Protocol proof checking simplified with SMT

Mark R. Tuttle
Strategic CAD Lab
Intel Corporation
Hudson, MA
tuttle@acm.org

Amit Goel
Strategic CAD Lab
Intel Corporation
Hillsboro, OR
amit1.goel@intel.com

***Abstract*—We believe that recent advances in formal verification are on the verge of making formal verification a viable option for any protocol designer, assuming the designer understands the protocol well enough to explain why it works. We demonstrate this with an SMT-based proof checker developed at Intel called the *Deductive Verification Framework (DVF)*. We show how DVF can be used to prove correct a classical, fault-tolerant, distributed protocol for *consensus*, and describe how a protocol expert starting from scratch, with little-to-no prior familiarity with SMT or DVF, was able to model the protocol and prove it correct in six days and nine pages.**

***Keywords*-Formal methods, model checking, theorem proving, Satisfiability Modulo Theories (SMT), Deductive Verification Framework (DVF), distributed computing, consensus.**

I. Introduction

There is nothing new about using formal methods like model checking and theorem proving to find bugs in protocols. This conference is already familiar with, for example, the impressive work at Oracle [1] to prove correct transactional memory implementations using the theorem prover PVS [2]. At Intel, we have some of the most sophisticated formal verification technology in the industry, and in some domains formal verification has completely replaced simulation as the validation methodology. But nearly every such success story in academia and industry has been the work of highly-trained experts, and the tools are rarely used by the men and women working in the trenches of the product groups. Will formal verification ever become easy enough that, for example, a program committee might consider making formal protocol verification a precondition for accepting a protocol paper? This paper demonstrates that recent advances in proof checking, embodied in a tool developed at Intel, make aspects of formal verification appear viable even for nonspecialists.

Inside Intel, many of us have been pushing model checking as a viable approach to validation and bug hunting in high-level models of company protocols. The push-button nature of model checking [3] makes it the formal approach most likely to be adopted during the design phase. As evangelists of model checking, we have pointed to the technical sophistication required to use a theorem prover, to the length of the proof script, to the deep understanding of how a theorem prover works and the amount of "hand holding" required to get a proof through a theorem prover.

Indeed, model checking protocols at Intel has produced valuable results over the last five to ten years, most notably in the context of cache coherence protocols. Model checkers got a boost from Moore's Law in the last decade that enabled us to check large and sophisticated models. Abstractions like parametric verification [4], [5], [6], [7], [8] let us prove correctness of protocols in arbitrary configurations (for any number of nodes, for example). However, we still find ourselves with models that strain the capacity of these tools, and theorem proving once again looks like the only formal approach that can scale to the size of our problems.

But the world has changed with the advent of powerful SMT engines [9], [10], [11]. *Satisfiability* (SAT) is the problem of finding a satisfying assignment for a Boolean formula. *Satisfiability Modulo Theories* (SMT) is "SAT on steroids." SAT is a decision procedure for the language of Boolean logic. Given a collection of decision procedures for other languages about things like array references and function application, SMT uses SAT to combine these decision procedures to find satisfying assignments for formulas in a much richer language than just Boolean formulas. With the rise of decision procedures and SMT engines, some forms of theorem proving now look like a viable alternative to model checking as an automated approach to formal verification that could be used by product groups.

We demonstrate this viability with the *Deductive Verification Framework* (DVF) [12], an SMT-based proof checker developed at Intel. The key to DVF is a modeling language that makes it easy to describe protocols (in the guarded-command style of TLA+ [13], Murphi [14], Spin [15], and Input-Output Automata [16]); but a language whose expressive power is restricted just enough that the language maps cleanly to the decision procedures of an SMT solver.

We use DVF to prove the correctness of a classical, fault-tolerant, distributed algorithm for *consensus* (also known as *agreement*). In this problem, each process begins with an input bit and chooses an output bit, but processes must agree on the value of this output bit. A process can, however, fail by crashing in the middle of the protocol, confusing other processes with inconsistent information. Consensus looks

simple, but it is provably impossible to solve in realistic models of computation [17].

