Tech Report 150427

Implementation of invFinder

Kaiqiang Duan

# Summary

The tool invFinder is to search auxiliary invariants, especially in cache coherence protocols. Besides, it tries to generate some information about consistent relations between the invariants found and the rules.

To run with invFinder, one must model a system first, giving the needed state variables, transition rules and initial invariants. Then, put the initial invariants into a queue, and try to generate new auxiliary invariants and corresponding causal relations by the following cycling until the queue is empty:

1. Fetch an invariant by dequeuing the invariant queue;
2. For each rule, generate a causal relation, and try to find a new auxiliary invariant, if found, mark it as an optional invariant;
3. Enqueue the invariants marked as optional that haven’t been found yet.

The auxiliary invariants and corresponding causal relations are basis of the next step for proving a parameterized system.

# Architecture

See Figure 1.

Paramecium

invFinder Core

Semantics

Formula

3rd Party

Operation

Generalize

ToStr

SMT

SMV

SMT2

SMV

Debug

Figure Architecture of invFinder

1. Paramecium: Fundamental modeling language for invFinder;
2. Semantics: Semantics of the modeling language;
3. invFinder Core: Core program for generating auxiliary invariants and causal relations;
4. Formula: Useful functions for formula, including:
   1. Generalize: Genaralize a ground formula to a parameterized formula with its ground parameters;
   2. Operation: Includes some formula operations, e.g., simplifying, normalizing, symmetric checking, etc.;
   3. 3rd Party: 3rd Party tools, including SMT solver Z3 and model checker SMV;
5. ToStr: Convert a formula to its string representation, e.g., smt2 representation, smv representation, and debug representation for debug information.

# Core Algorithm

For a given protocol, which includes its name, type definitions, variable definitions, init states, rules and properties, we could find the auxiliary invariants and causal relaitons through the following steps:

1. Give model of the protocol, and initial invariants, i.e., property instances;
2. For each property instance, put it into a queue respectively, and for each queue, repeat
   1. If the queue is empty, then return the invariant library and relations;
   2. Else, dequeue the queue and get a property instance, i.e., an invariant *cinv*;
   3. Generate rule instances *crules* corresponding to *cinv* with a policy **rule instantiating policy**;
   4. Search new auxiliary invariants and generate causal relations between *cinv* and *crules*;
   5. Put the new invariants and relations to their library.

## Searching strategy

Given a transition rule *r*, a property *f*, and a property set *F*. Suppose pre *r* is guard of a rule *r*, and act *r* is statements of the rule; preCond *f* (act *r*) is property to be changed to after execution of rule *r*. either of the three relations holds:

1. invHoldForRule1: pre *r* → preCond *f* (act *r*), i.e., after rule *r* is executed, property *f* still holds;
2. invHoldForRule2: *f* = preCond *f* (act *r*), i.e., execution of rule *r* does not change state variables in property *f*;
3. invHoldForRule3: , i.e., there exists an property *f’* in *F* such that conjunction of guard of rule *r* and *f’* implies that *f* holds after the execution of rule *r*.

So, given a rule instance *crule*, a property instance *cinv*, and the property library *invs*, the searching strategy is:

1. Compute preCond *cinv* (act *crule*), then **simplify** it, name it as *obligation*;
2. If *obligation* is symmetric with *cinv*, then invHoldForRule2 holds;
3. Else if pre *crule* → *obligation* is a tautology, then invHoldForRule1 holds;
4. Else, invHoldForRule3 must hold, i.e., there must exist a auxiliary invariant, and we try to search for it with a **choose policy**, which could find a new invariant in either of the four levels:
   1. Level Tautology: the new invariant is a tautology;
   2. Level Implied: the new invariant can be implied by old ones;
   3. Level NewInv: the invariant found is really brand new;
   4. Level NotInv: none of *invs* conjuncted with pre *crule* could imply that *obligation* still holds, but this level should never occur, if occurred, there should be some bugs.

## Choose policy

All policies generate some invariant candidates, and check if they are real invariants with **check level** algorithm.

1. Policy 0: Since changes on global state variables might affect invariants strongly, if the change could imply *obligation*, then a new invariant is generated;
2. Policy 1: If pre *crule* has same parameter references with *obligation*, and global components of pre *crule* could imply *obligation*, then generate a new invariant;
3. Policy 2: Cast pre *crule* to conjuntion normal form, and strength every component of it, if exists one of the strengthen formulae which could imply *obligation*, then get a new invariant;
4. Policy 4: Since invHoldForRule3 must holds, we simply try to remove the unnecessary components in the conjunction, and get a new invariant.

All policies generate some invariant candidates, and check if they are real invariants with **check level** algorithm.

## Check Level Algorithm

Level of a new invariant found could be determined with following strategy:

1. Level Tautology: check the invariant with an SMT solver;
2. Level Implied: a new invariant is in level Implied if any old invariant could imply *obligation*; given an old invariant *old* and *obligation*, check:
   1. If difference of state variables between *old* and *obligation* is not empty, then could not imply;
   2. Else if *old* has no global state variables, then check if *old* → *obligation* is a tautology, if so then could imply else could not;
   3. Else if *obligation* is not **compatible** with *old*, then could not imply;
   4. Else if *obligation* is symmetric with *old*, then could imply;
   5. Else, generalize *old* and instantiate it to more invariant *olds* with compatible parameters corresponding to *obligation*, if any of *olds* could imply *obligation*, then could imply, otherwise could not.
3. Level NewInv: check the invariant with SMV model checker;
4. Level NotInv: if none of the three levels above holds.

## Compatible checking

**Definition**. Suppose parameter type set of invariant *inv1* is *types1*, and *types2* to *inv2*; suppose length of *types1* is *m*, and length of *types2* is *n*; *inv1* is compatible with *inv2* iff

1. *types2* is subset of *types1* (so ), and
2. Suppose parameter sets of each type in *inv1* are *params1*[*i*] for , and in *inv2* are *params2*[*j*] for , then for .

The compatible checking algorithm returns empty list if the two invariants are not compatible, and return the compatible parameters if they are compatible.

# Limitations

There are some limitations in current implementation of invFinder:

1. To check if a formula is a tautology, we use SMT solver Z3, whose input must be a file, which could leed to wasting of CPU time;
2. We recompute the reachable set with SMV model check every time need to check a invariant because of limited work time, which must be reconsidered;
3. We do have an effective instantiation policy for rules in some kinds of systems, but it is hard to find a general one.

# Future Work

1. SMT check: how to check tautology of a formula without write and read files? Optional answer: Z3 ocaml interface.
2. SMV check: develop a master/slave structure for check invariants effectively, even concurrently.
3. A general instantiation policy for rules.
4. Upward extension: extend expression ability of the model language.
5. Downward extension: isabelle support.
6. Tool extension: tool configuration.