# Finding Counter Examples in Induction Proofs

Koen Claessen and Hans Svensson

Chalmers University of Technology, Gothenburg, Sweden
{koen,hanssv}@chalmers.se

**Abstract.** This paper addresses a problem arising in automated proof of invariants of transition systems, for example transition systems modelling distributed programs. Most of the time, the actual properties we want to prove are too weak to hold inductively, and auxiliary invariants need to be introduced. The problem is how to find these extra invariants. We propose a method where we find *minimal counter examples* to candidate invariants by means of *automated random testing* techniques. These counter examples can be inspected by a human user, and used to adapt the set of invariants at hand. We are able to find two different kinds of counter examples, either indicating (1) that the used invariants are too strong (a concrete trace of the system violates at least one of the invariants), or (2) that the used invariants are too weak (a concrete transition of the system does not maintain all invariants). We have developed and evaluated our method in the context of formally verifying an industrial-strength implementation of a fault-tolerant distributed leader election protocol.

## 1 Introduction

This paper gives a partial report on our experiences on using (semi-)automated theorem proving to formally verify safety properties of an industrial-strength implementation of a fault-tolerant leader election protocol in the programming language Erlang [19].

Leader election is a basic technique in distributed systems; a fixed set of processes has to determine a special process, the *leader*, among them. There is one basic safety property of such algorithms ("there should never be more than one leader"), and one basic liveness property ("eventually there should be one leader"). In *fault-tolerant* leader election, processes can die and be restarted at any point in time (during or after the election), making the problem immensely tricky.

Erlang is a language for distributed programming originally developed for implementing telecommunication systems at Ericson [3, 2]. A key feature of the systems for which Erlang was primarily designed is fault-tolerance; Erlang has therefore built-in support for handling failing processes.

The implementation of the leader election algorithm we verified was developed by us, after we had uncovered some subtle bugs in an earlier existing implementation using testing techniques [4]. Our new implementation is based on an adaptation of a standard fault-tolerant leader election algorithm by Stoller [18] and is now a standard library in Erlang. In our implementation, we had to make some changes to Stoller's original algorithm because of the way processes communicate in Erlang (via asynchronous message passing over unbounded channels) and the way fault-tolerance is handled in Erlang (a process can monitor another process, in which case it receives a special message when the other process dies).

From our previous experience, we knew that it is extremely hard to get these kinds of algorithms right. Indeed, we started by extensively testing the new implementation using our testing techniques [4], leading to our increased confidence in the correctness of the implementation. However, we had some reasons to be cautious. Firstly, our implementation was based on an adaptation of Stoller's original algorithm, so even if Stoller's algorithm were correct, our adaptation of it might not be. Secondly, Stoller never gives a formal proof of correctness in his paper [18]. His algorithm is in turn an adaptation of a classical leader election algorithm (called "The Bully Algorithm") by Garcia-Molina, which in turn only has been proven correct in the paper in a very informal way [12]. Stoller claims that his modifications are so minor that giving a new proof is not needed: *"The proofs that the Bully$_{FD}$ Algorithm satisfies SLE1 and SLE2 are very similar to the proofs of Theorems A1 and A2 in [GM82] and are therefore omitted."*

When we decided to formally verify our implementation, we first tried a number of different model checking methods (among others SPIN [13] and our own model checker McErlang [11]). Unfortunately, these could only be used for extremely small and unconvincing bounds on the number of processes, sizes of message queues, and number of times processes can die. This is partially due to the huge state space generated by the combination of asynchronous message passing and fault-tolerance.

The alternative we eventually settled on was to prove invariants of the system inductively by means of automated first-order logic theorem proving. Here, we model the implementation as an abstract transition system, and express the properties we want to prove as invariants on the states of the transition system. The reasons we chose this approach were (1) using first-order logic allowed us to prove the implementation correct for any number of processes, using unbounded message queues and an unbounded number of occurring faults, and (2) automated first-order theorem provers are relatively autonomous, in principle only requiring us to interact with the verification process at the level of choosing the invariants.

The main obstacle in this approach is that, most often, the (relatively small) set of invariants one is interested in establishing is not inductively provable. This means that the original set of invariants has to be strengthened by changing some of the invariants or by augmenting the set with new invariants, until the set is strong enough to be inductive. Very often, this is a non-trivial and labour-intensive task. In our case, we started with one invariant ("there should not be more than one leader") and we ended up with a set of 89 invariants. This is the sense in which we call our method *semi-automated*; if the right set of invariants is picked (manually), the proof is carried out automatically. Thus, the user of the method does not have to carry out proofs, but only has to formulate proof obligations.

The task of finding the right set of invariants is not only non-trivial, but can also be highly frustrating. The reason is that it is very easy for a user, in an attempt to make the set of invariants stronger, to add properties to the set which are in fact not invariants. When certain invariants can not be proven, the first-order theorem provers we use do not in general provide any reason as to why this is the case, leaving the user in the dark about what needs to be done in order to get the proof through.

We identified 4 different reasons for why a failed proof of a given invariant occurs: (1) the invariant is invalid, i.e. there exists a path from the initial state to a state where the invariant is falsified, (2) the invariant is valid, but too weak, i.e. it indeed holds in all reachable states, but it is not maintained by the transition relation, (3) the invariant is valid and is maintained by the transition relation, but the current axiomatization of the background theories is too weak, and (4) the invariant is valid and should be provable, but the theorem prover at hand does not have enough resources to do so.

