mCRL2 Term Library

Maurice Laveaux

May 18, 2019

This document contains a description of the term library as implemented in the mCRL2 toolset. The structure of the report is as follows. In Section 1 the underlying terms are formally defined. In Section 2 the programming interface and its high level requirements are presented. In Section 3 a generic implementation is described to highlight the system architecture and its intended behaviour. In Section 4 the generic implementation is improved with several performance optimizations that have been applied.

1 Definitions

We define *terms* in the context of a set of *function symbols*.

Definition 1 (Function symbols) *Let $\mathcal{S}$ be an arbitrary set of symbols. The set of function symbols is defined $\mathcal{F} \subseteq \mathcal{S} \times \mathbb{N}$.*

The natural number indicates the *arity* of a function symbol. The arity function α is returns this arity for every function symbol, formally for a function symbol (f, i) then $\alpha((f, i))$ is equal to i.

Definition 2 (Terms) *The set of terms $\mathcal{T}$ is inductively defined. For terms $t_0, ..., t_n \in \mathcal{T}$ and function symbol $f \in \mathcal{F}$ where $\alpha(f)$ is equal to n the* function application *$f(t_0, ..., t_n) \in \mathcal{T}$.*

The function symbol in a function application is referred to as *head* function symbol. Note that function symbols with arity zero are terms as well. Let $args$ be a function on terms that denotes the set of terms in a function application. Formally, for a function application $f(t_0, ..., t_n)$, let $args(f(t_0, ..., t_n))$ be equal to the list of arguments $[t_0, ...t_n]$.

1.1 Classes

There are several useful classes of terms that can be identified. As an example the class of *list* terms. Let $(+\!+, 2)$ and $([\,], 0)$ be function symbols to define list concatenation and the empty list.

Definition 3 (Term lists) *The set of* list *terms $\mathcal{L}$ is inductively defined as:*

1. *The* empty *list* $([\,], 0) \in \mathcal{L}$.
2. *If $t \in \mathcal{T}$ and $l \in \mathcal{L}$, then the* concatenation *function application $+\!+\,(t, l) \in \mathcal{L}$.*

Besides lists there are also different classes, such as *term string* where a string is stored as part of the function symbol name and *binary tree* where every left and right sub-tree is stored as arguments.

2 Term Library

The term library provides the storage (in memory and on disk) of terms and function symbols. Formally, the term library stores a finite subset of terms $\mathcal{T}' \subseteq \mathcal{T}$ and finite subset of function symbols $\mathcal{F}' \subseteq \mathcal{F}$. The library is designed with the following goals in mind:

1. Minimize the amount of memory used to the set of terms.
2. Fast creation (and deletion) of terms.
3. Fast equality comparison between terms.

2.1 Application Programming Interface

The library is implemented in the C++11 compliant subset of the C++ language. It consists of several classes that allow the user to create function symbols and term applications via their constructors. A constructor is a language construct that allow a user to instantiate an object of a specific class. The set of symbols $\mathcal{S}$ is represented by a set of strings, where each string indicates the name of a function symbol. The relation between the mathematical elements and these constructors are listed in Figure 1.

Definition	API
$(name, arity) \in \mathcal{F}$	function_symbol$(name, arity)$
$f(t_0, ..., t_n) \in \mathcal{T}$	term_appl$(f, t_0, ..., t_n)$

Figure 1: The relation between the definition and the API

For the sub-classes of terms there are specific constructors defined as shown in figure 2.

Definition	API
$([\,], 0) \in \mathcal{F}$	`function_symbol(<empty_list>,0)`
$(+\!+, 2) \in \mathcal{F}$	`function_symbol(<list_constructor>,2)`
$+\!+\,(t, l) \in \mathcal{L}$	`term_list(t, l)`

Figure 2: The relation between the definition and the API

Besides the creation of terms there are additional operators and functions defined. There are binary operators to check equality (and inequality) of terms. Note that there is no semantic inequality defined, but from an implementation viewpoint this is useful to have. This inequality is implemented by comparing the position where they are stored in memory.

There are also various functions defined to access the information carried by a term. There is a `function` function that maps a term to the function symbol of which it was constructed. The `arg` function is defined that takes a function application and an index as input and returns the argument term at the specified index.

