A Thread-Safe Term Library

A Thread-Safe Term Library
(with a New Fast Mutual Exclusion Protocol)

Jan Friso Groote(✉), Maurice Laveaux, and P. H. M. van Spaendonck

Department of Mathematics and Computer Science,
Eindhoven University of Technology, Eindhoven, The Netherlands
{J.F.Groote,M.Laveaux,P.H.M.v.Spaendonck}@tue.nl

Abstract. Terms are one of the fundamental mathematical concepts in computing. E.g. every expression characterisable by a context free grammar is a term. We developed a *thread-safe* Term Library. The biggest challenge is to implement hyper-efficient multi-reader/single-writer mutual exclusion for which we designed the new *busy-forbidden protocol*. Model checking is used to show both the correctness of the protocol and the Term Library. Benchmarks show this Term Library has little overhead compared to sequential versions and outperforms them already on two processors. Using the new library in an existing state space generation tool, very substantial speed ups can be obtained.

Keywords: Term library · Mutual exclusion · Thread-safe · Model checking

1 Introduction

A term is a common mathematical structure. Many concepts can be represented as terms, such as programs, specifications and formulas. Many operations in computing are term transformations, such as compilation. In computer science a term is a far more commonly used concept than structures such as arrays, lists or matrices. This makes it remarkable that terms are not a standard data structure in common programming languages such as `C++` and Java.

To our knowledge the first term library stems from the realm of program transformations. In [2,4–6,17] an ATerm library of so called *annotated terms* has been proposed, which contains terms with meta information. Stripping away all bells and whistles from this ATerm format, a very plain and elegant term data structure remains.

Our terms are defined in the standard way. We start out with a given set of function symbols F where each function symbol $f \in F$ has an arity ar_f. Each constant function symbol, i.e. with arity 0, is a term. Given a function symbol $f \in F$ with $ar_f > 0$, and terms $t_1, \ldots, t_{ar_f}$, the expression $f(t_1, \ldots, t_{ar_f})$ is also a term. These are the only two ways to construct a term.

Supported by projects 612.001.751 (NWO, AVVA) and 00795160 (TTW, MASCOT).

T. Margaria and B. Steffen (Eds.): ISoLA 2022, LNCS 13701, pp. 422–459, 2022.
https://doi.org/10.1007/978-3-031-19849-6_25

As an example, we provide terms where some constants represent variables. We can have function symbols $\{0, 1, x, y, +\}$ and have terms $0 + 1$, $x + 1$ and $x + y$. The 'constants' x and y allow for different operations than the constants 0 and 1, as it is natural to define a substitution operation for the constant x, whereas that would be less natural for the constant 0. In a similar way, terms with binders can be represented. For instance, in the term $\lambda x.t$ the λ is just a binary function symbol and the first subterm is the variable x.

As in the ATerm library, terms are stored in a maximally shared way. Once created, terms remain as stable structures in memory until they are garbage collected. This leads to a smaller memory footprint, because equal terms are only stored once. Comparing terms is also computationally cheap, as two terms are equal iff they occupy the same address in memory. Also note that handing a term over to another thread is also cheap, as only the address of the term needs to be transferred. This avoids serialising and deserialising terms as done in [3]. A disadvantage is that subterms cannot be replaced. If a subterm needs to be changed, the whole surrounding term must be reconstructed.

With a steadily increasing number of computational cores in computers, it is desirable to have a parallel implementation of a term library. As terms have a tree-like structure, one would expect concurrent tree algorithms, as provided by the EXCESS project [30] or the PAM library [28], to be a useful solution. However, these tree libraries concentrate on manipulating the trees themselves, by adding and removing nodes, and rebalancing when required. This would not allow maximal sharing of terms, which have to be static structures in memory.

Early attempts to create a thread-safe term library led to intriguing wait-free algorithms [9,10,16]. The assumption was that thread synchronisation was the root cause of performance issues, and this is avoided when algorithms are wait-free. But this did not turn out to be entirely true. As the operations to create, inspect and destroy terms occur frequently and are computationally very cheap, a thread-safe implementation allows for hardly any overhead. Wait-free algorithms are intricate and their overhead is deadly for performance in this case. The same applies to the introduction of mutex variables surrounding construction, inspection and deletion of terms.

