Enforcing correct programming of libraries: A case study on hash tables

Néstor Cataño and Gerald Lüttgen

Department of Computer Science, University of York
Heslington, York YO10 5DD, U.K.
{catano, luettgen}@cs.york.ac.uk

Abstract. This paper presents the formalisation and correctness proofs of large parts of a hash table library used to represent multi-valued decision diagrams (MDDs). This data structure is used, for instance, in *Saturation*, a non-trivial algorithm used to store state spaces of asynchronous systems. We have conducted the correctness proofs of the main functionalities of the hash table in PVS. We outline a proof approach that can be re-used by practitioners interested in using formal methods to check their applications. We show how PVS can be used to discover inconsistencies in the implementation of an application, or to improve it.

1 Introduction

A mathematical proof is the demonstration in logic of a statement, when certain axioms are assumed. Although rigorous in essence, hand-written mathematical proofs are not fully exhaustive as it can become impossible for human beings to keep track of all the details involved in the proof. Mathematical proofs can be improved with the aid of a theorem prover, which considers all cases and can find inconsistencies. However, tool-assisted proofs tend to be tedious, *e.g.*, the tool considers all the obvious details that otherwise could simply be omitted. Tool-assisted proofs face a trade-off between level of detail and confidence in the correctness of the results.

In computer science, mathematical proofs can serve to show the correctness of a program, with respect to a specification, so to increase the confidence that the program behaves correctly under any circumstances. In particular, the correctness proofs of a library not only increases our confidence about the correctness of the implementation of the library itself, but about any other library from which it is imported.

The work presented here is in the context of the formal specification and the correctness proofs of a library for Multi-valued Decision Diagrams (MDDs) [1–3], a data-structure used, *e.g.*, in the symbolic state-space generation algorithm *Saturation* [5]. The algorithm has been implemented within the Smart tool [4]. Implementation of MDDs heavily makes use of a hash table library. We have modelled the hash table in the logic of the PVS theorem prover [10], and used it to prove the correctness of the main functions of the hash table.

The contributions of this paper are three fold. First, we present the formalisation and the correctness proofs of large parts of a hash table library used to represent MDDs (Sections 3 and 4), which are employed in a non-trivial algorithm used to store state spaces of asynchronous systems [5]. Second, we sketch a proof approach that can be re-used by practitioners not familiar with the use of formal verification tools to check their own MDD-based applications (Section 4). To do so, a set of axioms faithfully describing the implementation are defined. Based on the documentation of the application, a set of elementary lemmas expressing properties which are expected to be true are formalised. The PVS theorem prover is then used to verify the lemmas. One of the benefits of using a type-based tool, such as PVS, to do mathematical proofs is that even from the *type-correctness conditions* one is aware of inconsistencies. The PVS theorem prover counts on several (semi-) decision procedures that can be used to discharge all the obvious proofs while keeping a high level of detail that otherwise, *e.g.*, by doing hand-written proofs, is hard to keep. Third, although *Saturation* has been defined in terms of MDD operations, its implementation employs hash tables. Hence, we prove the existence of an isomorphism between MDDs and hash tables so to be sure that the implementation conforms with the specification (Section 5). We further show how this apparently simple mathematical proof, when carried out in a theorem prover, can be helpful to outline well-formed conditions on the hash table so to improve its implementation or just to adjust it. The work presented here is part of a larger work which attempts to prove the correctness of *Saturation*. Having a formal proof of the existence of such isomorphism ensures that the *Saturation* correctness proofs are sound with respect to the data structures employed by the algorithm.

Related work. In [6] Huisman, Jacobs and van den Berg verify a safety property of the standard Java's `Vector` class, which states that the number of elements a vector contains is less than or equal to its capacity. The verification is carried out in the framework of the Loop tool which takes specifications written in the Java Modelling Language (JML) [7] and transforms them in proofs obligations (for invariants, as well as for pre- and post-conditions). Unlike this work, we deal here with safety properties expressed in PVS directly.

In [13], Verma et al. describe a proof of correctness of Binary Decision Diagrams (BDDs) in Coq. Because of the extraction mechanism underlying Coq, BDDs certified algorithms are generated in Caml. In [11], R. Sumners conducts a modular proof of a BDD manager in ACL2. In [8], Krstic and Matthews describe the use of monadic interpreters to verify BDD algorithms. Because of efficiency reasons, algorithms in *Saturation* are directly implemented using MDDs, nevertheless their work still serves as a reference for comparison.

2 Preliminaries

Binary Decision Diagrams (BDDs) were initially introduced by Akers in [1] and popularised by Bryant in [2, 3]. In [2] Bryant shows that a canonical representation for binary decision diagrams can be obtained from a Binary Decision Tree

(BDT) by defining a total order between the variables involved, and removing duplicate and redundant nodes. The BDD obtained in this manner is referred to as an Ordered Binary Decision Diagram (OBDD). If used in canonical form, checking equivalence of BDDs is reduced to checking identity. Multi-valued Decision Diagrams [12] (MDDs) extend BDDs by considering functions to have finite integer domains instead of boolean ones. When used in canonical form, MDDs are isomorphic to multi-valued functions. A multi-valued function $\mathcal{F}$ on K variables is defined as follows :

$$\mathcal{F} : S_K \times \cdots \times S_1 \rightarrow \mathit{boolean}$$

where, without loss of generality, each S_k, $K \geq k \geq 1$, is considered to be the set of positive natural numbers $\{1, \cdots, |S_k|\}$. Note that the definition of multi-valued functions presented here is a curried version of an equivalent one that considers multi-valued functions to have $K-1$ (sub-) domains and a range. The definition of multi-valued functions does not allow domains to be empty. Quasi-ordered MDDs, the particular variant of MDDs used in *Saturation*, admits duplicate nodes.