3 Implementation

In this section the architecture of the library is described and a generic implementation is described. The main architectural choice for this library is that terms and function symbols are maximally shared. This effectively implies that function applications refer to their arguments in the set $\mathcal{T}'$ and a function symbol in the set $\mathcal{F}'$. These sets can be represented by an unordered set as described in the C++ standard template library (STL) which can be efficiently implemented using a hash table. The reason for this sharing is to facilitate the minimize memory and constant time comparison goals stated in Section 2. For the purpose of sharing we assume that there exists a `shared_reference` class that internally counts the number of references to each term or function symbol. The `term` and `function_symbol` classes internally use this class. A reference means that the user created a term or function symbol object, that a term occurs as an argument in a function application or that a function symbol occurs in a function application.

Now there is a function symbol pool class that stores the subset of function symbols $\mathcal{F}'$ and a term pool to store the subset of terms $\mathcal{T}'$. The constructors described in the API use a single instance of each of these pools to ensure that every created term of function symbol is unique. The pool classes will be described in detail in the next sections.

3.1 Function symbol pool

This class provides the `create_function_symbol` method to define new function symbols. It also provides the destruction of function symbols with a reference count of zero.

Internally it uses the set $\mathcal{F}'$ which provides constant time insertion $\cup$, deletion $\setminus$ and contains $\in$ functions.

3.1.1 Creating function symbols

The constructor of a function symbol calls the following function to obtain a reference to an existing or new function symbol.

- create_function_symbol(name : String, arity : Nat) : $\mathcal{F}$

This method either returns an existing function symbol in $\mathcal{F}'$ or it results in a new function symbol function_symbol($name, arity$). This new term will also be added to the current set of function symbols. The pseudocode of this function becomes:

Algorithm 1 Creation of function symbols

```
1: procedure CREATE_FUNCTION_SYMBOL(name, arity)
2:     if function_symbol(name, arity) ∈ F' then
3:         return function_symbol(name, arity)
4:     end if
5:     f ← construct(name, arity)
6:     F' ← F' ∪ {f}
7:     return f
8: end procedure
```

The value returned in the if-condition is an element that already exists in F. In the implementation checking whether function_symbol$(name, arity) \in \mathcal{F}'$ can be done without constructing the function symbol, which can be expensive. The *construct* function allocates a new function symbol object on the heap with the given name and arity.

For each reference the reference counter indicates the amount of times that an object is referred to. For cleaning up objects two different approaches can be taken. The *destructor* of a class is a function that is called whenever an instance of that class is destroyed. It is possible to immediately destroy the object when its reference count becomes zero. This approach will be referred to *direct-destruction*. Another approach is to periodically clean up all terms with a zero reference count. This approach will be referred to as *garbage collection*.

For function symbols the *direct-destruction* method is used, which means that the allocated space is immediately freed whenever its reference count becomes zero. This destruction is implemented by the `destroy` function that deallocates the memory used and removes it from the set S.

Algorithm 2 Destruction of function symbols

```
1: procedure DESTROY(f)
2:     deallocate(f)
3:     F' ← F' \ {f}
4: end procedure
```

The *destroy* function cleans up the heap memory used by the function symbol f.

3.2 Term pool

To be able to share terms, all constructors interact with a single instance of a class called *term_pool*. The term pool is the class that stores the set of terms $\mathcal{T}'$. This class provides the *create_appl* method to define new function applications. It also provides the destruction of terms with a reference count of zero.

3.2.1 Creating terms

The constructor of a function application calls the following function to obtain a reference to an existing or new term.

- create_appl(f : $\mathcal{F}$, $t_0, ..., t_n : \mathcal{T}$) : $\mathcal{T}$

The function symbol f and terms t_0 to t_n should be elements of the current subset of stored function symbols $\mathcal{F}'$ and terms $\mathcal{T}'$ respectively. The pseudocode of this function is:

Algorithm 3 Creation of term applications

```
1:  procedure CREATE_APPL(f, t_0, ..., t_n)
2:      if term_appl(f, t_0, ...t_n) ∈ T' then
3:          return term_appl(f, t_0, ...t_n)
4:      end if
5:      t ← construct(f, t_0, ..., t_n)
6:      if should-collect-garbage() then
7:          collect()
8:      end if
9:      T' ← T' ∪ {t}
10:     return t
11: end procedure
```

The terms are stored on the heap as follows:

Name	Size
function symbol reference	8
reference counter	8
t_0 reference	8
...	...
t_n reference	8

A reference is typically stored as a pointer; which requires 8 bytes on a 64bit program.

