SMT-Based System Verification with *DVF*

Amit Goel, Sava Krstić, Rebekah Leslie, Mark R. Tuttle

Intel Corporation

Abstract

We introduce the *Deductive Verification Framework* (*DVF*), a language and a tool for verifying properties of transition systems. The language is procedural and the system transitions are a selected subset of procedures. The type system and built-in operations are consistent with SMT-LIB, as are the multisorted first-order logical formulas that may occur in *DVF* programs as pre- and post-conditions, assumptions, assertions, and goals. A template mechanism allows parametric specification of complex types within the confines of this logic. Verification conditions are generated from specified goals and passed to SMT engine(s). A general assume-guarantee scheme supports a thin layer of interactive proving.

1 Introduction

This paper introduces the *Deductive Verification Framework* (*DVF*), 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 multisorted first-order logic supported by modern SMT solvers. The design of the language 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.

Over the last decade, research and validation groups at *Intel* have performed "high-level modeling" for several interesting examples, where protocols or architectural features were modeled, debugged, and verified in *Murϕ* [9] or *TLA+/TLC* [12]. In both, transitions are conveniently presented as guarded commands. In our own experience, *Murϕ*'s procedural style was most often fitting; however, its set of available types was constricting, and the state space of our models too large for *Murϕ* (or other model checkers) to fully explore.

When more naturally presented with datatypes such as sets, multisets and sequences, our models became more amenable to Hoare-style reasoning, but the hand-proofs of the generated verification conditions were large and ungainly—hardly counting as reliable proofs, despite their mathematical shallowness. Thus, a Hoare-style program analyzer supported by SMT-based proving of the generated conditions seemed to be better suited for our needs. From the class of existing tools of this kind, we picked *Why* [11] to experiment with. It worked on small examples, but was not a perfect fit. *Why*, like other tools in its class, is designed for analysis of traditional programs; our rule-based transition system descriptions required inelegant rewriting to become inputs for *Why*. Additionally, we were stymied by artifacts of the semantic translation(s) used by *Why* to create verification conditions for SMT from the source program and properties described in a polymorphic language.

We designed *DVF* to overcome these difficulties. We strived for clarity and for keeping the semantic gap between our language and SMT-LIB small in order to minimize the translational overhead and to use SMT solvers efficiently. Our support for Hoare-style and assume-guarantee reasoning sets the foundation for further development of interactive proving features. Finally, *DVF* is conceived as a framework that will include additional tools such as simulators and model checkers.

In this paper, we present the language and the tool on simple examples, but we have applied it to verify sizable cache-coherence and consensus protocols as well. In our industrial setting, *DVF* has been successfully used to reason about security properties of instruction-level architectural features. In an ongoing project, *DVF* is also being used to model and verify complex system-on-chip boot flows.

The basic *DVF* language is explained in the following section. The form and use of templates are discussed in Section 3, and Section 4 expounds on proof decomposition in the assume-guarantee style.

2 The *DVF* Language

Listing 1 shows a *DVF* program that models a mutual exclusion algorithm. *DVF* provides a simple procedural programming language for guarded transition systems. Procedures may be annotated with pre- and post-conditions. A *DVF* program can additionally be sprinkled with assertions, assumptions, and goal declarations. All of these formulas are written in a multisorted first-order logic.

Listing 1 A two-process mutual exclusion algorithm

```
type process = enum {n1, n2}
type processes = array(process, bool)

const processes empty =
  mk_array[process](false)

const process other(process n) =
  n=n1 ? n2 : n1

var process turn
var processes want = empty
var processes critical = empty
def bool my_turn(process n) = turn = n

proc unit set_critical(process n, bool v)
ensure (critical[n]=v)
{critical := store(critical, n, v);
 return ();
}

transition req_critical(process n)
require (¬want[n])
{want[n] := true;}

transition enter_critical(process n)
require (want[n])
require (¬critical[n])
require (my_turn(n))
{call set_critical(n,true);}

transition exit_critical (process n)
require (critical[n])
{call set_critical(n,false);
 want[n] := false;
 turn := other(n);
}

def bool mutex =
  ¬(critical[n1] ∧critical[n2])

goal g0 = invariant mutex

def bool aux =
  ∀(process n)
    (critical[n] ⇒my_turn(n))

goal g1 = invariant (mutex ∧aux)
```

The state space of the transition system is defined by the global (state) variables. Initial states of the system are defined by initial values assigned to the state variables. Some procedures in the program are marked as transitions; the system can transition from a state s to a state s' if and only if one of the procedures marked as a transition leads from s to s'. The rest of this section describes the language in more detail.