Definition 1 (Multi-valued Decision Diagrams) Multi-valued decision diagrams are directed acyclic edge-labelled multi-graphs with the following properties :

1. Nodes are organised into $K+1$ levels from 0 to K. We write $\big\langle k|p\big\rangle$ to denote a generic node, where k is the node's level and p is a unique index for a node at that level.
2. Level 0 consists of two terminal nodes $\big\langle 0|0\big\rangle$ and $\big\langle 0|1\big\rangle$.
3. Level K contains only a single non-terminal node $\big\langle K, r\big\rangle$, the root, whereas levels $K-1$ through 1 contain one or more non-terminal nodes.
4. A non-terminal node $\big\langle k|p\big\rangle$ has $|S_k|$ arcs pointing to nodes at level $k-1$. An arc from the position $i_k \in S_k$ to the node $\big\langle k{-}1|q\big\rangle$ is denoted by $\big\langle k|p\big\rangle[i_k] = q$. Arcs describe *local states*.
5. No two nodes are *duplicate*, *i.e.*, there are no nodes $\big\langle k|p\big\rangle$ and $\big\langle k|q\big\rangle$ such that $p{\neq}q$ and for all $1 \leq i_k \leq S_k$ $\big\langle k|p\big\rangle[i_k] = \big\langle k|q\big\rangle[i_k]$.

In contrast to [12], not fully- but quasi-reduced ordered MDDs are considered, hence *redundant* nodes, *i.e.*, nodes $\big\langle k|p\big\rangle$ such that $\big\langle k|p\big\rangle[i] = \big\langle k|p\big\rangle[j]$ for all $i{\neq}j$, are valid according to our definitions. A sequence σ of local states $(i_k, \ldots, i_1)$ is referred to as a *global state*. Given a node $\langle k|p\rangle$, the node reached from it through a sequence σ of local states $(i_k, \ldots, i_1)$ is defined as follows :

$$\langle k|p\rangle[\sigma] = \begin{cases} \langle k|p\rangle & \text{if } \sigma = (), \text{ the empty sequence} \\ \langle (k{-}1)|\langle k|p\rangle[i_k]\rangle[\sigma'] & \text{if } \sigma = (i_k, \sigma'), \text{ with } i_k \in \mathcal{S}_k\,. \end{cases}$$

A global state σ constitutes a *path* starting at node $\langle k|p\rangle$, if $\langle k|p\rangle[\sigma] = \langle 0|1\rangle$ or $\langle k|p\rangle[\sigma] = \langle 0|0\rangle$.

MDDs to store state spaces. While the multi-valued decision diagrams proposed by Kam et. al in [12] are implemented through BDDs, implementing MDDs directly may result in a greater efficiency during state-space generation as in the case of *Saturation* [9]. This is because accessing and manipulating a local state requires to work on a single MDD node only, rather than on multiple BDD nodes that need to be traversed recursively. The MDD implementation in *Saturation* manages K separate pools of nodes, one per level. To avoid duplicate nodes, each pool is managed by a separate *unique table* stored using *extensible arrays*, whose sizes can be increased or reduced during execution. Nodes at a given level k are stored using a node array and an arc array. The array is organised according to the MDD node indexes, *i.e.*, the constant portion of the data for node $\langle k|p\rangle$ is stored in $node[k][p]$. The latter is indexed by the *start* and *size* field of the node array, *i.e.*, if $node[k][p].start = a$ and $node[k][p].size = b$, then $\langle k|p\rangle[i]$ is stored in $arcs[k][a+i]$, for $0 \leq i < b$. The arcs are stored in *truncated full format*, *i.e.*, $b < n_k$ means that $\langle k|p\rangle[b] = \cdots = \langle k|p\rangle[n_k-1] = 0$, where n_k is the node size at level k.

We use PVS to formalise and carry out the correctness proofs of the hash table. The following section briefly describes the PVS system.

2.1 The PVS system

The PVS prover system [10] is based on Higher Order Logic, *i.e.*, in PVS functions can be used as parameters, be returned as values, or be quantified. In PVS, specifications are organised in `theory`s and `datatype`s. Specifications can be elaborated with the help of definitions and `axiom`s. Further, specifications for several fundamental theories such as integers, functions and sets are available already in the system, so users do not need to (re-) define fundamental theories.

The PVS type checker analyses user-defined theories to establish their consistency. The PVS system is not decidable, so users must provide type consistency for their specifications. The type checker may generate proof obligations, which must be discharged before the theory is considered to be type correct. These proof obligations are referred to as *type-correctness conditions* (TCCs).