The remedies for being in each of these cases are very different: For (1), one would have to weaken the invariant at hand; for (2) one would have to strengthen it; for (3) one would have to come up with extra axioms or induction principles; for (4) one would have to wait longer or break the problem up into smaller bits.

Having a concrete counter example to a proof attempt would show the difference between cases (1), (2) and (3). Thus, having a way of finding counter examples would greatly increase the productivity of the proposed verification method. Providing counter models to first-order formulas (or to formulas in more complex logics) is however an undecidable problem.

We have developed two novel methods, based on random property-based testing using the automated testing tool QuickCheck [9], that, by automatically re-using the invariants as test generators and test oracles, can automatically and effectively find counter examples of categories (1) and (2). Finding counter examples of category (3) remains future work.

Establishing inductive invariants is a very common method for verifying software (in particular in object-oriented programs, see for example [5, 21]). We believe that the methods for finding counter examples in this paper can be adapted to other situations than verifying distributed algorithms.

The contributions of this paper are:

– A classification of different categories of counter examples in the process of establishing inductive invariants using a theorem prover
– Two methods for finding two of the most common categories of counter examples based on random testing
– An evaluation of the methods in the context of the verification of an industrial-strength implementation of a leader election protocol

The rest of the paper is organized as follows. The next section explains the method of verification we use in more detail. Section 3 explains the testing techniques we use. Section 4 reports on the results of our method in the verification of the leader election implementation. Section 5 concludes.

## 2 Verification Method

In this section, we describe the basic verification method we use to prove invariants. The method is quite standard; an earlier description of the method in the context of automated first-order logic reasoning tools can be found in [8]. The system under verification and the invariants are described using three components:

- A predicate *Init* describing the initial state,
- A predicate *Inv* describing the invariant,
- A predicate transformer [*Sys*] that abstractly describes one transition of the system.

For the predicate transformers, we borrow notation also used in dynamic logic [5] and the B-method [1, 21]. For a program $S$ and a post-condition $Q$, we write $[S]Q$ to be the weakest pre-condition for $S$ that establishes $Q$ as a post-condition. This in turn means that we can write

$$P \rightarrow [S]\, Q$$

which has the same meaning as the Hoare triple $\{P\}S\{Q\}$; in all states where $P$ holds, making the transition described by $S$ leads to states where $Q$ holds.

The language we use to describe *Sys* is very simple. The three most important constructs are assignments, conditionals, and non-deterministic choice. The definition of predicate transformers we use is completely standard, and we will only briefly discuss the concepts here. For more details, the reader can consult [21]. Here are the definitions for the predicate transformers for assignments, conditionals, and non-deterministic choice, respectively.

$$\begin{aligned} [x := e]\, P &= P\{e/x\} \\ [\mathbf{if}\, Q\, \mathbf{then}\, S\, \mathbf{else}\, T]\, P &= (Q \rightarrow [S]P) \wedge (\neg Q \rightarrow [T]P) \\ [S \mid T]\, P &= [S]P \wedge [T]P \end{aligned}$$

Establishing *Inv* as an invariant amounts to proving the following two statements:

$$\begin{gathered} \mathit{Init} \rightarrow \mathit{Inv} \\ \mathit{Inv} \rightarrow [\mathit{Sys}]\, \mathit{Inv} \end{gathered}$$

In practice, *Inv* is really a conjunction of a number of smaller invariants:

$$\begin{aligned} \mathit{Init} &\rightarrow \mathit{Inv}_1 \wedge \mathit{Inv}_2 \wedge \cdots \wedge \mathit{Inv}_n \\ \mathit{Inv}_1 \wedge \mathit{Inv}_2 \wedge \cdots \wedge \mathit{Inv}_n &\rightarrow [\mathit{Sys}]\, (\mathit{Inv}_1 \wedge \mathit{Inv}_2 \wedge \cdots \wedge \mathit{Inv}_n) \end{aligned}$$

The above two proof obligations are split up into several sub-obligations; for the initial states, we prove, for all $i$, several obligations of the form:

$$Init \rightarrow Inv_i$$

For the transitions, we prove, for all $i$, several obligations of the form:

$$\left( \bigwedge_{j \in P_i} Inv_j \right) \rightarrow [Sys]\, Inv_i$$

So, for each invariant conjunct $Inv_i$, we have a subset of the invariants $P_i$ that we use as a pre-condition for establishing $Inv_i$. Logically, we can use all invariants $Inv_j$ as pre-condition, but in practice the resulting proof obligations would become too large to be manageable by the theorem provers we use. Also, from a proof engineering point of view, it is good to "localize" dependencies, so that when the set of invariants changes, we only have to redo the proofs for the obligations that were involved in the invariants we changed. (Note that the set $P_i$ can actually include the invariant $Inv_i$ itself.)

To simplify the problems as much possible, we also use an aggressive case splitting strategy, in the same way as described in [8]. Thus each of the above proof obligations is proved in many small steps.

In Fig. 1 we show an example of an invariant. The function $\texttt{host}(p)$ returns the host for a given process $p$, the predicate $\mathbf{elem}(m, q)$ is true if a message $m$ is present in a message queue $q$. In this example we have an incoming message queue $\texttt{queue}(h)$ for each host $h$. (This simplification from having a message queue per process is possible since there is only one process alive per host.)

$$\begin{array}{l}
\forall Pid, Pid2.( \\
\quad (\mathbf{elem}(\texttt{m\_Halt}(Pid), \\
\qquad\qquad \texttt{queue}(\texttt{host}(Pid2))) \\
\qquad \rightarrow (\texttt{host}(Pid2) > \texttt{host}(Pid)) \\
\quad ) \\
)
\end{array}$$

The invariant states that `Halt`-messages are only sent to processes with lower priority: If there is a `Halt`-message from *Pid* in the queue of $\texttt{host}(Pid2)$, then *host(Pid2)* is larger than *host(Pid)*. (Hosts with low numbers have high priority.)