2.1 Types and Expressions

Booleans, mathematical integers and fixed-size bit-vectors are built-in, with their usual operators [2]. *DVF* also has the unit type, with the single value (), and tuples, with projection operators as in (1,**true**,3)#3 = 3. Arrays parametrized over their domain and range are also supported, with the read operator as in the expression want[n], and the write operator as in **store**(critical,n,v). In addition, arrays come equipped with a **mk_array** constructor: for example, **mk_array**[process m](m=n) can be used to characterize the set containing the single element n, and **mk_array**[process](**false**) is the array mapping all processes to **false**. Since it can represent arbitrary functions as arrays, **mk_array** is not supported by array decision procedures. For compilation to SMT problems, we introduce a fresh constant for each **mk_array** term, and add a quantified assertion to specify its behavior; thus, the term **mk_array**[process](**false**) would generate the assertion ∀(process n)(k[n]=**false**), where k is fresh.

We can also declare enumerations, records, sums, and uninterpreted types. The uninterpreted types are useful for under-specification and for defining new types via axioms that constrain operations on the type. (For example, the type of naturals with the zero constant and the successor operator could be introduced this way.) Records are constructed by simultaneous assignment to all fields or by updating some of the fields of an existing record; record fields are accessed using dot notation. Sums are deconstructed using (exhaustive) match expressions. Since SMT solvers with support for sum types (or, more generally, algebraic data types) typically provide constructors and selectors, we compile pattern matches into variable bindings and expressions using those operators.

Expressions in *DVF* are pure and statically typed. The language is designed to enable simple type inference. In keeping with this desire, enumerations, records and sums are typed nominally. We require annotations wherever it would not be possible to infer the type, say in **mk_array**[process](**false**) to specify the type of the domain.

All of the built-in *DVF* types and operators (with some translation as mentioned above) are supported by SMT solvers such as *CVC3* [3], *Yices* [10] and *Z3* [8].

2.2 Variables, Constants and Definitions

A *DVF* program's global variables define the state of the transition system being modeled. Variable initialization serves to describe the initial state(s) of the system, with the understanding that when the initial value for a variable is not specified, it can be arbitrary. Thus, in Listing 1, want is initialized to empty and turn can initially be either n1 or n2. In addition to global variables, we can use local variables as temporary stores inside procedures and transitions.

The **const** keyword introduces constants and (total) functions. See lines 4-5 and 6-7 in Listing 1 for examples of constant and function declarations. Constants and functions do not need to be defined; much like uninterpreted types, uninterpreted constants and functions are useful for under-specification and for specifying behavior with axioms.

The **def** keyword is used to define state predicates and other functions that take the system state as an implicit argument. Note that the function my_turn defined on line 13 of Listing 1 uses the state variable turn in its definition without having it as an explicit argument. We treat **def**s as macros to be inlined at points of use.

2.3 Statements, Procedures and Transitions

DVF offers a standard set of statements: assignments, conditionals, while loops (annotated with loop invariants), procedure calls and returns. Statements may be grouped into blocks with local variables. The examples

```
turn := other(n);     want[n] := true;     cache[i].c_state := invalid;     x[31:16] := y[15:0];
```

show assignments to a variable, a single array element, individual record field, and a bit-range in a bit-vector. The last three forms are syntactic sugar; for example want[n] := **true** abbreviates want := **store**(want,n,**true**). Parallel assignment via pattern matching on tuples is allowed, as in (x,y) := (y,x). In common with languages like Boogie [14], we have assert, assume and non-deterministic assignment statements.

Procedures take a (possibly empty) list of typed arguments and have a return type; when the return type is omitted, it is assumed to be **unit**. All paths in a procedure must have a **return** statement unless the return type is unit, in which case a **return** () is implicitly assumed. The keywords **require** and **ensure** are used to annotate procedures with preconditions and postconditions. In postconditions, **result** refers to the procedure's returned value.

Transitions are procedures that are identified as such. The system executes transitions atomically, choosing non-deterministically from transitions whose requirements are satisfied by the current state for some choice of argument values. A system can transition from the state s to the state s' if and only if one of the procedures marked as a transition is enabled in s for some values of its arguments, and when executed with these values, it terminates in s'.