Although the need and advantages of having terms that can be accessed by multiple threads have already been stressed in the original publications, it turns out to be hard to make a thread-safe term library that is competitive with sequential implementations. This is most likely the reason that no thread-safe term libraries exist, except for a non-published Java implementation [20].

In this article we present a thread-safe term library that is competitive with sequential term libraries. We first observe that with some minor adaptations, *i.e.*, essentially introducing a Treiber stack [29] in a hash table, inspection and construction of terms can happen concurrently. Secondly, we note that garbage collection on the one hand, and construction/moving/copying of terms on the other hand must be mutually exclusive, and construction happens far more often than garbage collection.

Therefore, we require a mutual exclusion algorithm with behaviour of a readers-writer lock [26], where construction of terms can happen simultaneously (=readers), and garbage collection (=writer) must be done in isolation. However, standard readers-writer locks are too expensive. We designed the *busy-forbidden protocol* that employs this asymmetric access pattern as well as the cache structure of modern processors. Obtaining access to construct a term only requires access to two bits, virtually always available in the local cache of the current processor. Besides this, we developed thread-safe term protection mechanisms, either using atomic operations for reference counting, or by employing explicit thread-local protection sets.

Experiments show that the new Term Library scales well and for practical tasks it is already beneficial when only two processors are available. The solution with a standard readers-writer lock and especially the Java implementation are substantially slower than our implementation with the busy-forbidden protocol.

The correctness of thread-safe implementations is subtle. Therefore, we use the mCRL2 model checking toolset [13] to design both the busy-forbidden protocol and the Term Library, and prove their correctness properties, before implementation. This turned out to be very effective, as we did not have to struggle with obscure faults due to parallel behaviour in the algorithm. It is intended that the new thread-safe Term Library will form the heart of the new release of the mCRL2 toolset. The currently existing early prototype already achieves speed ups of a factor 12 on 16 processors for a computationally intensive task, namely state space generation, which is more than just promising.

2 The Term Data Structure

In [4,5] a term library has been proposed. A term is a very frequently used concept within computer science. The original motivation for terms as a basic data structure came from research in software transformation [6,17]. The model checking toolset mCRL2 uses terms to represent all internal concepts, such as modal formulas, transition systems and process specifications [13].

2.1 The External Behaviour of the Term Library

Terms are constructed out of *functions symbols*, or for short *functions*, from some given set F. Each function $f \in F$ has a number of arguments ar_f, generally called the *arity* of f. A function symbol with arity 0 is called a *constant*.

Definition 1. *Let F be a set of function symbols. The set of terms T_F over F is inductively defined as follows:*

$$\text{if } f \in F, f \text{ has arity } ar_f \text{ and } t_1, \ldots, t_{ar_f} \in T_F, \text{ then } f(t_1, \ldots, t_{ar_f}) \in T_F.$$

Simple numeric expressions are typical examples of terms. The function symbols are $0, 1, 2, 3, +, *$ where $0, 1, 2, 3$ are constants and $+$ and $*$ have arity 2. An example of a term as a tree structure is given in Fig. 1.

The term library in [4,5] allows to annotate terms, hence the name ATerm, but we do not use this feature. This original ATerm proposal also supported special terms representing numbers, strings, lists and even 'blobs' containing arbitrary data. We made our own implementation of a term library where besides terms as defined in Definition 1, there are also facilities for lists and 64-bit machine numbers. As these are in many respects the same as terms constructed out of function symbols, we ignore lists and numbers in this exposition.

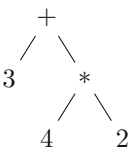


Fig. 1. The tree for $3 + 4 * 2$.