**Fig. 1.** Example invariant

## 2.1 Failed Proof Attempts

This paper deals with the problem of what to do when a proof attempt of one of the proof obligations fails. Let us look at what can be the reason for a failed proof attempt when proving the proof obligations related to a particular candidate invariant $Inv_i$. We can identify 4 different reasons:

(1) The candidate invariant $Inv_i$ is not an invariant of the system; there exists a reachable state of the system that falsifies $Inv_i$.

(2) The candidate invariant $Inv_i$ actually is an invariant of the system, but it is not an inductive invariant. This means that there exists an (unreachable) state where all invariants in the pre-condition set $P_i$ of $Inv_i$ are true, but after a transition, $Inv_i$ is not true. This means that the proof obligation for the transition for $Inv_i$ cannot be proven.

(3) The candidate invariant $Inv_i$ actually is an invariant of the system, and it is an inductive invariant. However, our background theory is not strong enough to establish this fact. The background theory contains axioms about message queues, in what order messages arrive, what happens when processes die, etc. If these are not strong enough, the proof obligation for the transition for $Inv_i$ cannot be proven.

(4) The proof obligations are provable, but the theorem prover we use does not have enough resources, and thus a correctness proof cannot be established.

When a proof attempt for a proof obligation fails, it is vital to be able to distinguish between these 4 cases. The remedies in each of these cases are different:

For (1), we have to *weaken* the invariant $Inv_i$, or perhaps remove it from the set of invariants altogether.

For (2), we have to *strengthen* the set of pre-conditions $P_i$. We can do this by strengthening some invariants in $P_i$ (including $Inv_i$ itself), or by adding a new invariant to the set of invariants and to $P_i$.

For (3), we have to *strengthen* the background theory by adding more axioms.

For (4), we have to *simplify* the problem by for example using explicit case-splitting, or perhaps to give the theorem prover more time.

## 2.2 Identifying the Categories

How can we identify which of the cases (1)-(4) we are in? A first-order logic theorem prover does not give any feedback in general when it does not find a proof. Some theorem provers, including the ones we used (Vampire [20], E-prover [16], SPASS [10], and Equinox [7]) do provide feedback in certain cases, for example in the form of a finite-domain counter model or a saturation, but this hardly ever happens in practice.

One observation that we can make is that for cases (1)-(3), there exist counter examples of different kinds to the proof obligations.

For (1), the counter example is a concrete trace from the initial state to the reachable state that falsifies the invariant $Inv_i$.

For (2), the counter example is a concrete state that makes the pre-conditions $P_i$ true, but after one transition the invariant $Inv_i$ does not hold anymore.

For (3), the counter example is a concrete counter model that makes the background theory true but falsifies the proof obligation. This counter model must be a *non-standard* model of the background theory, since the proof obligation is true for every standard model (which is implied by the fact that no concrete counter example of kind (2) exists).

We would like to argue that, if the user were given feedback consisting of (a) the category of counter example above, and (b) the concrete counter example, it would greatly improve productivity in invariant-based verification.

In the next section, we show how we can use techniques from random testing to find counter examples of type (1) and (2) above. We have not solved the problem of how to find counter examples of type (3), which remains future work. (This is an unsolvable problem in general because of the semi-decidability of first-order logic.) Luckily, cases (1) and (2) are most common in practice, because, in our experience, the background theory stabilizes quite quickly after the start of such a project.

We would like to point out a general note on the kind of counter examples we are looking for. Counter examples of type (1) are counter examples in a logic in which we can define transitive closure of the transition relation. This is necessarily a logic that goes beyond first-order logic. This logic for us exists only on the meta-level, since we are merely performing the induction base case and step case with theorem provers that can not reason about induction. Counter examples of type (2) are only counter examples of the induction step (and do not necessarily imply the existence of counter examples of the first kind). In some sense, these can be seen as non-standard counter examples of the logic used in type (1) counter examples. Counter examples of type (3) are also counter examples of the induction step, but they do not follow the intended behavior of our function and predicate symbols, and are therefore non-standard counter examples of the induction step.

## 3 Finding Counter Examples by Random Testing

This section describes the random testing techniques that we used to find concrete counter examples to the proof obligations.

### 3.1 QuickCheck

QuickCheck [9] is a tool for performing specification-based random testing, originally developed for the programming language Haskell. QuickCheck defines a simple executable specification logic, in which universal quantification over a set is implemented as performing random tests using a particular distribution. The distribution is specified by means of providing a test data generator. QuickCheck comes equipped with random generators for basic types (Integers, Booleans, Pairs, Lists, etc) and combinator functions, from which it is fairly easy to build generators for more complex data structures.

When QuickCheck finds a failing test case (a test case that falsifies a property), it tries to *shrink* this test case by successively checking if smaller variants of the original failing test case are still failing cases. When the shrinking process terminates, a (locally) minimal failing test case is presented to the user. The user can provide custom shrinking functions that specify what simplifications should be tried on the failing case. This is a method akin to *delta debugging* [22].

For example, if we find a randomly generated concrete trace which makes an invariant fail, the shrinking function says that we should try removing one step from the trace to see if it is still a counter example. When the shrinking process fails, the trace we produce is minimal in the sense that every step in the trace is needed to make the invariant fail. One should note that it is very valuable to have short counter examples; it drastically reduces the time spent on analyzing and fixing the errors found.