The boolean function *should-garbage-collect* is used to decide when garbage collection should be performed. The garbage collection itself is implemented by the *collect* function. Currently the *should-garbage-collect* function is implemented by using a counter that is decremented on every call to *should-garbage-collect*. Whenever this counter reaches zero the function returns true. During garbage collection this counter is then set to the number of terms in $\mathcal{T}'$, which is equal to $|S|$.

3.2.2 Garbage collection

The advantage of garbage collection is that terms with a reference count of zero can be reused instead of being destroyed. In this case it is not necessary to allocate this term again, but only to increase its reference counter. Although not explained in detail, in practice it was observed that during term rewriting this occurs often. The garbage collection is implemented by calling destroy on every term with a reference count of zero, as shown in the following pseudocode:

Algorithm 4 Garbage collection of terms

```
1: procedure COLLECT
2:     for t ∈ 𝒯′ do
3:         if reference-count(t) == 0 then
4:             destroy(t)
5:         end if
6:     end for
7:     collect-countdown ← |𝒯′|
8: end procedure
```

The *destroy* method recursively destroys all arguments which have a single reference, because that reference is the current function application.

Algorithm 5 Destroying individual terms

```
1: procedure DESTROY(t : 𝒯′)
2:     for p ∈ args(t) do
3:         if reference-count(p) == 1 then
4:             Destroy(p)
5:         end if
6:     end for
7:     𝒯′ ← 𝒯′ \ {t}
8:     deallocate(t)
9: end procedure
```

The *deallocate* function frees the heap memory used by this term. Note that this function prematurely destroys arguments with a single reference count. The reason for this is that the function `arg` is no longer defined for t after it has been deallocated.

3.3 Data structures

In this section the data structures in the implementation are described in detail.

3.4 Reference counting

The function symbol and term classes in the API internally have a shared reference to the maximally shared function symbol or term. To make sure that this function symbol or term can also be cleaned up at some point a so-called reference counter is introduced that keeps track of the number of references to it.

The shared reference is implemented by a class named `shared_reference` which consists of a reference and a reference counter. The invariant states that the reference counter is always equal to the number of instances that point to the same referred term. In C++ there are a number of operators to construct, move and copy classes. A move is an operator where an object is constructed from another object, but the other object can be left in an undefined state. The reference count invariant is satisfied by implementing these operations in the following way:

1. When a `shared_reference` instance is constructed from a reference its reference count is incremented by one.

2. When a `shared_reference` instance is copy-constructed its reference count is incremented by one.

3. When a `shared_reference` instance is move-constructed the reference count is kept the same, but the`shared_reference` instance that was moved from will become a null reference.

4. When a `shared_reference` instance is assigned to its current reference count is decremented by one. The reference count of the assigned reference is incremented by one.

5. When a `shared_reference` instance is move-assigned its current reference count is decremented by one. The `shared_reference` instance that was moved from will become a null reference.

6. When a `shared_reference` is destructed the reference count will be decremented by one.

For terms and function symbols all shared references are constructed with a default term or function symbol respectively. Whenever the reference count for a reference is equal to zero the referred to term can be cleaned up. In the case of function symbols this immediately triggers the destroy function.

3.5 Hash table

The terms and functions symbols are both stored in a set S which provides constant time insertion $\cup$, deletion $\setminus$ and contains $\in$ functions. It was mentioned that these operators could be implemented using a hash-table. In this section the implementation of that hash-table will be described.

The hash-table will store the set terms and function symbols. To be able to find, insert and delete elements we need to define a hash function and an equals function. The hash function is a unary function that takes a term (or function symbol) and returns a natural number. For terms the hash function is defined by combining the values of all the references. This results in the following pseudocode:

Algorithm 6 Hashing terms

```
1: procedure HASH(f(t_0, ..., t_n))
2:     hnr ← hash(f)
3:     for t ∈ t_0, ..., t_n do
4:         hnr ← combine(hnr, t)
5:     end for
6:     return hnr
7: end procedure
```

Notice that the term is constructed from references to the function symbol and the arguments. Instead of hashing these by value the hash can also be constructed from the value of their pointers which is often faster to compute in

practice. The combine function takes a hash number and a term and combines the result into a new hash number. For function symbols the hash function is the combination of a hash for its name and the arity.