3 Multi-valued functions and MDDs in PVS

The formalisation of MDDs (multi-valued functions) is parametrised by the maximum number of levels (number of domains) `K`. Further, a function `Sk` returns the size of a MDD node at a particular level `k` (the size of the $\texttt{k}^{th}$ domain S_k of a multi-valued function). In PVS, the notation `[P -> Q]` describes a function type from `P` to `Q`; further `posnat` is the set of positive natural numbers. Definition of multi-valued functions requires domains to be not empty. This is achieved in PVS from the standard definition for positive natural numbers, `posnat: nonempty_type = {i: nat | i>0} containing 1`, which declares that `posnat` is a `nonempty_type` which contains at least element `1`. Since `Sk` is defined for positive values only, the size of any node at level `0` is undefined. Nodes at level `0` are $\big\langle 0|0\big\rangle$ and $\big\langle 0|1\big\rangle$.

```
K: nat
Sk: [{k:posnat|k<=K} -> posnat]
```

Sequences. In a multi-valued decision diagram, sequences appear in two forms, either vertically describing *global states* and *paths*, or horizontally describing nodes. Type `below_rho(k)` below symbolises all natural numbers less than or equal to the natural number `k`. Because of the `containing 0` declaration, we know that `0` has type `below_rho(k)` regardless of `k`. Type `below_tau(k)` is similar to `below_rho(k)`, but restricted to positive natural numbers. Should you think of `below_rho` as related to a node at any level, and `below_tau` as related to a node at any level but 0. The size of any node at level `0` is undefined; similarly `below_tau` is undefined for `k = 0`.

```
below_rho(k:nat): nonempty_type = { n:nat | n<=k } containing 0
below_tau(k:nat): type = { n:posnat | n<=k }
```

Type `fseq` below formalises a finite (possibly empty) sequence of elements of some type `T`. In PVS, the notation `[# ... #]` is used for the type record; `ln` and `sq` are the fields of the record. The size of a sequence of type `fseq` is `ln`, and the elements of the sequence are mapped by `below_tau(ln)`.

```
fseq: type = [# ln: nat, sq: [below_tau(ln) -> T] #]
```

Vertical sequences are introduced by the type `Seq(k)`. It defines a sequence of elements leading from a node at level `k` to a node at level `0`. In PVS, `` x`ln `` stands for the field `ln` of record `x`. Additionally, `fseq[T]` in the type declaration for `x`, *i.e.*, `x:fseq[T]`, refers to the particular type formed from `fseq` after replacing `T` in the declaration of `fseq` by `T` in the declaration of `Seq`. A sequence `x` of type `Seq(k)` has length `k`, *i.e.*, `` x`ln = k ``. Since `k` has type `below_rho(K)`, vertical sequences of length 0 are well-defined. The size of a horizontal sequence `Seq_Sk(k)` is given by `Sk(k)`.

```
Seq(k:below_rho(K)): type = { x:fseq[T] | x`ln=k }
Seq_Sk(k:below_tau(K)): type = { x:fseq[T] | x`ln = Sk(k) }
```

In order to be consistent, vertical sequences are to be well-defined horizontally, *i.e.*, *global state* components are to be *local states*. The type `Sequence(k)` models all the vertical sequences `x`, `x:Seq[posnat](k)`, such that their sizes are correctly described by `Sk`, `` x`sq(i) <= Sk(i) ``.

```
Sequence(k:below_rho(K)): type =
  { x:Seq[posnat](k) | ∀(i:below_tau(k)): x`sq(i) <= Sk(i) }
```

Definition for sequences introduced here are general in the sense that they are parametrised by a type `T`.

Multi-valued Decision Diagrams. MDDs are introduced by the *data type* `mddstr` below, which is parametrised by `K` and `Sk`. In PVS, elements of a data type are formed using *type constructors*, *e.g.*, `zero`, `one` and `node`. Type constructors may have parameters. *Type recognisers*, *e.g.*, `zero? one? node?` are predicates that group elements of a certain sub-type. For instance, `one?` evaluated in an element `m` of type `mddstr` evaluates to true only if `m` has been formed using the constructor `one`. In other words, `one?` groups all the elements of type `mddstr` formed using the type constructor `one`. Non-trivial MDDs, which are formed using the type constructor `node`, are parametrised by their level `ln`, a unique identifier `index`, and a sequence of `subnodes` of type `mddstr` and length `ln`, `subnodes: Seq_Sk[mddstr](ln)`. Notice that, because of the definition of `below_tau`, non-trivial MDDs are defined for positive values of `ln` only. Definition of `node` does not constraint `subnodes` to refer to `ln - 1`, contrarily to what the definition of quasi-ordered MDD imposes. Parameter `ln` in `subnodes` only expresses that the number of `subnodes` of any non-trivial MDD node is `Sk(ln)`. Additionally, parameters in PVS are also functions, *e.g.*, `ln` evaluated in a node `m` such that `node?(m)` returns its level. Node `zero(k)` represents a *null node* at level `k`.

```
mddstr[K:nat, Sk:[{n:posnat|n<=K} -> posnat]]: datatype
 begin
  zero(l0: below_rho(K)): zero?
  one (l1: below_rho(K)): one?
  node(ln: below_tau(K), index: nat, subnodes: Seq_Sk[mddstr](ln)): node?
 end mddstr
```