### 3.2 Trace counter examples

A *trace counter example* is a counter example of type (1) in the previous section. We decided to search for trace counter examples in the following manner (this is inspired by 'State Machine Specifications' in [14]). Given a set of participating processes, we can construct an exhaustive list of possible operations (examples of operations could be: process X receives a `Halt`-message, process Y crashes, process Z is started, etc). We constructed a QuickCheck generator that returns a random sequence of operations. To test the invariant we then create the initial state for the system (where all participants are dead and all message queues are empty) and apply the operation sequence. The result is a sequence of states, and in each state we check that the invariant holds.

If a counter example to the invariant is found, shrinking is performed by simply removing some operations. To further shrink a test case we also try to remove one of the participating processes (together with its operations). We illustrate how all of this works with the (trivially incorrect) invariant $\forall Pid.\neg isLeader(Pid)$ (i.e. there is never a leader elected). Formulated in QuickCheck, the property looks as follows:

```
prop_NeverALeader =
  \path -> checkPath leStoller (forAll pid (nott (isLeader pid))) path
```

We use the function `checkPath`, which takes three arguments: a model of an Erlang program (in this case `leStoller`), a first-order formula (the property) and a trace (called `path`), and checks that the given formula is true for all states encountered on the specified path. The QuickCheck property states that the result should be true for all paths. Running QuickCheck yields:

```
*QCTraceCE> quickCheck prop_NeverALeader
*** Failed! Falsifiable (after 3 tests and 3 shrinks):
Path 1 [AcStart 1]
```

The counter example is a path involving one process (indicated by "`Path 1`", and one step where we start that process (indicated by "`AcStart 1`"), and clearly falsifies the property. (The leader election algorithm is such that if there is only a single participant, it is elected immediately when it is started.) This counter example has been shrunk, in 3 shrinking steps, from an initial, much larger, counter example. The steps it went through, removing unnecessary events, in this case were:

```
Path 1 [AcOnMsg 1 AcLdr,AcOnMsg 1 AcDown,AcOnMsg 1 AcAck,AcOnMsg 1 AcHalt,
        AcStart 1,AcStart 1,AcOnMsg 1 AcNormQ,AcPer 1]
Path 1 [AcStart 1,AcStart 1,AcOnMsg 1 AcNormQ,AcPer 1]
Path 1 [AcStart 1,AcStart 1]
Path 1 [AcStart 1]
```

Here, "`AcOnMsg` $p$ $m$" indicates that process $p$ receives a message of type $m$. The different message types ("`AcLdr`", "`AcDown`", "`AcAck`", etc.) are part of the internal details of Stoller's leader election protocol [18] and are not explained here.

Being able to quickly generate locally minimal counter examples to candidate invariants greatly improved our productivity in constructing a correct set of invariants.

### 3.3 Induction step counter examples

*Step counter examples* are counter examples of type (2). To find step counter examples is more challenging. Step counter examples can be expected when the stated invariant holds, but its pre-conditions are too *weak* to be proved. The proof fails in the step case, that is there exists a (**non**-reachable) state $s$ such that the invariant is true in $s$, but false in some state $s'$, such that $s' \in next(s)$. The difference from trace counter examples is that we are now looking for non-reachable states, which are significantly harder to generate in a good way.

Our first, very naive, try was to simply generate completely random states, and check if the proof obligation can be falsified by these. We implemented this strategy by constructing a random generator for states and tried to use QuickCheck in the straightforward way. However, not surprisingly, this fails miserably. The reason is that it is very unlikely for a randomly generated state to fulfill all pre-conditions of the proof obligation for the transition. Other naive approaches, such as enumerating states in some way, do not work either, since the number of different states are unfeasibly large, even with very small bounds on the number of processes and number of messages in message queues.

The usual way to solve this in QuickCheck testing is to make a *custom generator* whose results are very likely to fulfill a certain condition. However, this is completely unpractical to do by hand for an evolving set of about 90 invariants.

Instead, we implemented a *test data generator generator*. Given a first-order formula $\phi$, our generator-generator automatically constructs a random test data generator which generates states that are very likely to fulfill $\phi$. So, instead of manually writing a generator for each invariant $Inv_i$, we use the generator-generator to generate one. We then use the resulting generator in QuickCheck to check that the property holds.

Our generator-generator, given a formula $\phi$, works as follows. Below, we define a process, called `adapt` that, given a formula $\phi$ and a state $s$, modifies $s$ so that it is more likely to make $\phi$ true. The generator first generates a completely random state $s$, and then successively *adapts* $s$ to $\phi$ a number of times. The exact number of times can be given as a parameter.

The `adapt` process works as follows. Given a formula $\phi$ and a state $s$, we do the following:

1 Check if $s$ fulfills $\phi$. If so, then we return $s$.
2 Otherwise, look at the structure of $\phi$.
   - If $\phi$ is a conjunction $\phi_1 \wedge \phi_2$, recursively adapt $s$ to the left-hand conjunct $\phi_1$, and then adapt the result to the right-hand conjunct $\phi_2$.
   - If $\phi$ is a disjunction $\phi_1 \vee \phi_2$, randomly pick a disjunct $\phi_i$, and adapt $s$ to it.
   - If $\phi$ starts with a universal quantifier $\forall x \in S.\psi(x)$, $S$ will be concretely specified by the state $s$. We construct a big explicit conjunction $\bigwedge_{x \in S} \psi(x)$, and adapt $s$ to it.
   - If $\phi$ starts with an existential quantifier $\exists x \in S.\psi(x)$, construct a big explicit disjunction $\bigvee_{x \in S} \psi(x)$, and adapt $s$ to it.
   - If $\phi$ is a negated formula, push the negations inwards and adapt $s$ to the non-negated formula.
   - If $\phi$ is a (possibly negated) atomic formula, change $s$ so that the atomic formula is true, if we know how to (see below). Otherwise, just return $s$.