2.4 Goals

A program in *DVF* entails a set of proof obligations, arising from the use of **assert** statements, **require** and **ensure** clauses, as well as from explicitly specified goals, as described below.

We can ask *DVF* to prove the logical validity of formulas. State variables may occur in such formulas, in which case the validity of the formula establishes the property for *all* states, reachable or not. For example, a lemma in our proof of the German cache-coherence protocol from [5] is stated as follows.[1]

```
def bool coherence_thm =
  ∀(node i, node j)
    (i≠j ∧cache[i].c_state = exclusive ⇒cache[j].c_state = invalid)

def bool coherence_cor =
  ∀(node i, node j)
    (i≠j ∧cache[i].c_state=shared ⇒cache[j].c_state = invalid ∨cache[j].c_state = shared)

goal thm_implies_cor = formula (coherence_thm ⇒coherence_cor)
```

For a procedure or transition τ with arguments $a_1, \ldots, a_n$, we can write Hoare triples $\{\theta\}\tau(e_1, \ldots, e_n)\{\phi\}$, where the triple stands for $\theta \Rightarrow \mathrm{wlp}(\tau[e_1/a_1, \ldots, e_n/a_n], \phi)$ and wlp is the weakest-liberal precondition. For example, memory integrity could be specified with the three triples:

```
goal wr1 = {true} write(i,v) {mem[i]=v}
goal wr2 = {mem[i]=v ∧i≠j} write(j, v') {mem[i]=v}
goal rd = {mem[i]=v} read(i) {result=v}
```

In triples, the arguments to the procedure may be omitted, in which case the condition is checked for all possible argument values.

[1] The proof needs the fact that caches can be in one of three states, invalid, shared or exclusive. See Appendix A for the *DVF* encoding of the protocol and proof of coherence_thm.

The *DVF* goal **initially** ϕ states that ϕ is true in all initial states, and **invariant** ϕ states that ϕ is true in all reachable states. *DVF* attempts to prove invariants inductively. Thus, proving the goal g0 on line 41 of Listing 1 reduces to the proof obligations **initially** mutex, {mutex}req_critical{mutex}, {mutex}enter_critical{mutex}, and {mutex}exit_critical{mutex}. Often, the invariants we want to prove are not inductive; in our example, {mutex}enter_critical{mutex} is not valid. In such cases we interactively strengthen the invariant until it does become inductive. Strengthening mutex with an auxiliary property aux (line 42) creates mutex ∧aux, which is inductive and goal g1 in Listing 1 is easily discharged. In Section 4, we will describe the support *DVF* provides for managing invariance proofs for larger systems.

3 Axioms and Parametrized Templates

Consider modeling a reference counting system, where there is a set of resources (with a special null resource) and a set of processes, each with a pointer to a resource. The system may allocate a free resource, at which point the resource becomes valid. A valid resource may be referenced by a process whose pointer is then set to the resource, and the resource's reference counter goes up by 1. Processes may subsequently dereference the resource they point to, at the same time decrementing the counter for the resource. Finally, a resource with a count of 0 may be freed by the system, setting its status to invalid. A *DVF* model of this system is shown in Listing 2.

Listing 2 Reference counting

```
type process
type resource
const resource null

var array(resource, int) count
var array(resource,bool) valid =
    mk_array[resource](false)
var array(process, resource) ptr =
    mk_array[process](null)

transition alloc(resource r)
require (r ≠null)
require (¬valid[r])
{valid[r] := true;
 count[r] := 0;
}

transition ref(process p, resource r)
require (valid[r])
require (ptr[p] = null)
{ptr[p] := r;
 count[r] := count[r] + 1;
}

transition deref(process p)
require (ptr[p] ≠null)
{var resource r = ptr[p];
 ptr[p] := null;
 count[r] := count[r] − 1;
}

transition free(resource r)
require (valid[r])
require (count[r] = 0)
{valid[r] := false;}

def bool prop =
  ∀(process p)
    (ptr[p]≠null ⇒valid[ptr[p]])

def bool refs_non_zero =
  ∀(process p)
    (ptr[p]≠null ⇒count[ptr[p]] > 0)

goal g0 = invariant prop

goal g1 = invariant (prop ∧refs_non_zero)
```

We want to verify a basic property: *if a process points to a non-null resource, then that resource must be valid.*

It turns out that this property is not inductive; in particular, one cannot prove the goal {prop}free{prop}. Insight into the failing proof leads to the consideration of an auxiliary property, refs_non_zero, which states that the reference count of a resource cannot be 0 if there is a process pointing to it. Attempting to prove the strengthened invariant (the conjunction of the two properties), we soon realize that it would be convenient to track the set of processes pointing to a resource, and to show that the cardinality of this set is equal to the reference count of the resource.