The equivalence between terms can be determined by comparing the references of the function symbols and the references for each of its arguments. If any do not match the terms are not equivalent.

The hash table that was chosen is a closed addressing hash table with singly linked-list bucket.

An alternative hash table that had been tried is called hopscotch hashing. This is an open addressing, or closed hashing, which means that collisions are resolved by means of probing. Generally probing can become inefficient on a high number of collisions, which means a high load factor, because of the number of number of probes. Hopscotch hashing partially resolves this issue by having an upper bound on the number of probes.

As in most open addressing hash table it uses a continuous array of n so-called *buckets*. For each bucket a *neighborhood* is defined by a small collection of neighboring buckets. Let H be natural number, typically 32 or 64 bits, that defines the size of a neighborhood. This can then be visualized as virtual bucket of the current bucket and $H-1$ consecutive buckets. Hopscotch hashing maintains the invariant that each element is stored inside of this virtual neighborhood of the bucket that it hashes into. This invariant is maintained by moving elements inside of their neighborhood when the neighborhood of the inserted element is full. However, if this is not possible the hash-table is resized and all of the elements are rehashed.

The hopscotch hashing can be sped up by maintaining information on which entries in a neighborhood are already filled. As the neighborhood size is typically 32 or 64 bit that information can be stored in a single word where each bit indicates when that index is filled.

However, the downside of this approach was that by resolving collisions on top of other possible values the number of required equivalence checks grows. For terms checking equivalence has a complexity in the arity of the corresponding function symbol. This results in an about 10-15 percent slowdown compared to the simple linked list hashtable.

4 Optimizations

There are several optimizations that can be performed on this generic implementation.

4.1 Term pool per arity

Most function applications only consist of a small number of arguments. Let k be a constant such that a function application is *small* whenever its arity is less than or equal to k. We will use $\mathcal{T}^i$, for any natural number i, to denote a subset of $\mathcal{T}$ with function symbols that have an arity equal to i.

Now, instead of having one pool to store the finite subset $\mathcal{T}' \subseteq \mathcal{T}$, there will be a number of term pools:

- For small terms, k different pools to store $\mathcal{T}^0 \cup ... \cup \mathcal{T}^k \subseteq \mathcal{T}'$.

- A pool to store the set of terms $\mathcal{T}'' \subset \mathcal{T}'$ that do no occur in the other pools, i.e. $\mathcal{T}'' = \mathcal{T}' \setminus (\mathcal{T}^0 \cup ... \cup \mathcal{T}^k)$.

Note that all pools are disjoint, i.e. $\mathcal{T}^0 \cap ... \cap \mathcal{T}^k \cap \mathcal{T}'' = \emptyset$ and all pools combined store the whole subset, i.e. $\mathcal{T}^0 \cup ... \cup \mathcal{T}^k \cup \mathcal{T}'' = \mathcal{T}'$. The interface of the term pool remains unchanged. Internally this term pool uses the arity of a function symbol and the creation method to decide which pool will be used to create the term. For example calls to `create_appl(f)` will always go to the pool storing $\mathcal{T}^0$.

The effect of these changes is that the complexity for all loops over the arguments of function symbols with an arity up to and including k will become constant. The decision procedure has constant complexity as well.

Less theoretical this enables compiler optimizations such as loop unrolling. This optimization reduces the number of branch misses which increases the effective run-time performance as well. In practice this optimization performed well.

4.2 Alternative reference counting

If terms can be accessed by multiple threads at the same time it is necessary to introduce *atomic* reference counters. These atomic reference counters make sure that no data races can occur when incrementing or decrementing its value. However, in practice it was observed that atomic reference counters slow down the performance of creating terms by a factor of four. Here, we introduce a way to relax the reference counting for function application arguments.

Instead of considering function application arguments as a reference, they can be considered as a *weak reference*. A weak reference is a reference that does not change the reference counter of the referred to term. If the arguments of a function application are weak references then the number of reference count increments and decrements can be reduced. However, this means that the garbage collection has to be adapted as a reference count of zero does not necessarily imply that the term is not referred to via a function application.