Quantifiers in $\phi$ always quantify over things occurring in the state $s$, for example the set of all processes, or the set of all processes currently alive, etc. When adapting $s$ to $\phi$, these sets are known, so we can create explicit conjunctions or disjunctions instead of quantifiers.

When randomly picking a disjunct, we let the distribution be dependent on the size of the disjuncts; it is more likely here to pick a large disjunct than a small disjunct. This was added to make the process more fair when dealing with a disjunction of many things (represented as a number of binary disjunctions).

Finally, we have to add cases that adapt a given state $s$ to the atomic formulae. The more cases we add here, the better our adapt function becomes. Here are some examples of atomic formulae occurring in $\phi$, and how we adapt $s$ to them:

– "message queue $q1$ is empty", in this case we change the state $s$ such that $q1$ becomes empty;
– "process $p1$ is not alive", in this case we remove $p1$ from the set of alive processes in $s$;
– "queue $q1$ starts with the message $Halt$", in this case we simply add the message $Halt$ to the queue $q1$.

Note that there is no guarantee that an adapted state satisfies the formula. For example, when adapting to a conjunction, the adaption process of the right-hand conjunct might very well undo the adaption of the left-hand conjunct. It turns out that successively adapting a state to a formula several times increase the likelihood of fulfilling the formula. There is a general trade-off between adapting a few states many times or adapting many states fewer times. The results of our experiments suggest that adapting the same state 4-8 times is preferable (Sect. 4).

The final property we give to QuickCheck looks as follows; remember the problem

$$\left(\bigwedge_{j\in P_i} Inv_j\right) \rightarrow [Sys]\, Inv_i$$

and let `invs` be the left hand side of the implication, `inv` is $Inv_i$ and `applySys` corresponds to the []-operation:

```
prop_StepProofObligation invs inv sys =
  \state ->
    forAll (adapt formula state) $ \state' ->
      checkProperty formula state'
  where formula = and (nott inv' : invs)
        inv'    = applySys sys inv
```

This can be read as: For all states $s$, and for all adaptions $s'$ of that state $s$ to the proof obligation, the proof obligation should hold. The function `adapt` is our implementation of the adapt generator-generator, and `checkProperty` checks if a given formula is true in a given state. Remember that we want to find a counter example state, that is why we try to adapt the state so that the pre-conditions (`invs`) are fulfilled but `inv'` is not.

The experimental results are discussed in the next section.

## 4 Results

In this section we present some results from the usage of search for counter examples in the verification of the leader election algorithm. Since the data comes from only one verification project it might not be statistically convincing, but it should be enough to give some idea of how well the search for counter examples works in practice.

### 4.1 Trace Counter Examples

To illustrate the effectiveness of trace counter examples we first show one particular example. In Fig. 2 we see an invariant $A$ that was added to the set of pre-conditions in order to be able to prove another invariant $B$ (i.e. this was the action taken after a failed proof attempt in category 2, as described in Sect. 2.1). The original invariant $B$ was easily proved after this addition, however we could not prove the new invariant $A$. After several days of failed proof attempts, we managed to (manually) find a counter example. The counter example was really intricate, involving four different nodes and a non-trivial sequence of events.

With this unsatisfying experience in fresh memory, we were eager to try the trace counter example finder on this particular example. The result was very positive, the counter example was quickly found (in the presented run after 170 tests), and we could quickly verify that it was equivalent to the counter example

$\forall Pid, Pid2, Pid3.(($
$((Pid \in \texttt{alive})$
$\wedge \textbf{elem}(\texttt{m\_Down}(Pid2),$
$\texttt{queue}(\texttt{host}(Pid)))$
$\wedge (\texttt{lesser}(\texttt{host}(Pid)) \subseteq$
$(\texttt{down}[\texttt{host}(Pid)] \cup \{\texttt{host}(Pid2)\}))$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_1}))$
$\rightarrow \neg((\texttt{pendack}[\texttt{host}(Pid3)] > \texttt{host}(Pid))$
$\wedge (Pid3 \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid3)] = \texttt{elec\_2}))$
$)$
$)$

Whenever a process (*Pid*) is alive, in the first election phase (elec_1) and it receives a Down-message such that *Pid* has received Down-messages from everyone with higher priority (that is the hosts in the set lesser(host(*Pid*))). Then no other process (here *Pid3*) is alive, in the second election phase and having communicated with *Pid* (i.e. having a pendack value larger than host(*Pid*)).

**Fig. 2.** A broken invariant

```
*** Failed! Falsifiable (after 170 tests and 30 shrinks):
Path 4 [AcStart 2,AcStart 3,AcCrash 2,AcStart 1,AcCrash 1,
        AcOnMsg 3 AcDown,AcStart 2,AcOnMsg 3 AcDown,AcStart 1,AcCrash 1]
```

**Fig. 3.** Trace counter example

that we found manually. The result of the QuickCheck run on this example is presented in Fig. 3.

The counter example consists of a Path value. From this value we can conclude that the counter example involves four processes. We can also see the sequence of operations leading to a state where the invariant is falsified. This sequence contains five process starts (AcStart), three process crashes (AcCrash) and two receives of Down-messages by process number 3 (AcOnMsg). It is interesting to see that the fourth process is never started, and never actually does anything, nevertheless it must be present in order to falsify the invariant (or else the shrinking would have removed it).