Quasi-ordered MDDs. The definition of `orderedmdd?` below imposes the condition that MDDs and their sub-nodes occur in consecutive levels. Any MDD `m` such that `zero?(m)` or `one?(m)` is `orderedmdd?` regardless of its level. The expression **measure** makes reference to the parameter that decreases in the recursive call to `orderedmdd?`. This is needed for termination. More concretely, **measure** `m by <<` expresses that, if considering the sub-term order relation `<<` on `mddstr`, in any recursive call to `orderedmdd?(m: mddstr)`, `m` decreases.

```
orderedmdd?(m: mddstr): recursive bool =
 zero?(m) ∨ one?(m) ∨
 ( node?(m) ∧
   ∀(i: below_tau(subnodes(m)`ln)): ln(m)=ln(subnodes(m)`sq(i))+1 ∧
    orderedmdd?(subnodes(m)`sq(i)) )
measure m by <<
```

Additionally predicate `orderedmddk?` characterises all the ordered MDDs at some level `k`. More concretely, `orderedmddk?` is formalised as the conjunction of predicates `orderedmdd?` and `mddk?`, defined below.

```
mddk?(k:below_rho(K))(m:mddstr): bool =
 (zero?(m) ∧ l0(m)=k) ∨ (one?(m) ∧ l1(m)=k) ∨ (node?(m) ∧ ln(m)=k)

orderedmddk?(k:below_rho(K))(m:mddstr): bool = mddk?(k)(m) ∧ orderedmdd?(m)
```

Not fully- but quasi-reduced MDDs are considered in *Saturation*, *i.e.*, the property "no two sub-nodes are duplicate" is enforced although redundant nodes are permitted. An MDD `m` `has_no_duplicates?`, if any pair of subnodes `m1` and m2 of m, subterm(m1,m) and subterm(m2,m), differ in at least one sub-node at some position i, *i.e.*, subnodes(m1)‘sq(i) /= subnodes(m2)‘sq(i).

```
has_no_duplicates?(m: mddstr): bool =
∀(m1,m2:mddstr):
 subterm(m1,m) ∧ subterm(m2,m) ∧ m1 /= m2 ∧
 nontrivialmdd?(m1) ∧ nontrivialmdd?(m2) ∧
 subnodes(m1)‘ln=subnodes(m2)‘ln ⇒
  ∃(i:below_tau(subnodes(m1)‘ln)): subnodes(m1)‘sq(i)/=subnodes(m2)‘sq(i)
```

Predicate quasiordered? combines the definitions of orderedmdd? and has_no_duplicates?, and mdd expresses quasiordered? as a type: in PVS, given a predicate P, (P) is the type induced when considering only those elements satisfying the predicate. Notice that quasiordered?(zero(k)) holds for any k.

```
quasiordered?(m: mddstr): bool = orderedmdd?(m) ∧ has_no_duplicates?(m)
mdd: type = (quasiordered?)
```

Paths in MDDs. We define paths in MDDs following the definition of *path* introduced in Section 2. In mddpath1 below, a sequence σ of local states $(i_k, \ldots, i_1)$ refers to a sequence s of type Seq[nat](k) of local states s‘sq(i). Conditions k=0 and one?(m) ensure that $\langle k|p\rangle[\sigma] = \langle 0|1\rangle$ when $k = 0$. For k>0, a recursive call to mddpath1 is made passing the subsequence subSeq[nat](k,k-1)(s) of length k-1 as parameter. Conditions s‘sq(s‘ln)>0 and s‘sq(s‘ln)<=subnodes (m)‘ln ensure that the kth element of s is correctly described by Sk. MDDs end in zero(0) as well, consequently, a similar definition for mddpath0 is provided to reflect this. Paths in MDDs, mddpath, are defined as the logical disjunction between mddpath1 and mddpath0.

```
mddpath1(k:below_rho(K))(m:mddstr)(s:Seq[nat](k)): recursive bool =
 ( k=0 ∧ one?(m) ∧ l1(m)=k ) ∨
 ( k>0 ∧ node?(m) ∧ s‘sq(s‘ln)>0 ∧ s‘sq(s‘ln)<=subnodes(m)‘ln ∧
   mddpath1(k-1)(subnodes(m)‘sq(s‘sq(s‘ln)))(subSeq[nat](k,k-1)(s))
 )
measure k

mddpath(k:below_rho(K))(m:mddstr)(s:Seq[nat](k)): bool =
 mddpath1(k)(m)(s) ∨ mddpath0(k)(m)(s)
```

Multi-valued functions. A multi-valued function F on k variables is a function from a domain D, a Sequence(k), to boolean numbers. Notice that this is a curried version of the equivalent definition of multi-valued functions having a domain formed of k - 1 variables (k > 1) and a range.

```
D(k:below_rho(K)): type = Sequence(k)
F(k:below_rho(K)): type = [D(k) -> bool]
```

4 Formalising hash tables

Hash tables use hash functions to associate keys to array indexes. Ideal hash functions are collision-free. Two different keys collide if their indexes are the same. It is always possible to define a collision-free hash function when the number of possible keys is finite. We assume here the existence of a collision-free hash function. This assumption does not introduce any contradiction since the number of possible keys we can define for our hash tables is finite. The hash table formalisation presented here is oriented by the hash table implementation introduced in Section 2, and is close to the formalisation of MDDs. For instance, hash tables are also organised per levels. In a hash table, sub-entries indexes (sub-nodes indexes) are unique per level. Therefore, sub-entries are used as hash table keys and their uniqueness is axiomatised.