Our proof was done by a protocol expert (the first author) with no prior familiarly with DVF and little experience with SMT. In three days, we could model the problem, the model of computation, the model of failure, and the protocol. In three more days, we could prove all the key ideas for why the protocol worked. In another three days, we could stitch these ideas into a complete proof. The proof consists of a short list of invariants that can be discharged in ten seconds. The invariants are interesting because all but two are statements about the algorithm itself that go directly to why the protocol works (only two go to working around shortcomings of SMT solvers). The proof is interesting because it is an assertional proof for an algorithm usually proven correct using behavioral (operational) proofs. The whole thing — model, invariants, and proof script — is just nine pages long.

II. Consensus

The *consensus* problem [18] is a fundamental problem in fault-tolerant distributed computation that abstracts the problem of process coordination to its essentials. A distributed *system* consists of n processes or *nodes*. Each node starts with an *input* value, either 0 or 1. The nodes communicate via messages for a while, then each node chooses an *output* value, either 0 or 1. The difficulty is that all nodes must choose the same value to output, even if some nodes fail.

The consensus problem requires that the output values satisfy the following properties:

- *Validity*: Each output value chosen must be the input value of one of the nodes.
- *Agreement*: All output values chosen must be the same.
- *Termination*: All nonfaulty nodes must choose an output value.

Validity rules out the trivial solution in which all nodes choose the output 0 regardless of the actual inputs. Termination says that the nonfaulty nodes are required to make a choice. Agreement says that nodes much reach consensus or agreement on the output value.

One model in which consensus can be solved is a *synchronous message-passing model* with *crash failures*. Our formulation of this model comes from [19], [20]. A computation in this model is a sequence of rounds of message passing. The system consists of a finite collection of *nodes*, each pair of which is connected by a two-way communication link. Nodes share a discrete global clock that starts at time 0 and advances in increments of one. Communication in the system proceeds in a sequence of *rounds*, with round k taking place between time $k-1$ and time k. Each node starts in some *initial state* at time 0. Then, in every following round, each node sends messages to others nodes, then each node receives the messages sent to it during the round, and then each node performs some internal computation such as choosing its output or updating its local state. The nodes follow a *protocol* which specifies exactly what messages each node is required to send during a round (and what internal computation it performs between rounds) as a deterministic function of its local state. Some nodes, however, may be faulty. A nonfaulty node sends every message it is required by the protocol to send. A faulty node sends every message it is required to send until some round k, but in round k sends only some subset of the required messages, and then sends no message in any following round $k' > k$. We say the node *fails* or *crashes* in round k. We assume that at most t nodes can fail in any execution.

```
procedure consensus (node n)
  state := { input }
  for each round r = 1,2,...,t+1 do
    send state to all nodes
    receive states s_1,s_2,...,s_k from other nodes
    state := s_1 ∪ s_2 ∪ ... ∪ s_k
  output := min(state)
```

Figure 1. Consensus protocol

One protocol [21] solving consensus in this synchronous, crash failure model appears in Figure 1. The state of a node is a set of values, initialized to its own input value. Then every round, every node broadcasts its state to all the other nodes, receives the states sent to it, and sets its next state to the union of the states received (a union of sets of input values). After $t+1$ rounds, where t is the number of nodes allowed to fail, every node chooses the minimum value in its state as its output value.

A proof that this protocol solves consensus appears in most distributed algorithms textbooks [22], [23]. The protocol satisfies *validity* because each node chooses its output value from a set of input values. The protocol satisfies *termination* because each node chooses its output at the end of round $t+1$. That the protocol satisfies *agreement*, however, is not so obvious.

The simplest proof of agreement is based on the notion of a clean round. A *clean round* [19] is one in which no nodes crash. The proof proceeds as follows:

1) There is at least one clean round within the first $t+1$ rounds, since at most t nodes can fail and make at most t rounds "unclean."
2) At the end of a clean round, all nonfaulty nodes have the same state, since they all receive the same set of states and all set their next state to the same union of these states.
3) Once all nonfaulty nodes have the same state, they continue to have the same state.
4) At the end of round $t+1$ they all choose the same minimum value from their states as their output values.

This is the proof we want to check with DVF.