**Evaluation of Trace Counter Examples** Although the verification process was complicated, we did not have very many badly specified invariants around to test with. The presented example was the most complicated and in total we had some five or six *real* 'broken' invariants to test with. (All of them produced a counter example.) To further evaluate the trace counter example search in a more structural way, we used a simplistic kind of *mutation testing*. We took each invariant and negated (or if it was already negated, removed the negation) all sub-expressions occurring on the left hand side of an implication. Thereafter we tried the trace counter example search for each of the mutated invariants.

In total we generated 272 mutated invariants. We tried to find a trace counter example for each, and succeeded in 187 cases (where we randomly generated 300 test cases for each invariant). However, we should not expect to find a trace counter example in all cases, since some of the mutated invariants are still true invariants. Manual inspection of 10 of the 85 ($272 - 187 = 85$) failed cases revealed only two cases where we should expect to find a counter example. (A re-run of the two examples with a limit of 1000 generated tests was run, and a counter example was found in both cases.)

### 4.2 Induction Step Counter Examples

To illustrate how the inductive step counter examples could be used we use the invariant presented below as an example. This invariant was actually the last invariant that was added in order to complete the proof of the leader election algorithm. The invariant specifies a characteristic of the acknowledgement messages sent during election.

$$\begin{array}{l}
\forall \mathit{Pid}, \mathit{Pid2}, \mathit{Pid3}.( \\
\quad (((\mathit{Pid2} \neq \mathit{Pid3}) \\
\qquad \wedge\ \mathbf{elem}(\texttt{m\_Ack}(\mathit{Pid}, \mathit{Pid2}), \\
\qquad\qquad\qquad \texttt{queue}(\texttt{host}(\mathit{Pid}))) \\
\qquad \wedge\ (\texttt{host}(\mathit{Pid2}) = \texttt{host}(\mathit{Pid3}))) \\
\qquad\ \rightarrow \neg\mathbf{elem}(\texttt{m\_Ack}(\mathit{Pid}, \mathit{Pid3}), \\
\qquad\qquad\qquad \texttt{queue}(\texttt{host}(\mathit{Pid}))) \\
\quad ) \\
)
\end{array}$$

If *Pid2* and *Pid3* are two different processes at the same host, and an `Ack`-message from *Pid2* to *Pid* is in *Pid*'s queue, then there can not also be an `Ack`-message in the queue of *Pid* sent by *Pid3* to *Pid*.

**Fig. 4.** Invariant for step counter example example

The first proof attempt included invariants 3, 14 and 15 (which are also invariants that specify properties about `Ack`-messages), i.e. we tried to prove $(Inv_3 \wedge Inv_{14} \wedge Inv_{15} \wedge Inv_{89}) \rightarrow [Sys]\, Inv_{89}$. This proof attempt fails, and if we search for an induction step counter example we get the following state:

```
State with 2 processes:
* Alive: {(2,3),(2,5)}
* Pids:  {(2,3)}
[ Process: (1,2)
  Status: norm   Elid: (2,3)  Ldr: 1  Pendack: 2
  Queue: [Ack (1,2) (2,3)]
  Acks: {}   Down: {},

  Process: (2,3)
  Status: wait   Elid: (1,2)  Ldr: 2  Pendack: 2
  Queue: [Halt (1,2)]
  Acks: {}   Down: {}]
```

The system state consists of two sets `alive` (that contains the process identifiers of all processes currently alive) and `pids` (that contains all process identifiers ever used). A process identifier is implemented as a pair of integers. Furthermore, the individual state of each process is also part of the system state. Each process state has a number of algorithm-specific variables (`Status`, `Elid`, etc.), and an incoming message queue.

In the counter example we see that the second process has a `Halt`-message from the first process in its queue at the same time as there is an `Ack`-message in the queue of the first process. That means that in the next step the second process could acknowledge the `Halt`-message, and thus create a state in which the invariant is falsified. Indeed such a situation can not occur, and we actually already had an invariant (with number 84) which stated exactly this. Therefore, if we instead try to prove: $(Inv_3 \wedge Inv_{14} \wedge Inv_{15} \wedge Inv_{84} \wedge Inv_{89}) \longrightarrow [Sys]\, Inv_{89}$ we are successful.

**Evaluation of Step Counter Examples** In the verification of the leader election algorithm we used 89 sub-invariants which were proved according to the scheme

$$(Inv_1 \wedge Inv_2 \wedge \cdots \wedge Inv_k) \longrightarrow [Sys]\, Inv_1$$

Since the automated theorem provers are rather sensitive to the problem size, we put some effort into creating *minimal* left hand sides of the implication. That is, we removed the sub-invariants that were not needed to prove a particular sub-invariant.

Therefore, a simple way to generate evaluation tests for the step counter example search is to remove yet another sub invariant from the left hand side and thus get a problem which in most cases (the minimization was not totally accurate) is too weak to be proved in the step case. Thus, we generate a set of problems like

$$(Inv_1 \wedge Inv_2 \wedge \cdots Inv_{k-1} \wedge Inv_{k+1} \wedge \cdots \wedge Inv_n) \longrightarrow [Sys]\, Inv_1$$

and evaluate the step counter example search on this set of problems.

In this way, the 89 proof obligations were turned into 351 problems to test the step counter example search with. More careful analysis revealed that 30 of the problems were actually still provable, thus leaving 321 test cases. The result of running the step counter example search in QuickCheck with 500 test cases for each problem, and a varying number of adapt rounds, is presented in Fig. 5.