The correctness proof approach described here can be followed by practitioners interested in the use of formal verification tools to check (part of) their developments. In a first stage, a set of axioms faithfully describing the implementation are defined. These axioms are to be simple and of course not contradictory. The axioms as well as the whole formalisation are based on the documentation of the application. As for programs in any programming language, axioms should pass through an iterative refinement process. A natural way to check axioms is to define a set of elementary lemmas, based on the documentation of the application, expressing properties that are expected to be true, and then using a formal methods tool, *e.g.*, The PVS prover, to verify the lemmas. In our case, the axiomatisation should provide a formal basis to hash table indexes and to how particular indexes relate to particular keys at particular levels. When lemmas cannot be proved correct, two possible actions are to be undertaken. Either the axiomatisation is wrong, *e.g.*, it is contradictory, so a new one must be provided, or our understanding of what the implementation should be is incorrect and so the implementation must be improved or just changed.

In a second stage, library functions are formalised. In the following we define hash tables structures, then we axiomatise hash tables. We finally formalise typical hash table functions. We describe how the process of formal modelling and verification using the PVS theorem prover give us insight on what the implementation of the hash table must be.

Hash table structure. Indexes `idx` are formalised as natural numbers.

```
idx: type = nat
```

A hash table is a function from indexes `idx` to table structures `tablestr` (defined below). This definition of hash table is more general than the implementation presented in Section 2, *e.g.*, this definition is not bounded to extensible arrays.

```
hash_table: type = [idx -> tablestr]
```

In `tablestr`, an element `zeroentry(v0)` represents the entry for the MDD `zero(v0)`, similarly `onentry(v1)` represents the entry for `one(v1)`. A non-trivial entry `nontrvlentry` represents a non-trivial `node` in memory. Furthermore, `nonentry` is the default value for indexes not bounded to any particular entry in the hash table.

```
tablestr[K: nat, Sk: [{n:posnat|n<=K} -> posnat]]: datatype
 begin
  zeroentry(v0: below_rho(K)): zeroentry?
  onentry(v1: below_rho(K)): onentry?
  nontrvlentry(vn: below_tau(K), entries: Seq_Sk[nat](vn)): nontrvlentry?
  nonentry: nonentry?
 end tablestr
```

Hash table axiomatisation. We first axiomatise hash tables and then define a set of lemmas to check whether the axiomatisation precisely describes our understanding of the implementation. We define three special indexes `idx_0`, `idx_1` and `root`.

```
 idx_0, idx_1, root: idx
```

Furthermore, indexes `idx_0` and `idx_1` should be distinct.

```
 idx_0_1_are_distinct: axiom idx_0 /= idx_1
```

and they should be the only indexes associated to `zeroentry(0)` (Axiom `hash_table_ax0`) and `onentry(0)` (Axiom `hash_table_ax1`) respectively.

```
hash_table_ax0: axiom ∀(t:hash_table,p:idx): t(p)=zeroentry(0) ⇔ p=idx_0
hash_table_ax1: axiom ∀(t:hash_table,p:idx): t(p)=onentry(0) ⇔ p=idx_1
```

We still need to formalise `root`, in particular we should formalise how it relates to `idx_0` and `idx_1`. In the simplest case, when `K=0`, `root` should be exactly one of `idx_0` or `idx_1` (Axiom `root_ax1`). Otherwise, `K>0` and `root` is neither `idx_0` nor `idx_1` (Axiom `root_ax2`).

```
root_ax1: axiom K=0 ⇒ root = idx_0 ∨ root = idx_1
root_ax2: axiom K>0 ⇒ root /= idx_0 ∧ root /= idx_1
```

The sub-entries of a hash table entry `t(p)` are given by `entries(t(p))`. The uniqueness of hash tables sub-entries is axiomatised by `hash_table_is_collision_free` below. This axiom states that for any hash table `t` and any entry indexes `p1` and `p2`, if the sub-entries indexes for `p1` and `p2` are the same, `entries(t(p1)) = entries(t(p2))`, then the entry indexes `p1` and `p2` coincide, `p1=p2`.

```
hash_table_is_collision_free: axiom
 ∀(t:hash_table, p1,p2:idx):
  nontrvlentry?(t(p1)) ∧ nontrvlentry?(t(p2)) ⇒
    entries(t(p1)) = entries(t(p2)) ⇒ p1=p2
```

Now, let us check whether our formalisation faithfully describes our understanding of the implementation. Let us prove whether non-trivial entries are only associated with indexes other than `idx_0` and `idx_1`, and vice-versa. We prove the two cases of the logical equivalence separately. First, lemma `nontrvlentry_-assoc_with_nontrvlidx` states that, for any hash table `t` and any index `p`, if `t(p)` is non-trivial, `nontrvlentry?(t(p))`, then `p` is none of `idx_0` or `idx_1`.

```
nontrvlentry_assoc_with_nontrvlidx: lemma
 ∀(t:hash_table,p:idx): nontrvlentry?(t(p)) ⇒ p/=idx_0 ∧ p/=idx_1
```