III. DVF

The *Deductive Verification Framework* (DVF) is a language and automated proof-checker for the specification and verification of transition systems. In DVF, systems and their properties are modeled in an expressive language that enables a clean and efficient mapping of proof obligations into the multi-sorted first-order logic supported by modern SMT solvers [9], [10], [11]. The language was designed to be clear, and to keep the semantic gap between the DVF and SMT input languages as small as possible to minimize the translation overhead and to use SMT solvers efficiently. This careful language design and the automation from SMT solvers make DVF arguably easier to use than interactive theorem provers, while making it possible for us to verify systems that are out of the reach of model checkers.

Instead of giving a rigorous presentation of DVF — a more complete presentation of the language and proof checker has already been written [12] — we will just present the model and the proof and discuss language features as they arise. Indeed, our experience is that designers have little trouble reading and commenting on a model after a short introduction to DVF.

DVF models a system as a state machine consisting of states and transitions. A state assigns values to a collection of variables of various types. DVF provides the standard primitive types like Booleans, integers, and enums, along with arrays and records and so on. A transition is described as a *guarded command* as is commonly done in distributed computation [22], [23] and in modeling languages [13], [14], [15], [16]. A guarded command consists of a state predicate called a *guard* that is true of a state iff the transition is enabled in that state, and a sequence of statements called a body or *command* that describes how the state is modified by the transition. An execution is an interleaving of transitions: specifically, an execution is a sequence of states such that for every adjacent pair of states s_i and s_{i+1}, there is a transition enabled in s_i that produces s_{i+1}. This should be enough of DVF to get us started.

IV. DVF PROTOCOL MODEL

In the DVF model shown here, we have simplified the consensus protocol with some procedure in-lining to remove some pointless generality in the actual model, and we have omitted a few trivial constant definitions like an empty state (an empty set of input values), but the model given here is essentially what we came up with after a few days of work.

The definition of the consensus problem in Figure 2 begins with type declarations for nodes and values. These types are completely unconstrained. For example, we do not say how many nodes there are in the system or what their names are. Next, we give type declarations for sets of nodes and values. The construct `Set<type node>` is an instance of a template mechanism in DVF that simulates parameterized types. The `Set` template defines a type `t` for a set of values of the parameterized type, some constants like `empty` for the empty set, and some operators (constant functions) on sets like `mem(value,set)` and `cardinality(set)` to test for set membership and compute set cardinality. We then define three mappings that map a node to its input value, output value, and a chosen bit indicating whether the node has actually chosen an output value. (The **`mk_array`** constructor is used for *constant* arrays. For example, **`mk_array`**`[node](`**`false`**`)` is an array mapping all nodes to **`false`**.) We then define the notions of faulty and nonfaulty nodes, which we do by defining two state variables holding two sets of faulty nodes, namely those nodes that failed in prior rounds and those node that are failing in the current round. We then define an integer state variable `round` giving the current round number and initialized to 0, and we define an integer constant `t` giving the bound on the number of nodes that can fail, a constant whose value is constrained only by an axiom stating that its value should be positive. Finally, we can state the validity, agreement, and termination conditions that define consensus.

The crash failure model is given in Figure 3. The key idea here is the notion of a *failure pattern* `snd` for a round, where `snd[n][m]` is true iff sending a message from node n to node m is successful in that round. A failure pattern for a round is consistent with the crash failure model if nodes that have crashed in prior rounds are silent in this round, and nodes that do not send in this round have either crashed in a prior round or are crashing in this round. The types `node_set` and `node_pair_set` are arrays mapping a node or a pair of nodes to a Boolean value indicating presence in the set, and we omit their definition.