In the figure we see that with only one iteration of adapt we find a counter example for around 75% of the tested problems. By increasing the number adapt rounds, we find a counter example for 97% of the tested problems within 500 test cases.

In reality, case-splitting [8] turned these 321 into 1362 smaller problems of which 524 are provable. The results of running the step counter example search in QuickCheck for each of these smaller problems are presented in Fig. 6. The results are quite similar to the results in the earlier figure.

Our conclusion is that this way of finding counter examples is remarkably effective, especially keeping in mind that the counter example search we presented is a fully automatic and a very cheap method. Running QuickCheck for a failed proof attempt takes only from a few seconds, sometimes up to a few minutes.

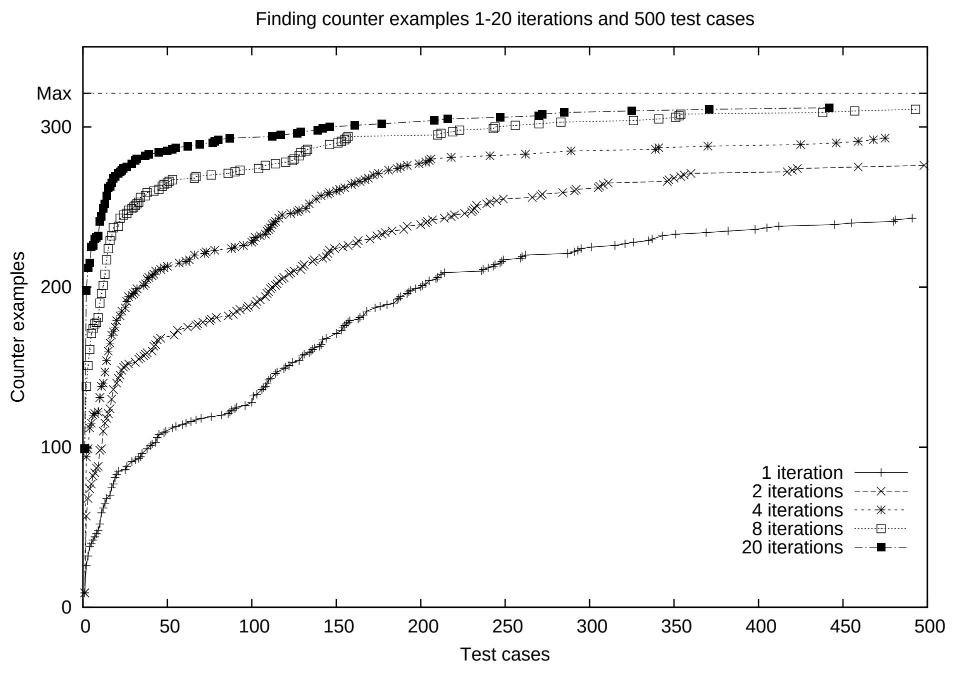

**Fig. 5.** Step counter example results

Another important aspect is the quality of the counter examples; i.e. given an induction step counter example, how hard is it to figure out how to strengthen the invariant to make it provable. Of course this is hard to measure, and any judgement here is highly subjective. We randomly selected some of the found counter examples and inspected them more carefully. In most cases it was easy to find out which sub-invariant to add, which was the original purpose of the method.

Interestingly, in some examples, the counter example indicated that a certain sub-invariant was missing, which was different from the sub-invariant we had removed. (Remember, we generated the tests by removing one sub-invariant from already proved examples.) It turned out that we could actually prove the problem by either using the removed sub-invariant *or* the sub-invariant suggested by the counter example. For example: from the (already proved) problem $(Inv_4 \wedge Inv_7 \wedge Inv_8) \longrightarrow [Sys]Inv_8$ we removed $Inv_4$. This resulted in a counter example, which indicated that adding $Inv_2$ would probably make it possible to prove the sub-invariant. Indeed the problem $(Inv_2 \wedge Inv_7 \wedge Inv_8) \longrightarrow [Sys]\,Inv_8$ could be proved. The reason for this is that $Inv_2$ and $Inv_4$ were partially overlapping. The conclusion must nevertheless be that an induction step counter example is most often very useful.