From axiom `hash_table_ax0` (`hash_table_ax1`) we know that for any hash table `t` and any index `p`, if `t(p)` is different to `zeroentry(0)` (`onentry(0)`), *e.g.*, nontrvlentry?(t(p)), then `p/=idx_0` (`p/=idx_1`). Therefore, the lemma is proved. The actual PVS proof script follows the proof described here.

```
nontrvlidx_assoc_with_nontrvlentry: lemma
 ∀(t:hash_table,p:idx): p/=idx_0 ∧ p/=idx_1 ⇒ nontrvlentry?(t(p))
```

Now, let us prove the other direction of the implication. When trying the same proof script as before in PVS, the proof is not finished. Some possible reasons for this is that the axiomatisation provided to `idx_0` and `idx_1` are incorrect, *e.g.* the axiomatisation is contradictory, or a different proof script should be tried instead, or our intuition about the hash table implementation is incorrect. If axioms `hash_table_ax0` and `hash_table_ax1` are checked carefully one realises that `p/=idx_0` (`p/=idx_1`) entails only that `t(p)/=zeroentry(0)` (`t(p)/=onentry(0)`), but never that `nontrvlentry?(t(p))`. Further, from our formalisation we know already that `zeroentry(k)` and `onentry(k)` can exist at any level `k` (even at some positive level). We are now aware of that this fact has a consequence that `nontrvlentry?` and non-trivial entries (`zeroentry(k)` for `k>0` for instance) are far to be the same. The problem we have found here probably also would have been found doing our proofs by hand instead of using PVS, but by using a tool one is sure not to forget some cases. Further, writing the formal specifications forces one to think very precisely about the intended behaviour of programs, which helps in finding inconsistencies.

Hash table functions. Function `memberTbl?` below checks for the existence of an associated key `s`. More concretely, given a hash table `t`, a key `s` has previously been associated in `t`, if an index `p` such that `t(p) = nontrvlentry(k,s)` exists. Notice that the type definition for index `p` in the existential quantifier, ensures that `p` is different to `idx_0` and `idx_1`. This is in accordance with the fact that a node (entry) at level `0` has no sub-nodes (sub-entries). Additionally, the type definition for `p` suggests that indexes other than `idx_0` and `idx_1` can only be associated to `nontrvlentry`. This is stated by the lemma `nontrvlentry_assoc_-with_nontrvlkey` below. The lemma is proved in PVS using axioms `hash_-table_ax1` and `hash_table_ax0`. We can thus be sure that `memberTbl?` behaves as expected.

```
memberTbl?(k:below_tau(K), t:hash_table, s:Seq_Sk[idx](k)): bool =
 ∃(p:{x:idx | x/=idx_0 ∧ x/=idx_1}): t(p)=nontrvlentry(k,s)

nontrvlentry_assoc_with_nontrvlkey: lemma
 ∀(t:hash_table,p:key): nontrvlentry?(t(p)) ⇒ p/=key_0 ∧ p/=key_1
```

Further, an empty hash table can naively be defined as having associated every index `p` to `nonentry`, `t(p) = nonentry`. Let us check whether this contradicts our axiomatisation. From axioms `hash_table_ax0` and `hash_table_ax1`, we further need to accept empty hash tables to have indexes `key_0` and `key_1` associated to correct values. Think of this as being the base case in the definition of an induction scheme on hash tables. An empty hash table is introduced by the predicate `empty_hash_table?` below. The variable `empty_hash_table` is declared to have type `(empty_hash_table?)`.

```
empty_hash_table?(t:hash_table): bool =
 t(key_0)=zeroentry(0) ∧ t(key_1)=onentry(0) ∧
 (∀(p:key): p/=key_0 ∧ p/=key_1 ⇒ t(p)=nonentry)

empty_hash_table: (empty_hash_table?)
```

To check whether our definition of `empty_hash_table?` correctly describes empty hash tables, we prove whether an empty hash table has members (See Lemma `empty_hash_table_has_no_members`). This lemma is proved in PVS after expanding the definitions for `memberTbl?` and `empty_hash_table?`.

```
empty_hash_table_has_no_members: lemma
not ∃(k:below_tau(K), s:Seq_Sk[key](k)): memberTbl?(k,empty_hash_table,s)
```

Further, function `lookup` is defined resembling `memberTbl?`. The only difference is that `lookup` additionally returns the index that has already been associated to key `s`. Function `the` is a standard PVS function which takes a singleton set and returns "the" element in the set.

```
lookup(k:below_tau(K), t:hash_table,
       s:{sl:Seq_Sk[idx](k) | memberTbl?(k,t,sl)}): idx =
 the({p:{x:idx|x/=idx_0 ∧ x/=idx_1} | t(p)=nontrvlentry(k,s)})
```

Condition `memberTbl?(k,t,sl)` in the type definition for `s` ensures that an index `p` in **the(...)** exists. Additionally, theorem provers take care of subtle details that can easily be overlooked when doing mathematical proofs by hand, *e.g.*, the PVS theorem prover generates automatically a type correctness condition for the type of the set in **the(...)**, which must be singleton. The following type correctness condition, automatically generated by PVS, must thus be discharged:

```
lookup_TCC1: obligation
p/=idx_0 ∧ p/=idx_1 ∧ t(p)=nontrvlentry(k,s) ⇒
 ∀(y:({p:{x:idx | x/=idx_0 ∧ x/=idx_1} | t(p)=nontrvlentry(k,s)})): p=y
```

To prove this, suppose that an element `y` such that `y/=idx_0` and `y/=idx_1` and `t(y)=nontrvlentry(k,s)` exists. From `hash_table_is_collision_free`, it follows immediately that `y = p`. Therefore, the uniqueness is guaranteed and we are sure that our formalisation is consistent and that our implementation is correct. Again, the actual PVS proof script follows the proof described above. Keeping track of all the subtle conditions involved in a proof, *e.g.*, singletoness of a set, is less error prone with the aid of a formal verification tool.