The synchronous message-passing model given in Figure 4 is probably the most awkward to read. The first two-thirds of the model are simply stating the fact that an execution begins with an initialization phase, and then in every round nodes first send messages in a send phase, then receive messages in a receive phase, and then perform some internal computation in a compute phase. We define a set of program counters `init_done[n]` indicating that n has initialized, `send_done[n][m]` and `recv_done[n][m]` indicating that the message from n to m has been sent and received, and `comp_done[n]` indicating that n has performed its local computation. After defining a collection of useful predicates, we define two state variables: `state_set[n]` is the state of node n consisting of a set of input values (initially the null set), and `messages[n][m]` is the message (a state, a set of input values, initially the null set) currently in flight from node n to node m in this round. Finally, we define a history variable `clean` used in the proof to keep track of whether the first clean round in the execution happens before, during, or after the current round.

```
type node
type value

module Nodes = Set<type node>
module Values = Set<type value>

const array(node,value) input
var   array(node,value) output
var   array(node,bool)  chosen =
        mk_array[node](false)

var Nodes.t crashed  = Nodes.empty
var Nodes.t crashing = Nodes.empty

def bool noncrashed (node n) =
  ¬Nodes.mem(n,crashed)
def bool noncrashing (node n) =
  ¬Nodes.mem(n,crashing)
def bool nonfaulty (node n) =
  noncrashed(n) ∧ noncrashing(n)

var int round = 0
const int t
axiom positive_bound = t > 0

def bool validity =
  ∀ (node n)
    (chosen[n] ⇒
       ∃ (node m) (output[n] = input[m]))

def bool agreement =
  ∀ (node n, node m)
    (nonfaulty(n) ∧ nonfaulty(m) ∧
     chosen[n] ∧ chosen[m] ⇒
       output[n] = output[m])

def bool termination =
  round > t+1 ⇒
    ∀ (node n) (nonfaulty(n) ⇒chosen[n])
```

Figure 2. Consensus problem

```
type pattern = node_pair_set

const bool is_faulty(node p, pattern snd) =
  ∃ (node q) (¬snd[p][q])

const bool is_silent(node p, pattern snd) =
  ∀ (node q) (¬snd[p][q])

def bool is_crash_pattern (pattern snd) =
  ∀ (node p)
    (Nodes.mem(p,crashed) ⇒ is_silent(p,snd))
  ∧
  ∀ (node p)
    (is_faulty(p,snd) ⇒ Nodes.mem(p,faulty))
```

Figure 3. Crash failure model

```
type phase_state = enum
  {init_phase,
   send_phase, recv_phase, comp_phase}
var phase_state phase = init_phase

var node_set
  init_done = empty_node_set
var node_pair_set
  send_done = empty_node_pair_set
var node_pair_set
  recv_done = empty_node_pair_set
var node_set
  comp_done = empty_node_set

def bool is_init_phase = phase = init_phase
def bool is_send_phase = phase = send_phase
def bool is_recv_phase = phase = recv_phase
def bool is_comp_phase = phase = comp_phase

def bool init_phase_done =
  ∀ (node n) (init_done[n])
def bool send_phase_done =
  ∀ (node n, node m) (send_done[n][m])
def bool recv_phase_done =
  ∀ (node n, node m) (recv_done[n][m])
def bool comp_phase_done =
  ∀ (node n) (comp_done[n])

def bool no_send_done =
  ∀ (node n, node m) (¬send_done[n][m])
def bool no_recv_done =
  ∀ (node n, node m) (¬recv_done[n][m])

def bool round_done =
  is_comp_phase ∧ comp_phase_done
def bool round_start =
  is_send_phase ∧ no_send_done

// node states
type state = Values.t
type state_set = array(node,state)
var state_set global_state = null_state_set

// messages in flight
type message = Values.t
type message_set = array(node,message)
var array(node,message_set) messages =
  mk_array[node](null_message_set)

// this round's failure pattern
var pattern snd = full_node_pair_set

// history variable
type clean_state = enum {before, active, after}
var  clean_state clean = before

def bool before_clean = clean = before
def bool in_clean = clean = active
def bool after_clean = clean = after
def bool round_clean = crashing = Nodes.empty
```