DVF does not provide a built-in type for sets. We could represent sets using arrays that map elements to Booleans (e.g., valid in Listing 2), but that would still leave us wanting for a cardinality operator. Instead, we introduce a new type for sets, as well as constants for the empty set, set membership, adding and removing an element from a set, and cardinality. We then write axioms that constrain the behavior of these constants. At this point, crucially, we wish for some sort of parametricity to avoid having to encode sets for each new element type.

Typically, and perhaps most naturally, parametricity is introduced via polymorphic type constructors in the language. However, this creates a discrepancy with the non-polymorphic logic of SMT solvers and would necessitate a logical translation between the front and the back end of the tool. This complication is avoided in *DVF* by the use of a template construct. Listing 3 shows the encoding of sets, parametrized by the type of elements, that we use in the next section to complete the proof of our reference counting example. In general, *DVF* templates may be parametrized by types and values; for instance, a template for bounded queues might include the type of elements and the depth of the queue as parameters. Templates are instantiated by modules where all parameters are replaced with concrete types and values. Thus, every *DVF* system description is essentially monomorphic, matching the SMT-LIB logic.

Listing 3 Sets with cardinality

```
template <type elem> Set
{
  // Signature
  type set
  const set empty
  const bool mem (elem x, set s)
  const set add (elem x, set s)
  const set del (elem x, set s)
  const int card (set s)

  // Axioms for set membership
  axiom mem_empty =
    ∀(elem e)
      (¬mem(e,empty))

  axiom mem_add =
    ∀(elem x, elem y, set s)
      (mem(x,add(y,s)) = (x=y ∨mem(x,s)))

  axiom mem_del =
    ∀(elem x, elem y, set s)
      (mem(x,del(y,s)) = (x≠y ∧mem(x,s)))

  // Axioms for cardinality
  axiom card_empty =
    card(empty) = 0

  axiom card_zero =
    ∀(set s)
      (card(s)=0 ⇒s = empty)

  axiom card_non_negative =
    ∀(set s)
      (card(s) ≥0)

  axiom card_add =
    ∀(elem x, set s)
      (card(add(x,s)) =
        (mem(x,s) ? card(s) : card(s)+1))

  axiom card_del =
    ∀(elem x, set s)
      (card(del(x,s)) =
        (mem(x,s) ? card(s)−1 : card(s)))
}
```

4 Compositional Reasoning

Continuing with our reference counting example, we modify the system with an auxiliary variable, handles, that tracks the set of processes pointing to each resource. This allows us to state the property count_eq_card, asserting that the reference count of a resource is equal to the cardinality of its set of handles. Unfortunately, even with this strengthening we do not have an inductive invariant. We need two more properties of the system, ptr_in_handles and handle_is_ptr, to establish that the set of handles for a resource is precisely the set of processes pointing to the resource. Listing 4 shows the system with all these additions.

This process of incrementally strengthening the invariant can quickly become unmanageable as more and more conjuncts are added. The proof goals become harder for the SMT solver and in case of proof failure, it is harder to diagnose the reason for the failure. Also, it is easy to lose track of an understanding for why the auxiliary invariants were necessary. Proofs of this kind are more easily carried out compositionally by breaking down the temporal induction step into separate proofs for each invariant, where each invariant is derived assuming: (1) some of the other invariants that have been proven *for the previous time value*, and (2) yet others that have been proven to hold *now*. This leads to potentially invalid circular reasoning. The following paragraph summarizes a well-known sound compositional reasoning scheme.