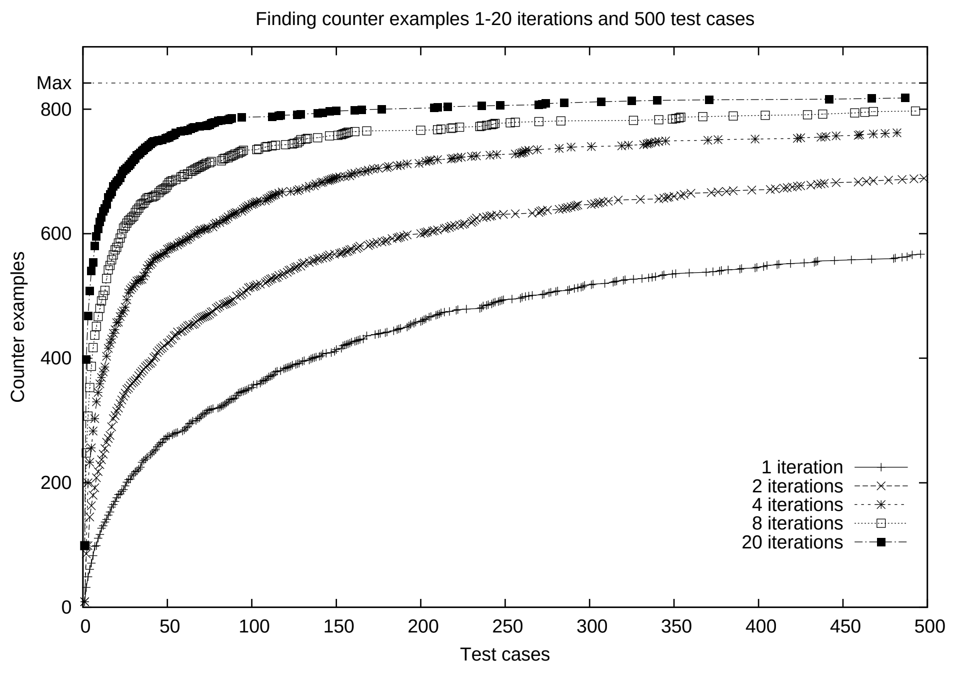

**Fig. 6.** Step counter example results

## 5 Discussion and Conclusion

We have identified different categories of reasons why proof attempts that establish inductive invariants may fail, and developed a method that can identify 2 of these categories by giving feedback in terms of a concrete counter example.

We would like to argue that the results show that this is a useful method; very often counter examples are found when they should be found, and they are easy to understand because of the (local) minimality. The method is also very cheap, once the system is set up, it does not take much time or resources to run 300 random tests. Every time we make changes to the set of invariants, a quick check can be done to make sure no obvious mistakes have been made.

For related work, just like pure first-order logic theorem provers, interactive theorem proving systems usually do not provide feedback in terms of a counter example either. ACL2 [15] provides feedback by producing a log of the failed proof attempt. While sometimes useful, we would like to argue that feedback in terms of counter examples (and in terms of different kinds of counter examples) is more directly useful for a user. In some work in the context of rippling [17], a failed proof attempt is structurally used to come up with an invariant for while-loops in imperative programs.

The interactive higher-order logic reasoning system Isabelle comes with a version of QuickCheck [6]. However, there is no control over generators or shrinking present in this version. The work presented here can possibly be integrated with Isabelle by extending their QuickCheck with the necessary features.

Some might argue that the main problems presented in the paper disappear when moving to a reasoning system that supports induction, for example ACL2 or a higher-order theorem prover. However, in such systems it is still useful to have a notion of different reasons why inductive proofs fail, and the three types of counter examples (1), (2) and (3) are just as useful in such systems.

For future work, we are looking to further reduce the gap between problems where proofs are found and problems where counter examples are found. We are currently working to augment a theorem prover to also give us feedback that can be used to identify categories (3) and (4). For category (3), an approximation of a non-standard counter model is produced, for category (4), the theorem prover can tell why it has not found a proof yet.

Moreover, we want to study liveness more closely, and integrate liveness checking (and finding counter examples) in the overall verification method.

Finally, to increase the applicability of our work, we would like to separate out the different parts of our current system; the counter example finding from the Erlang-specific things, and the leader-election-specific axioms and invariants from the general Erlang axioms.

## References

1. J-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. J. Armstrong. *Programming Erlang – Software for a Concurrent World.* The Pragmatic Programmers, http://books.pragprog.com/titles/jaerlang, 2007.
3. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang.* Prentice-Hall, 1996.
4. Thomas Arts, Koen Claessen, and Hans Svensson. Semi-formal development of a fault-tolerant leader election protocol in Erlang. In *Lecture Notes in Computer Science*, volume Vol. 3395, pages 140 – 154, Feb 2005.
5. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.
6. T. Berghofer, S.; Nipkow. Random testing in isabelle/hol. *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, pages 230–239, 28-30 Sept. 2004.
7. Koen Claessen. Equinox, a new theorem prover for full first-order logic with equality. Presentation at Dagstuhl Seminar 05431 on Deduction and Applications, October 2005.
8. Koen Claessen, Reiner Hähnle, and Johan Mårtensson. Verification of hardware systems with first-order logic. In *Proc. of Problems and Problem Sets Workshop (PaPS)*, 2002.
9. Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.
10. C. Weidenbach et al. SPASS: An automated theorem prover for first-order logic with equality. http://spass.mpi-sb.mpg.de.

11. L-Å. Fredlund and H. Svensson. McErlang: A model checker for a distributed functional programming language. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2007.
12. Hector Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48–59, January 1982.
13. Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, September 2003. ISBN: 0-321-22862-6.
14. John Hughes. QuickCheck testing for fun and profit. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, volume 4354 of *LNCS*, pages 1–32. Springer-Verlag, Berlin Heidelberg, 2007.
15. Matt Kaufmann and J Strother Moore. ACL2 - A Computational Logic / Applicative Common Lisp. http://www.cs.utexas.edu/users/moore/acl2/.
16. Stephan Schulz. The e equational theorem prover. http://eprover.org.
17. Jamie Stark and Andrew Ireland. Invariant discovery via failed proof attempts. In *Proceedings, 8th International Workshop on Logic Based Program Synthesis and Transformation*, 1998.
18. S.D. Stoller. Leader election in distributed systems with crash failures. Technical Report 481, Computer Science Dept., Indiana University, May 1997. Revised July 1997.
19. Hans Svensson and Thomas Arts. A new leader election implementation. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 35–39, New York, NY, USA, 2005. ACM Press.
20. Andrei Voronkov. Vampire. http://www.vampire.fm.
21. J.B. Wordsworth. *Software Engineering with B.* Addison-Wesley, 1996.
22. Andreas Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, New York, NY, USA, 2002. ACM.