Figure 4. Synchronous message passing model

```
transition initialize (node n)
require (round = 0)
require (is_init_phase)
require (¬init_done[n])
{
  global_state[n] :=
    Values.add(input[n],global_state[n]);
  init_done[n] := true;
}

transition send (node n, node m)
require (is_send_phase)
require (¬send_done[n][m])
{
  messages[n][m] :=
    (snd[n][m]
     ? global_state[n]
     : null_message
    );
  send_done[n][m] := true;
}

transition receive (node n, node m)
require (is_recv_phase)
require (¬recv_done[n][m])
{
  global_state[m] :=
    Values.union(global_state[m],
                 messages[n][m]);
  recv_done[n][m] := true;
}

transition compute (node n)
require (is_comp_phase)
require (¬comp_done[n])
{
  if (round = t+1) {
    output[n] := choose(global_state[n]);
    chosen[n] := true;
  }
  comp_done[n] := true;
}
```

```
transition start_sending ()
require (is_init_phase ∧ init_phase_done ∨
         is_comp_phase ∧ comp_phase_done)
{
  phase     := send_phase;
  send_done := empty_node_pair_set;
  recv_done := empty_node_pair_set;
  comp_done := empty_node_set;

  crashed := Nodes.union(crashed,crashing);

  nondet crashing;
  assume Nodes.disjoint(crashed,crashing);
  assume Nodes.cardinality(crashed) +
         Nodes.cardinality(crashing) ≤ t;

  nondet snd;
  assume is_crash_pattern(snd);

  messages :=
    mk_array[node](null_message_set);
  clean :=
    (clean = before
     ? (¬round_clean ? before : active)
     : after
    );
  round := round + 1;
}

transition start_receiving ()
require (is_send_phase ∧ send_phase_done)
{
  phase := recv_phase;
}

transition start_computing ()
require (is_recv_phase ∧ recv_phase_done)
{
  phase := comp_phase;
}
```

Figure 5. The consensus protocol

Finally, the protocol itself is described in Figure 5 as a set of transitions. Each transition is a guarded command, where the guard is given by a list of preconditions of the form **require** (**formula**), and each **formula** must be true for the guard to be true and the transition to be enabled; and the effect of the transition is given by the sequence of assignments within curly braces. The interesting transitions are `initialize`, `send`, `receive`, and `compute`, and should be self-explanatory given the protocol pseudocode in Figure 1.

The transitions `start_sending`, `start_receiving`, and `start_computing` simply advance the round to the next phase when the prior phase completes. The transition `start_sending` is a bit of an eye-full, however, since it starts the next round and must reset the send, receive, and compute program counters for the round, as well as reset the list of messages in flight and advance the round number. In the middle, however, something interesting is happening: the transition makes nondeterministic choices for the set `crashing` of nodes crashing in this round and for the failure pattern `snd` of messages sent and received in this round. Following each nondeterministic choice, however, is a sequence of **assume** clauses that constrain the choice to reasonable values. In particular, the nodes crashing in this round must not have crashed previously, they cannot push the number of faulty nodes over the maximum number t of nodes that are allowed to fail, and the failure pattern in this round must be consistent with the crash failure model.

V. DVF CORRECTNESS PROOF

Proving protocol correctness requires proving validity, termination, and agreement. Recall that the proofs of validity

and termination are almost obvious; in DVF the proofs of these properties are just thirteen and five lines long, respectively. The interesting part of the proof is agreement: that all nonfaulty nodes choose the same output value at the end of round $t+1$. Recall that the proof sketch for agreement consisted of four steps. We repeat these steps here and give their formulation in DVF.

A *There is a clean round within the first $t+1$ rounds.*

```
def bool clean_round_by_round_tplus1 =
  round ≥ t+1 ⇒ ¬before_clean
```

The predicate `before_clean` means the history variable `clean` says rounds so far have not been clean.

B *All nonfaulty states are equal at the end of a clean round.* First we say what it means for nonfaulty states to be equal

```
def bool equal_noncrashed_states =
  ∀ (node n, node m)
    (noncrashed(n) ∧ noncrashed(m) ⇒
       global_state[n] = global_state[m])
```

and then we prove that once the sending and receiving phases of a clean round are done, the nonfaulty states are equal

```
def bool state_equality_in_clean =
  in_clean ∧
  send_phase_done ∧ recv_phase_done ⇒
     equal_noncrashed_states
```

C *All nonfaulty states remain equal after a clean round.* First we define a notion of message equality

```
def bool equal_nonempty_messages =
  ∀ (node n, node m)
    (messages[n][m] ≠ Values.empty ⇒
       messages[n][m] = global_state[n])
```

and then we prove both state and message equality after a clean round to make the property easier to prove invariant

```
def bool state_equality_after_clean =
  after_clean ⇒
    equal_noncrashed_states ∧
    equal_nonempty_messages
```

D *All nonfaulty nodes choose same output value from their equal states.* That is, that all nonfaulty nodes choose the same output value, and hence satisfy the agreement property:

```
agreement
```

The proofs of these properties in DVF are remarkably short: just 7, 127, 12, and 25 lines long, respectively, assuming normally formatted 80 character lines and no comments.