Compositional proofs are conveniently represented in graphical form as in Figure 1, with purported invariants at the nodes, and with some edges marked as *latched*. For every node ϕ, let A_ϕ be the conjunction of all ψ such that there is a latched edge from ψ to ϕ. Similarly, let B_ϕ be the conjunction of all ψ such that there is an unlatched edge from ψ to ϕ. The set of proof obligations generated by the graph consists of conditions **initially** ϕ and triples $\{A_\phi \wedge \phi\}\tau\{B_\phi \Rightarrow \phi\}$, for every node ϕ and every transition τ. *If the graph contains no combinational cycles, then this set of initial conditions and triples implies invariance of all node formulas.* This is a special case of McMillan's result on assume-guarantee reasoning [15].[2]

To finish with our example, the goals main and aux1−aux4 in Listing 4 correspond to the five nodes in Figure 1, and are all provable. Notice that all edges coming into any particular node are of the same type: latched or non-latched. Restricting proof graphs so that all nodes are one of these two "pure" types does not lose generality.[3] The "latched" nodes correspond to *DVF* goals of the form **invariant** ϕ **assuming** A_ϕ, which creates proof obligations **initially** ϕ and $\{A_\phi \wedge \phi\}\tau\{\phi\}$. The "unlatched" nodes correspond to **formula** $(B_\phi \Rightarrow \phi)$, which is stronger than $\{\phi\}\tau\{B_\phi \Rightarrow \phi\}$, but is simpler and suffices in practice.

5 Related Work

As a procedural system description language, *DVF* is most closely related to *Murϕ* [9] which is designed for efficient explicit-state model checking. *DVF*, on the other hand, is an SMT-based deductive verification tool. Like *DVF*, the system verification tools *UCLID* [4] and *SAL* [17] are SMT based. *UCLID*'s modeling language is restricted by the decision procedure it relies on (for a logic that covers counter arithmetic, non-extensional arrays, and uninterpreted functions). With *SAL* we share reliance on full-blown SMT solving, but differ in the presentation of systems: *SAL*'s (and *UCLID*'s) system descriptions are relational, not procedural. While

[2] McMillan's technique applies to proving more general temporal formulas, not only invariants.

[3] *Proof.* If, for some ϕ, both A_ϕ and B_ϕ are non-empty, let ϕ' be the formula $B_\phi \Rightarrow \phi$ and modify the proof graph by adding the node ϕ' to it with: (1) a latched edge from ϕ to ϕ'; (2) an unlatched edge from ϕ' to ϕ; and (3) all latched edges into ϕ redirected to end in ϕ'. In the new graph, A_ϕ and $B_{\phi'}$ are empty. The proof obligations for ϕ' in the new graph are weaker than the proof obligations for ϕ in the old graph, and ϕ follows by modus ponens from its predecessors in the new graph.

Listing 4 Reference Counting with a Compositional Proof

```
type resource
type process
const resource null
module R = Set<type resource>
module S = Set<type process>

var R.set valid = R.empty
var array(resource, int) count
var array(process, resource) ptr =
    mk_array[process](null)
var array(resource, S.set) handles

transition alloc(resource r)
require (r ≠null)
require (¬R.mem(r,valid))
{valid := R.add(r,valid);
 count[r] := 0;
 handles[r] := S.empty;
}

transition ref(process p, resource r)
require (R.mem(r,valid))
require (ptr[p] = null)
{ptr[p] := r;
 count[r] := count[r] + 1;
 handles[r] := S.add(p, handles[r]);
}

transition deref(process p)
require (ptr[p] ≠null)
{var resource r = ptr[p];
 ptr[p] := null;
 count[r] := count[r] − 1;
 handles[r] := S.del(p, handles[r]);
}

transition free(resource r)
require (R.mem(r,valid))
require (count[r] = 0)
{valid := R.del(r, valid);}
def bool prop =
  ∀(process p)
    (ptr[p]≠null ⇒R.mem(ptr[p],valid))

def bool refs_non_zero =
  ∀(process p)
    (ptr[p]≠null ⇒count[ptr[p]] > 0)

def bool count_eq_card =
  ∀(resource r)
    (r≠null ∧R.mem(r,valid)
     ⇒count[r] = S.card(handles[r]))

def bool ptr_in_handles =
  ∀(process p)
    (ptr[p]≠null
     ⇒S.mem(p, handles[ptr[p]]))

def bool handle_is_ptr =
  ∀(process p, resource r)
    (r≠null ∧R.mem(r,valid)
     ∧S.mem(p, handles[r])
     ⇒ptr[p] = r)

goal main = invariant prop
            assuming refs_non_zero

goal aux1 = formula (count_eq_card
                     ∧prop
                     ∧ptr_in_handles
                     ⇒refs_non_zero)

goal aux2 = invariant count_eq_card
            assuming ptr_in_handles,
                     handle_is_ptr

goal aux3 = invariant ptr_in_handles
            assuming prop

goal aux4 = invariant handle_is_ptr
```

