariants for finite instances and beyond. In: FMCAD'13, pp.61–68 (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 (2015)
11. Dill, D.: The murphi verification system. In: CAV'96, pp.390–393 (1996)
12. Park, S., Das, S., Dill, D.: Automatic checking of aggregation abstractions through state enumeration. IEEE TCAD **19**(10) 1202–1210 (2000)
13. Y. Li, K.d.: Experiments on flash (2016) `http://lcs.ios.ac.cn/~lyj238/flash.html`.
14. Lv, Y., Lin, H., Pan, H.: Computing invariants for parameter abstraction. In: MEMOCODE'07, pp.29–38 (2007)