The garbage collection that will be described here is often referred to as *tracing garbage collection*. Its basis is that all terms that are *unreachable* will be destroyed in a two-phase algorithm called *mark* and *sweep*.

Algorithm 7 Garbage collection of terms with weak references

```
1: procedure COLLECT
2:     Mark()
3:     Sweep()
4: end procedure
```

Consider the terms as a graph where the set of vertices are given by $\mathcal{T}'$ and the edges are given by a relation between function applications and their arguments. Every term with a reference count above zero is reachable by definition. The set of these terms is called the *root* set. Now from this root set, every term that is reachable by following the edges in the graph is reachable as well and should not be destroyed.

The set of reachable nodes can be determined by recursively marking the reachable nodes, this is implemented by the *mark* function as follows. Indicating

whether a term is reachable is done by coloring the nodes in this graph, which will be referred to as a mark. A term can be marked by using the function *set-mark* and the mark can be removed by using *remove-mark*. The function *is-marked* returns true if and only if that term has been marked.

Algorithm 8 Marking reachable terms

```
1: procedure MARK
2:     for t ∈ S do
3:         if reference-count(t) > 0 then
4:             Mark(t)
5:         end if
6:     end for
7: end procedure
```

The *Mark* function applied to a term will mark itself and all of its arguments recursively as reachable.

Algorithm 9 Marking an individual term

```
1: procedure MARK(t : T')
2:     if ¬is-marked(t) then
3:         set-mark(t)
4:         for p ∈ args(t) do
5:             Mark(p)
6:         end for
7:     end if
8: end procedure
```

Now, all terms that are not marked, and as such not reachable, can be deallocated and removed from the set of terms S. It should also remove the mark of marked terms such that the next garbage collection can be performed. This is implemented by the *Sweep* function as follows:

Algorithm 10 Sweeping terms that are no reachable

```
1: procedure SWEEP
2:     for t ∈ S do
3:         if ¬is-marked(t) then
4:             S ← S \ {t}
5:             deallocate(t)
6:         else
7:             remove-mark(t)
8:         end if
9:     end for
10: end procedure
```

After this procedure the memory used by all terms that are not reachable has been freed.

4.3 Null term

Instead of introducing a default term that is used whenever the `shared_ptr` is default constructed we can introduce an actual *null* reference. This change breaks the invariant that any term always has an existing shared reference. That requires the `shared_ptr` to check whether is has a valid reference before trying to adapt its reference counter. The advantage of this change is a reduction in the number of reference count adaptations that have to be performed on the default term.

For the shared reference a function called *defined* is introduced that checks whether the shared reference is not equal to *null*.

In the 1394 protocol state space generation with the options `--cached` this optimization reduced the number of reference count increment and decrements by ten percent. The run-time performance was almost unaffected. However, in the case of atomic reference counters this had about 11 percent speed-up on the resulting state space generation.

4.4 Block allocator

The `allocate`, `construct` and `deallocate` functions are part of the allocator interface as defined in the STL. The `construct` function calls `allocate` and then the constructor of the object. The default allocator is wrapper around `new` and `delete`. These functions are a wrapper around `malloc` and `free`. Although no specific implementation is required for these functions they typically call kernel functions to obtain memory from the operating system for each call. In the Linux kernel a slab allocator is used that has no internal fragmentation for powers of two and a page size is typically 4KB. Compared to that, a single term of arity three only requires 40 bytes, which is not a power of two and also small.

The amount of terms allocated is typically very large and the exact size in bytes of each term is known beforehand. Therefor a useful approach is to allocate a block of memory, for example 4KB, and return references to slots in this block. A slot is space in a block that is large enough to store one element.

The block allocator has a single linked list of blocks called *blocks*. Each block can store `ElementsPerBlock` number of elements, which is equal to the number of slots in a block. The allocator stores a `currentIndex` of the first slot in the block that has never been used. To keep track of slots that have been deallocated in the mean time a *freeList* is used.

Algorithm 11 Allocate

```
1: procedure ALLOCATE
2:     if ¬freeList.empty() then
3:         return freeList.pop_front()
4:     end if
5:     if currentIndex >= ElementsPerBlock then
6:         blocks ← blocks.push_front()
7:         currentIndex ← 0
8:     end if
9:     firstBlock ← blocks.front()
10:    slot ← firstBlock[currentIndex]
11:    currentIndex ← currentIndex + 1
12:    return slot
13: end procedure
```

The deallocation of a term is the same as adding it to the freelist. This can be efficiently done by changing its next reference to `firstFreeSlot` and setting the first free slot to the new head of the freelist.