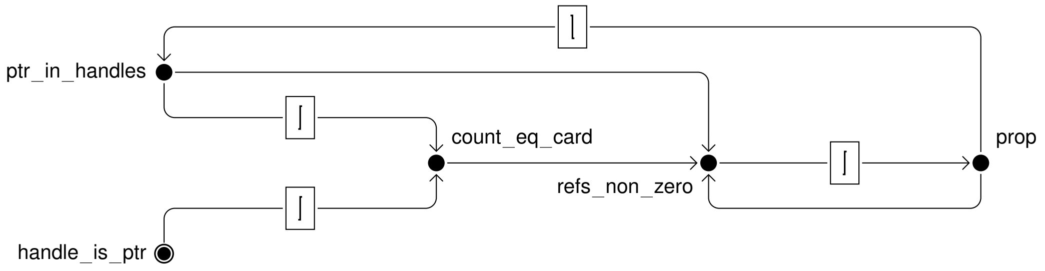

Figure 1: Proof Graph for Reference Counting Example

SAL has features (notably, system composition) currently not available in *DVF*, it does not have the expressivity achieved in *DVF* by axiomatizations and parametrized templates.

DVF uses Hoare-style verification techniques much like existing tools for program verification; for example, [1, 6, 13, 18]. However, these tools are not well suited for high-level modeling because of the limitations of their source languages. A better comparison is with the intermediate languages used by these tools. *Boogie* [14] and *Why* [11] have additional constructs (a richer set of types, polymorphism, the ability to add axioms) that make modeling easier, but they do not provide the support necessary for specification of systems and system properties. In fact, our work on *DVF* was motivated by the promise evident from our early experiments with *Why*, and was directed to overcome the limitations we faced. We found the threading of system invariants via contracts to be a major burden. We also found that the translations introduced to encode types from the polymorphic source language into terms in the multisorted language of SMT solvers made it harder to discharge the proofs. While there has been work [7, 14] on improving translations for subsequent proving efficiency, the parametricity enabled by *DVF* templates is a semantically cleaner solution. *DVF* is less expressive; for example, some aspects of memory encoding in *Boogie* would be difficult to mimic in *DVF*. In our experience this trade-off has been justified—we have found templates to work well for our system modeling tasks. Finally, we note that *DVF* templates are similar to parametrized theories in the interactive prover *PVS* [19].

6 Conclusion

We have presented *DVF*, a language and tool for the modeling and verification of transition systems. It is particularly suitable for systems that involve complex parametrized data types (like sets, multisets, sequences, queues, partial orders). Such types are represented axiomatically by means of a template construction, which allows all logical manipulations in *DVF*—from program annotation to the generation of verification conditions for SMT solvers—to cleanly stay within the multisorted first-order logic.

The niche of *DVF* is systems beyond the capacity of model checkers and not readily admitting model-checkable finite-state abstractions as in, say, [16, 5]. Its primary application so far has been to verify new and complex architectural features in hardware systems where Hoare-style reasoning has scaled well, requiring only seconds for verification tasks that generated up to a few hundred proof obligations. The human effort required to provide auxiliary proof artifacts has not been overwhelming. In fact, the auxiliary invariants have helped our and the designers' understanding, providing valuable documentation as well as checks for implementations.

Ignoring the logical matter, a *DVF* system description looks like an ordinary program with an interface defined by several named procedures (transitions). This syntactic closeness to common programming languages is a significant convenience in the industrial environment, where model development is done in collaboration with designers unfamiliar with formal languages.

For future work, the diagnosis of proof failures is a priority; proof failures have been largely due to unprovable goals (rather than solvers' capacity limits), so we need meaningful counter-model production for our SMT queries. At the language level, we will need to add some modularity mechanism for modeling larger and more complex systems. Human-assisted skeletal proof construction (Section 4) can use additional proof rules and "tactics" borrowed from interactive provers. Finally, even the least sophisticated model-checking back-end would help with early debugging of system models.