Finally, function `insert` associates a key `s` with an index `p` in hash table `t`. This index must be formalised. It should be different to `idx_0` and to `idx_1` as they are reserved for entries that have no sub-entries. Function `nextTblIndex(k,t)` describes the type of index `p` in `insert`. Additionally to `nextTblIndex`, axiom `nextTblIndex_ax1` states that for any level `k` and for any hash table `t`, its next free index slot, `p=nextTblIndex(k,t)`, is not associated in `t`, `nonentry?(t(p))`. `nextTblIndex` is not constrained to a particular implementation, but to anyone that respects `nextTblIndex_ax1` and agrees with the type of `nextTblIndex`. This type clearly suggests a post-condition for `nextTblIndex` and a pre-condition for `insert`. These conditions can be used to improve the implementation, *e.g.* asserting the post-condition at the end of `nextTblIndex` and the pre-condition in any place where `insert` is called.

```
insert(k:below_tau(K), t:hash_table, p:idx, s:Seq_Sk[idx](k))
  : hash_table = t with[(p):=nontrvlentry(k,s)]

nextTblIndex(k:below_tau(K), t:hash_table): {x:idx | x/=idx_0 ∧ x/=idx_1}
nextTblIndex_ax1: axiom
 ∀(k:below_tau(K), t:hash_table):
  nextTblIndex(k,t)/=idx_0 ∧ nextTblIndex(k,t)/=idx_1 ∧
  (∀(p:idx): p=nextTblIndex(k,t) ⇒ nonentry?(t(p)))
```

In PVS, a type correctness condition, is automatically generated to prove the existence of the value returned by a function. This is the case for `nextTblIndex` above. More specifically, some index `x` different from `idx_0` and `idx_1` must exist for any level `k` of type `below_tau(K)`. To prove the existence of this element, it suffices to take `x = root` and use the axiom `root_ax2` introduced above. One lesson that can be learnt from this is the following : the use of a formal tool to check our specifications ensures that none of the checking is forgotten. Without the use of a formal tool it would have been very easy to forget to check the existence of this value.

Paths in hash tables. They are defined similarly to paths in MDDs except for the additional condition `p=idx_1` (`p=idx_0`) for `k=0` and `p/=idx_0` ∧ `p/=idx_1` for `k>0` in the definition of `tablepath1` (`tablepath0`).

Discussion. Following Section 2, each MDD level is managed by a unique table, stored using extensible arrays, implemented as linked lists. Checking whether a node $\langle k|p\rangle$ has been added to the unique table requires to look at the portion of code where nodes at level k are kept and sequentially check for index equality against p. This is largely inefficient if compared with the functional approach used in the formalisation of `memberTbl?` where no local search is needed as the checking is simply represented as `t(p)=nontrvlentry(k,s)`. In *Saturation*, referencing a node is a frequently used operation, so this search is a source of inefficiency. A way to alleviate this would be to introduce an order on the linked list implementing hash tables and to do ordered search instead.

5 Proving that MDDs and hash tables are isomorphic

This Section proves the existence of an isomorphism between MDDs and hash tables. Notice that the type `hash_table: [idx -> tablestr]` becomes finite when the type `idx` is restrained by the hash table axiomatisation described in Section 4. Proving the existence of an isomorphism between two structures requires us to find a bijective homomorphism between them. In the following, we show how such an homomorphism from multi-valued functions to hash tables can be constructed. To find this homomorphism we need to define a function so that a hash table can be constructed from a multi-valued function. In the following, we axiomatise this function, which we refer as `build`.

```
build(k:below_rho(K), f:F(k)): hash_table
```

We claim that this function must verify a `canonicity` property which states that if a multi-valued function `f` evaluates to true for the sequence `x`, then it is a path ending in `onentry(0)` in the hash table `build(K,f)`, `tablepath1(K, root, build(K,f))(x)`, and vice-versa. Something similar happens when `f` evaluates to false.

```
canonicity: lemma
 ∀(f:F(K), x:D(K)): (not f(x) ⇔ tablepath0(K,root,build(K,f))(x)) ∧
                     (f(x) ⇔ tablepath1(K,root,build(K,f))(x))
```

Existence of an injective homomorphism. We claim that the following homomorphism `h` between multi-valued functions and hash tables is injective.

```
h = λ(f:F(K)): build(K,f)
```

We prove that for all multi-valued functions `f1` and `f2`, if `build(K,f1)` is equal to `build(K,f2)`, then `f1=f2`. To prove this, notice that the following result holds due to lemma `canonicity`:

```
f1(x) ⇔ tablepath1(K,root,build(K,f1))(x)
f2(x) ⇔ tablepath1(K,root,build(K,f2))(x)
```

Therefore, replacing `build(K,f1)` by `build(K,f2)` in the first equivalence, we obtain `f1=f2`, as desired.