Algorithm 12 Deallocate

```
1: procedure DEALLOCATE(Reference r)
2:     freeList.push_front(r)
3: end procedure
```

Finally it would be useful to be able to erase blocks that don't store any elements. For this purpose the *consolidate* function was introduced, see Algorithm 13. First, recall that all elements in the free list can be visited by following the next reference starting from the *firstFreeSlot*. In the first part of the algorithm all elements of the free list are marked by a special value $\top$, which is a value that should not occur in any slot before consolidate is called, in lines 2 to 6. The next part is to reconstruct the *freeList* from the free elements in the block. Line 6 checks whether the block only contains elements that are free. If that is the case then this block can be erased from the list of blocks. This check can be efficiently implemented by checking it in the for loop over all slots in the block.

Algorithm 13 Consolidate

```
1: procedure CONSOLIDATE
2:     for slot ∈ freeList do
3:         slot ← ⊤
4:     end for
5:     for block ∈ blocks do
6:         if ∀slot ∈ blockslot = ⊤ then
7:             blocks.erase(block)
8:         else
9:             for slot ∈ block do
10:                if slot = ⊤ then
11:                    freeList.push_back(slot)
12:                end if
13:            end for
14:        end if
15:    end for
16: end procedure
```

An optimization that can be performed in the block allocator it to store the *freeList* inside the slots of the blocks directly. This can be implemented as follows. The first slot in this free list is pointed to by *firstFreeSlot*. The *freeList* can then be stored by storing the *next* reference, which contains the reference to the next element in the list, in place of the slots. We define the invariant that all slots that are reachable by following next references after `firstFreeSlot` are part of the freelist. This means that the *freeList* is empty when the *firstFreeSlot* points to null. The *push_front* operation can be achieved by letting the *firstFreeSlot* point to the reference that was pushed into the *freeList* and setting the next reference to the head of the *freeList*. Iteration over the *freeList* can be achieved by following the *next* reference until it is null.

In practice the consolidate function is only able to erase relatively few blocks.

4.5 Alignment

In a typical processor design the accesses to main memory are cached through a number of increasingly larger, but slower caches. A *cache line* is a block of consecutive memory units that a processor fetches from main memory at once and stores in a cache. When a processor wants to load or store an address from main memory it will actually load the whole block that contains this address. The start of each block is also also *aligned* to the width of the cache line. This means that its first address is a multiple of the cache line width.

In a modern processor the typical cache line has a width of 64 bytes. This means that every memory access will fetch 64 consecutive bytes. The fetches are also aligned to 64 bytes which means that the lowest address fetched is a multiple of 64. The optimization idea was to store the terms in memory in such a way that accesses to its members does not cross cache line boundaries. This would otherwise require multiple cache lines to be fetched. However, benchmarks indicated that this did not have a good impact on performance and it does carry a potential memory increase, which is also undesired.

4.6 Integral terms

In term rewriting with substitutions it is required to have a mapping from terms representing variables to their value. This is a mapping between terms. For this purpose a hash map can be used. This has constant complexity, but also a small overhead for computing the hash number. Better performance can be achieved by storing this mapping in an array. A value will then be used to index a position in this array to look up the term for its value. This index has to be dense and a unique index can be generated when required.

In practice this index value is stored as last argument in the function application. Consider the following variable term $var(a, b, c)$, whenever 5 would be the next index available this term becomes $var(a, b, c, 5)$ and in the substitution mapping array the index 5 will return the value for this variable. To be able to store natural numbers as argument to a function application the natural numbers will become terms as well. Formally the set of terms $\mathcal{T}$ will be extended as follows:

- If t is a *constant* term, i.e. $t \in \mathbb{N}$ then $t \in \mathcal{T}$.

The API is extended in the following way:

$c \in \mathbb{N}, c \in \mathcal{T}$	create_int(n : $\mathbb{N}$)

Figure 3: Extension to the API with constant terms

The `value` function is defined that maps constant terms to the natural number value of which it was constructed.