References

[1] M. Barnett, R. DeLine, M. Fähndrich, B. Jacobs, K. R. M. Leino, W. Schulte, and H. Venter. The Spec# programming system: Challenges and directions. In *VSTTE*, pages 144–152, 2005.

[2] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *SMT*, 2010.

[3] C. Barrett and C. Tinelli. CVC3. In *CAV*, pages 298–302, 2007.

[4] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *CAV*, pages 78–92, 2002.

[5] C.-T. Chou, P. K. Mannava, and S. Park. A simple method for parameterized verification of cache coherence protocols. In *FMCAD*, pages 382–398, 2004.

[6] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent c. In *TPHOLs*, pages 23–42, 2009.

[7] J.-F. Couchot and S. Lescuyer. Handling polymorphism in automated deduction. In *CADE*, pages 263–278, 2007.

[8] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[9] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *ICCD*, pages 522–525, 1992.

[10] B. Dutertre and L. D. Moura. The Yices SMT solver. Technical report, SRI, 2006.

[11] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, pages 173–177, 2007.

[12] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[13] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR (Dakar)*, pages 348–370, 2010.

[14] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *TACAS*, pages 312–327, 2010.

[15] K. L. McMillan. Circular compositional reasoning about liveness. In *CHARME*, pages 342–345, 1999.

[16] K. L. McMillan. Verification of infinite state systems by compositional model checking. In *CHARME*, pages 219–234, 1999.

[17] L. D. Moura, S. Owre, and N. Shankar. The SAL language manual. Technical report, SRI, 2003.

[18] Y. Moy and C. Marché. *Jessie Plugin,* Boron *version.* INRIA, 2010. `http://frama-c.com/jessie/jessie-tutorial.pdf`.

[19] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In *CADE*, pages 748–752, 1992.

A Verification of the German Protocol

Listing 5 shows a *DVF* translation of the German cache coherence protocol from its *Murϕ* presentation in [5]. We have added an auxiliary variable, exnode, to store the node, if any, that has exclusive access. Figure 2 shows the proof graph for the control property using the following auxiliary invariants:

```
def bool p1 = ∀(node n) (exgntd ⇒shrset[n] = (n = exnode))

def bool p2 = ∀(node n) (chan2[n].m_cmd = gnte ∨cache[n].c_state=exclusive ⇒exgntd)

def bool p3 =
  ∀(node n)
    (¬shrset[n] ⇒cache[n].c_state = invalid ∧chan2[n].m_cmd = empty ∧chan3[n].m_cmd = empty)

def bool p4 = ∀(node n) (invset[n] ⇒shrset[n])

def bool p5 =
  ∀(node n)
    (chan3[n].m_cmd = invack ⇒chan2[n].m_cmd = empty ∧cache[n].c_state = invalid)

def bool p6 =
  ∀(node n)
    (chan2[n].m_cmd = inv ∨chan3[n].m_cmd = invack ⇒¬invset[n] ∧shrset[n])

def bool p7 =
  ∀(node n)
    (chan2[n].m_cmd = inv ∨chan3[n].m_cmd = invack ⇒(curcmd = reqs ∧exgntd) ∨curcmd = reqe)
```