For example, the proof of A that there is a clean round within the first $t+1$ rounds is simply the following. First, we show that at least one node must fail in every round before the first clean round (because a round in which no node fails is clean):

```
def bool faulty_grows_until_clean_round =
  before_clean ⇒
    Nodes.cardinality(faulty) ≥ round
```

With this, DVF can immediately dispatch two invariants

```
goal faulty1 =
  invariant faulty_grows_until_clean_round
goal faulty2 =
  invariant clean_round_by_round_tplus1
  assuming  faulty_grows_until_clean_round
```

and we are done. Notice, by the way, that here we see our first example of the assume-guarantee style of compositional reasoning supported by DVF: First we prove that a property X is invariant, and then we prove that a property Y is invariant assuming the property X.

The proof of B is obviously the key to the proof, and the outline of B's proof consists of four steps:

B1 Once the sending phase of a round is over, if a nonfaulty node n has a value v in its state, then there must be a message to n containing v (perhaps in the message sent by this nonfaulty node n to itself):

```
def bool state_subset_messages_after_send =
  send_phase_done ⇒
    ∀ (node n)
      (nonfaulty(n)  ⇒
         ∀ (value v)
           (Values.mem(v,global_state[n]) ⇒
              ∃ (node m)
                (Values.mem(v,messages[m][n]))))
```

B2 This message containing v must be sent to all nodes (the sending node did not fail to send to any other node because the round is clean):

```
def bool clean_rounds_broadcast =
  in_clean ∧ send_phase_done ⇒
    ∀ (node n, node m1, node m2)
      (messages[n][m1] = messages[n][m2])
```

B3 Once the receiving phase of a round is over, if a message containing v was sent to node m, then m has v in its state:

```
def bool message_subset_state_after_recv =
  recv_phase_done ⇒
    ∀ (node n, node m)
      (∀ (value v)
         (Values.mem(v,messages[n][m]) ⇒
            Values.mem(v,global_state[m])))
```

B4 So at the end of a clean round (once the sending and receiving phases of the round are done), all nonfaulty nodes have the same global state:

```
def bool state_equality_in_clean =
  in_clean ∧
  send_phase_done ∧ recv_phase_done ⇒
    equal_noncrashed_states
```

The proofs of these steps are algorithmic and short: just 48, 34, 15, and 30 lines long.

But just to push one branch of this proof all the way to the end (and to demonstrate how little the DVF proof differs from the journal proof), consider the third property, B3, that once the send phase of a clean round is over, all nodes have broadcast the same message to all other nodes:

```
def bool clean_rounds_broadcast =
  in_clean ∧ send_phase_done ⇒
    ∀ (node n, node m1, node m2)
      (messages[n][m1] = messages[n][m2])
```

First, we observe that in the failure pattern `snd` for a clean round, a node n sends to all or none of the other nodes:

```
def bool clean_round_delivers_broadcast =
  in_clean ⇒
    ∀ (node n)
      ((∀ (node m) (snd[n][m])) ∨
       (∀ (node m) (¬snd[n][m])))
```

Second, we observe that a message received must have been sent (phrased here in the contrapositive):

```
def bool message_received_was_sent =
  ∀ (node n, node m)
    (¬snd[n][m] ∨ ¬send_done[n][m] ⇒
       messages[n][m] = Values.empty)
```