Existence of a surjective homomorphism. We prove that, for all hash table `t`, a multivalued function `f` of type `F(K)` exists such that `h(f) = t`. More specifically, we need to define an inverse homomorphism `h1` of `h` such that `h(h1(t)) = t`. We thus need to define "equality" of hash tables. We propose `hash_table_equality` as an appropriate definition for hash table equality. This definition states that two hash tables `t1` and `t2` are equal if only if any path in one table is a path in the other table and vice-versa.

```
hash_table_equality: assumption
 ∀(t1,t2:hash_table):
 t1=t2 ⇔
 (∀(s:Seq[idx](K)): tablepath1(K,root,t1)(s) ⇔ tablepath1(K,root,t2)(s) ∧
                    tablepath0(K,root,t1)(s) ⇔ tablepath0(K,root,t2)(s))
```

We claim that `h1 =` λ`(t: hash_table):` λ`(x: D(K)): tablepath(K,root,t) (x)` is an inverse homomorphism of `h`. The homomorphism `h1` takes a hash table `t` and returns the function in which the domain is formed of those elements `x` that are paths in `t` starting from the `root`, `tablepath(K,root,t)(x)`.

After expanding the definition of `h1` and `h`, the proof of `h1(h(t)) = t` reduces to `build(K,`λ`(x:D(K)): tablepath(K,root,t)(x))) = t`.

We do not prove the last equality. Instead, we state that this equality describes a property that a correct definition for `build` must satisfy. Additionally, from this equality some well-formed conditions on the implementation of the hash table can be outlined. The notion of `tablepath` suggests that elements in the hash table, *e.g.* node indexes, are to be referenced and reachable from each other. The use of linked lists, as in the case of the hash table implementation in *Saturation*, Section 2, is a valid alternative. The type of `x`, `Sequence` of size `K`, suggests that the hash table can be seen as composed of `K` levels. A special index `root` refers to level `K`. The MDD implementation in *Saturation* manages K separate pools of nodes, one per level, though other implementations are also possible. Further, when applying `hash_table_equality` to the equality above, it comes that `build`, which sees global states as function domains, and `tablepath` play complementary roles: paths in the hash table must be unique from top to bottom starting at index `root`. These conditions would have been outlined after a subtle analysis of the hash table requirements in *Saturation*, but having a formal methods tool ensures that the analysis is formally correct.

6 Conclusion and future work

The formalisation and proofs presented here are part of a larger work which involves the formalisation and correctness proof of the algorithm *Saturation* [5]. The correctness property can be phrased as the characterisation of a fixed-point of the state-space generated for a MDD node. To do this correctness proof we have considered necessary to formalise and conduct the correctness proofs of the infrastructural components used by the algorithm, *e.g.*, MDDs, before. Additionally, because the implementation of MDDs heavily makes use of a hash table library, we have proved the existence of an isomorphism between MDDs and hash tables. Having a formal proof of the existence of this isomorphism ensures that the *Saturation* correctness proofs are sound with respect to the data structures employed by the algorithm. Our proof approach can be used for practitioners interested in using formal methods to check their applications. By doing mathematical proofs in a theorem prover not only ensures that no details are overlooked but also serves to gain insight in how the implementation really works. This insight can then be used to improve the implementation.

In PVS, *type-correctness conditions* (TCCs) enable a separation of concerns, so that structural proof-constraints do not clutter the actual proof one is reasoning about. In our case, MDDs are very structured. If we would have to put every constraint regarding well-structuredness as part of our lemmas, our formalisation would look quite ugly. Additionally, PVS's ability to discharge most TCCs automatically allows us to conduct the proofs much quicker and thus makes formal methods more economical.

Acknowledgements. We thank Jim Woodcock, Laurent Théry and Leonardo Freitas for useful comments on earlier versions of this paper.

References

1. S. B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6), Jun. 1978.
2. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.
3. Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
4. G. Ciardo, R. L. Jones, A. S. Miner, and R. Siminiceanu. Logical and stochastic modeling with Smart. In *Modeling Techniques and Tools for Computer Performance Evaluation*, volume 2794 of *Lecture Notes in Computer Science*, pages 78–97. Springer-Verlag, Sep. 2003.
5. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state-space generation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 328–342, Genova, Italy, April 2001. Springer-Verlag.
6. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. *Software Tools for Technology Transfer (STTT)*, 3/3:332–352, 2001.
7. The Java Modeling Language. `http://www.cs.iastate.edu/~leavens/JML/`.
8. S. Krstic and J. Matthews. Verifying BDD algorithms through monadic interpretation. In *Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 2294 of *Lecture Notes in Computer Science*, pages 182–195. Springer, 2002.
9. A. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Applications and Theory of Petri Nets*, volume 1639 of *Lecture Notes in Computer Science*, pages 6–25. Springer-Verlag, Jun. 1999.
10. The PVS Specification and Verification System. `http://pvs.csl.sri.com/`.
11. R. Sumners. Correctness proof of a BDD manager in the context of satisfiability checking. ACL2 workshop, 2000.
12. R. K. Brayton Timothy Y.K. Kam, T. Villa and A. L. Sangiovanni-Vincentelli. Multi-valued decision diagrams: Theory and applications. *Multiple-Valued Logic*, 4(1–2), 1998.
13. K. N. Verma, J. Goubault-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In *Asian Computing Science Conference*, volume 1961, pages 162–181. Springer Verlag, 2000.