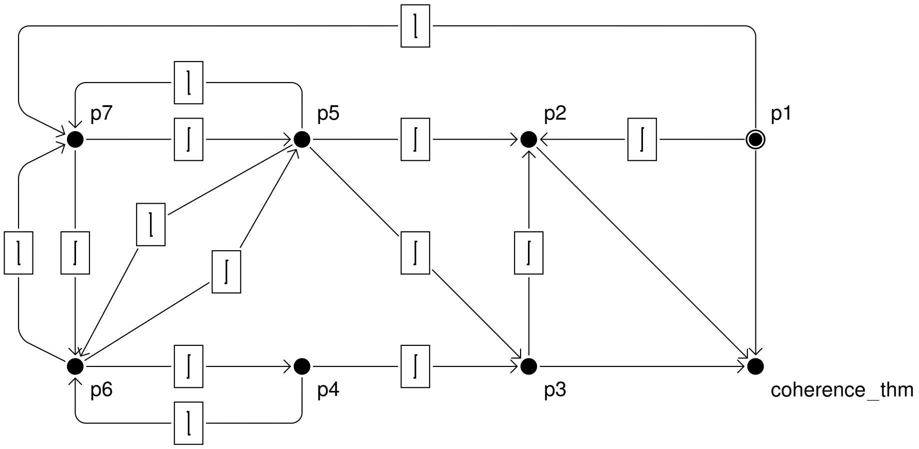

Figure 2: Proof Graph for the German Protocol

Listing 5 German Cache Coherence Protocol

```
// Types
type node
type data
type cache_state = enum {invalid, shared, exclusive}
type cache = struct {c_state: cache_state; c_data: data}
type msg_cmd = enum {empty,reqs,reqe,inv,invack,gnts,gnte}
type msg = struct {m_cmd: msg_cmd; m_data: data}
type chan = array(node, msg)
type caches = array(node, cache)
type nodes = array(node, bool)

// Constants for initialization
const data dummy
const msg imsg = {m_cmd = empty; m_data = dummy}
const cache icache = {c_state = invalid; c_data = dummy}

// State variables
var caches cache = mk_array[node](icache)
var chan chan1 = mk_array[node](imsg)
var chan chan2 = mk_array[node](imsg)
var chan chan3 = mk_array[node](imsg)
var nodes invset = mk_array[node](false)
var nodes shrset = mk_array[node](false)
var bool exgntd = false
var node exnode
var msg_cmd curcmd = empty
var node curptr
var data memdata

// Actions
transition send_req_shared (node i)
require (chan1[i].m_cmd = empty)
require (cache[i].c_state = invalid)
{chan1[i].m_cmd := reqs;}

transition send_req_exclusive(node i)
require (chan1[i].m_cmd = empty)
require (cache[i].c_state ≠exclusive)
{chan1[i].m_cmd := reqe;}

transition recv_req_shared (node i)
require (curcmd = empty)
require (chan1[i].m_cmd = reqs)
{curcmd := reqs; curptr := i;
 chan1[i].m_cmd := empty;
 invset := shrset;}

transition recv_req_exclusive (node i)
require (curcmd = empty)
require (chan1[i].m_cmd = reqe)
{curcmd := reqe; invset := shrset;
 curptr := i; chan1[i].m_cmd := empty;}

transition send_inv (node i)
require (chan2[i].m_cmd = empty)
require (invset[i])
require (curcmd=reqe ∨curcmd=reqs ∧exgntd)
{chan2[i].m_cmd:= inv; invset[i]:= false;}
```

```
transition send_invack (node i)
require (chan2[i].m_cmd = inv)
require (chan3[i].m_cmd = empty)
{chan2[i].m_cmd := empty;
 chan3[i].m_cmd := invack;
 if (cache[i].c_state = exclusive) {
   chan3[i].m_data := cache[i].c_data;}
 cache[i].c_state := invalid;
}

transition recv_invack (node i)
require (chan3[i].m_cmd = invack)
require (curcmd ≠empty)
{chan3[i].m_cmd := empty;
 shrset[i] := false;
 if (exgntd) {
   exgntd := false;
   memdata := chan3[i].m_data;}
}

transition send_gnt_shared ()
require (curcmd = reqs)
require (¬exgntd)
require (chan2[curptr].m_cmd = empty)
{chan2[curptr].m_cmd := gnts;
 chan2[curptr].m_data := memdata;
 shrset[curptr] := true; curcmd := empty;}

transition send_gnt_exclusive ()
require (curcmd = reqe)
require (¬exgntd)
require (chan2[curptr].m_cmd = empty)
require (∀ (node j) (¬shrset[j]))
{chan2[curptr].m_cmd := gnte;
 chan2[curptr].m_data := memdata;
 shrset[curptr] := true; curcmd := empty;
 exgntd := true; exnode := curptr;}

transition recv_gnt_shared (node i)
require (chan2[i].m_cmd = gnts)
{cache[i].c_state := shared;
 cache[i].c_data := chan2[i].m_data;
 chan2[i].m_cmd := empty;}

transition recv_gnt_exclusive (node i)
require (chan2[i].m_cmd = gnte)
{cache[i].c_state := exclusive;
 cache[i].c_data := chan2[i].m_data;
 chan2[i].m_cmd := empty;}

transition store_data (node i, data d)
require (cache[i].c_state = exclusive)
{cache[i].c_data := d;}

def bool coherence_thm =
  ∀(node i, node j)
    (i≠j ∧cache[i].c_state = exclusive
     ⇒cache[j].c_state = invalid)
```