Third, we observe that in a clean round a node n sends the same message to both $m1$ and $m2$: if n fails to both then the statement is trivially true, and if n sends to both then it sends its local state to both. We must, however, consider explicitly that moment in time when n has sent to $m1$ and not yet to $m2$. This is one of the few places where DVF requires that we be more precise than we would normally be when writing for a journal.

```
def bool nonfaulty_node_broadcasts_message =
 ∀ (node n, node m1, node m2)
  ((¬snd[n][m1] ∧ ¬snd[n][m2] ⇒
      messages[n][m1] = messages[n][m2]) ∧
   (snd[n][m1] ∧ snd[n][m2] ⇒
     (send_done[n][m1] = send_done[n][m2] ⇒
       messages[n][m1] = messages[n][m2]) ∧
     (send_done[n][m1] ∧ ¬send_done[n][m2] ⇒
       messages[n][m1] = global_state[n] ∧
       messages[n][m2] = Values.empty)))
```

DVF can prove that each of these observations is invariant

```
goal cr_del =
  invariant clean_round_delivers_broadcast
goal msg_sent =
  invariant message_received_was_sent
goal nf_bcast =
  invariant nonfaulty_node_broadcasts_message
  assuming message_received_was_sent,
           recv_phase_send_done,
           init_phase_no_send
```

under some trivially invariant assumptions, and that the goal `clean_rounds_broadcast` follows from them

```
goal cr_bcast =
  formula (nonfaulty_node_broadcasts_message ∧
           clean_round_delivers_broadcast
             ⇒ clean_rounds_broadcast)
```

VI. Conclusion

In this paper, we have described our experience using an SMT-based proof checker called DVF to model and prove correct a classical protocol for consensus. The model given looks very much like ordinary code, and the invariants used in the correctness proof are nearly all essential statements needed in any explanation for why the protocol works. The proof itself is mercifully short at only 170 lines, compared to the classical textbook proof that runs to about a page. Most important, the model and proof were completed in just six days by a person comfortable with proof but with little-to-no prior experience with the tool DVF or the underlying SMT technology, and little more than a short tutorial on DVF.

The point of this paper is that formally proving protocols correct is on the verge of becoming easy once you know the informal proof, thanks to recent advances in SMT technology. This does not mean that no work is involved. We all know that when writing up a protocol for a conference, the research is only 10% done until the correctness proof has been written down, because ideas that seem so obvious at the whiteboard usually require some work to nail down. Proof is hard work. But it now appears possible for the author of the proof — with his or her new-found understanding of why the protocol works — to mechanically check that proof with a tool requiring little more than this protocol understanding.

This can be in dramatic contrast to the experience many neophytes have had with powerful theorem provers like PVS [2], HOL [24], or Isabelle [25] when they write things that don't map cleanly to the SMT engines integrated into those provers. It can seem to require achieving a Vulcan mind meld with the implementation to figure out what is going wrong when things go badly. It can be frustrating when a theorem prover cannot make seemingly obvious inferences, and the experience of mechanically checking a proof feels more like spoon feeding a proof to a mechanical checker.

What has been delightful about our experience with DVF, with both the example presented here and applications inside Intel, is that the work remains almost entirely at the algorithmic level. There is virtually nothing in the proof that is there to hold the hand of the inference engine or deal with a pathological behavior of the engine in a special case. Nearly all of the properties included in the proof are logical formulations of ideas you would expect to find expressed in any informally written proof found in a conference paper. In fact, when DVF failed to prove a property invariant, it was almost never helpful to delve into the debugging information more deeply than to get a general impression of where in the protocol the tool was failing. The most effective approach was to think about the algorithm itself, to ignore the kinds of inference the tool might be trying, and to understand algorithmically why the invariant is not invariant.

Of course, DVF is not perfect. The proof checker does not provide much support for proof management that one

finds in modern theorem provers, although this would not be hard to add. For example, one could prove "invariant A assuming B" and to forget to prove B, and DVF would not flag this omission in the compositional reasoning, so the proof structure does need some manual review at the moment. Most important, the proof checker works well when the problem maps cleanly onto the theories understood by the SMT engines that DVF depends on, but we find that this is frequently the case. In spite of these shortcomings, we believe that DVF lies at a valuable sweet spot between the push-button, automated nature of a model checker and the scalable power of a general theorem prover.

Our proof is not perfect, either. Our proof does include one property for DVF to check that is there simply because of the limited support for quantification in the SMT engines DVF depends on. And the one blazing counterexample to our assertion that using DVF did not require spoon feeding obvious facts to the tool is our style of modeling the synchronous round-based communication structure in Figure 4. We devoted many lines to what are essentially program counters describing initialization, sending, receiving, and computation phases of a round, and teaching DVF that they always occur in that order, etc. In hindsight, the model and proof could have been dramatically simplified by using integers $0, 1, 2, 3$ to model phases in place of enums `init_phase`, `send_phase`, `recv_phase`, and `comp_phase`, and using the SMT engine's built-in understanding of integers and how to reason about their order. But that is the kind of nonsense you expect a beginner to write, and yet the proof was still easy to write, and that is the whole point.

ACKNOWLEDGMENT

The design and implementation of DVF was done by the second author with support and assistance from members of Intel's Strategic CAD Lab and Security Research Lab including the first author, Sava Krstić, Rebekah Leslie, Jim Grundy, Murali Talupur, and John O'Leary. We thank Sava, Jim, John, and our referees for their comments on this paper.

REFERENCES

[1] S. Doherty, L. Groves, V. Luchangco, and M. Moir, "Towards formally specifying and verifying transactional memory," *Formal Aspects of Computing*, Mar. 2012, doi:10.1007/s00165-012-0225-8.

[2] S. Owre, N. Shankar, and J. Rushby, "PVS: A prototype verification system," in *Proceedings of the 11th International Conference on Automated Deduction (CADE)*, ser. Lecture Notes in Artificial Intelligence, D. Kapur, Ed., vol. 607. Springer-Verlag, Jun. 1992, pp. 748–752.

[3] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 2001.

[4] K. L. McMillan, "Parameterized verification of the FLASH cache coherence protocol by compositional model checking," in *Proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, ser. Lecture Notes in Computer Science, vol. 2144. Springer, Sep. 2001, pp. 179–195.

[5] C.-T. Chou, P. K. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," in *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2004.

[6] S. Krstic, "Parameterized system verification with guard strengthening and parameter abstraction," in *Automated Verification of Infinite State Systems*, 2005.

[7] M. Talupur and M. Tuttle, "Going with the flow: Parameterized verification using message flows," in *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Nov. 2008, pp. 69–76.

[8] J. O'Leary, M. Talupur, and M. R. Tuttle, "Protocol verification using flows: An industrial experience," in *Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Nov. 2009, pp. 172–179.

[9] C. Barrett and C. Tinelli, "CVC3," in *Proceedings of the 19th International Conference on Computer Aided Verification (CAV)*, 2007, pp. 298–302.

[10] B. Dutertre and L. D. Moura, "The Yices SMT solver," SRI, Tech. Rep., 2006.

[11] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008, pp. 337–340.

[12] A. Goel, S. Krstić, R. Leslie, and M. R. Tuttle, "SMT-based system verification with DVF," in *Proceedings of the 10th Annual International Workshop on Satisfiability Modulo Theories (SMT Workshop)*, Manchester, Jun. 2012.

[13] L. Lamport, *Specifying Systems*. Addison-Wesley, 2002.

[14] C. N. Ip and D. L. Dill, "Better verification through symmetry," in *Proceedings of the 11th International Conference on Computer Hardware Description Languages and their Applications (CHDL)*, 1993, pp. 97–111.

[15] G. Holzman, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[16] N. A. Lynch and M. R. Tuttle, "An introduction to input/output automata," *CWI Quarterly*, vol. 2, no. 3, pp. 219–246, Sep. 1989.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[18] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM*, vol. 27, no. 2, pp. 228–234, 1980.

[19] C. Dwork and Y. Moses, "Knowledge and common knowledge in a byzantine environment: crash failures," *Information and Computation*, vol. 88, no. 2, pp. 156–186, Aug. 1990.

[20] Y. Moses and M. R. Tuttle, "Programming simultaneous actions using common knowledge," *Algorithmica*, vol. 3, no. 1, pp. 121–169, 1988.

[21] D. Dolev and H. R. Strong, "Authenticated algorithms for byzantine agreement," *SIAM Journal of Computation*, vol. 12, no. 4, pp. 656–666, Nov. 1983.

[22] N. Lynch, *Distributed Algorithms*. Morgan Kauffman, 1996.

[23] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[24] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[25] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2283.