From the perspective of a programmer terms are immutable maximally shared tree structures in memory. This means that if two (sub)terms are the same, they are represented by the same address in memory. The term library provides essentially the following limited set of operations on terms:

Create. Given a function symbol f and terms $t_1, \ldots, t_{ar_f}$ construct a term $f(t_1, \ldots, t_{ar_f})$. This operation can fail when there is not enough memory.
Destroy. Indicate that a term t will not be accessed anymore by this thread. Terms that are not accessed by any thread must ultimately be garbage collected.
Copy/move. Move or copy a term. This essentially means move or copy the address of the term.
Argument. Obtain the i-th subterm t_i of a term $f(t_1, \ldots, t_{ar_f})$.
Function. Obtain the function symbol f of a term $f(t_1, \ldots, t_{ar_f})$.
Equality. For terms t and u determine whether t and u are equal. Note that due to maximal sharing this operation only requires constant time.

Due to the immutable nature of terms in memory it is not possible to simply replace a subterm of a term. If a subterm must be changed, the whole surrounding term must be copied. On the other hand terms are very suitable for parallel programming. Threads can safely traverse protected terms in memory as the treads can be sure that these terms will not change.

By storing terms as maximally shared trees, the only non trivial operations on terms are the creation of a new term and the destruction of an existing term. Given a function symbol f and subterms $t_1, \ldots, t_{ar_f}$ it must be determined whether the term $f(t_1, \ldots, t_{ar_f})$ already exists. This is done using a hash table. If the term already exists, this term is returned. If not, a new term node labelled with f pointing to the subterms $t_1, \ldots, t_{ar_f}$ must be made.

The typical usage pattern of terms is that they are visited very often obtaining arguments or function symbols. Creation of a term is also a very frequent operation, where in the majority of cases a term is created that already occurs in the hash table. Only rarely a garbage collect is taking place.

2.2 Behavioural Properties of the Term Library

The Term Library guarantees the following properties, checked using model checking, see Sect. 4.

1. A term and all its subterms remain in existence at exactly the same address, with unchanged function symbol and arguments, as long as it is not destroyed.
2. Two stored terms t_1 and t_2 always have the same non-null address iff they are equal.
3. Any thread that is not busy creating or destroying a term, can always initiate the construction of a new term or the destruction of an owned term.
4. Any thread that started creating a term or destroying a term, will eventually successfully finish this task provided there is enough memory to store one more term than those that are in use. But it is required that other threads behave fairly, in the sense that they will not continually create and destroy terms or stall other threads by busy waiting.

Note that the properties above imply some notion of garbage collection in the sense that if a thread makes and destroys terms, and these are not garbage collected, at some point no new terms can be created due to a lack of memory and in that case property 4 above would be violated.

2.3 The Implementation of the Thread-Safe Term Library

Terms are implemented in the Term Library by storing them in a hash table. Whenever a term with function symbol f and arguments $t_1, \ldots, t_{ar_f}$ is created, the hash table is used to find out whether $f(t_1, \ldots, t_{ar_f})$ already exists. If yes, its current address is returned. If no, a new term $f(t_1, \ldots, t_{ar_f})$, is inserted in the hash table and its address is returned.

Another possible solution would be to use a CTrie [25] instead of the hash table. However CTries main advantage, memory conservation, over performance, makes it less suitable for our Term Library, which must be suitable to deal with huge numbers of term manipulations in short time spans.

Terms in our Term Library can be constructed and accessed in parallel. When a thread created a term, this term and all its subterms are immutable and stored at fixed addresses in memory, and this means that any term can be accessed safely by all threads that have not destroyed the term.

We have two ways to implement garbage collection in the thread-safe Term Library, namely reference counting and the use of protection sets, which ensure that non-destroyed terms remain in memory. Garbage collection is performed by a single thread. Note that mark-and-sweep algorithms exist where creation and destruction can be done simultaneously [10] but these are very complex. As garbage collection is relatively fast, such advanced algorithms are not necessary.

In reference counting, each term has a reference count that is incremented by one whenever a term is created or copied, and decremented by one if a thread drops a reference to the term. Terms that are not in use anymore have a reference count of zero and can be garbage collected. This can easily be performed by visiting all terms, which are stored in traversable structures.

An alternative is to use term protection sets. Whenever a term is stored at some address, this address is stored in a separate protection set, locally maintained by each thread. When the address is not used anymore for a term, it

is removed from the set. As every address can only be stored once, a simple hash table suffices to implement the protection set. Garbage collection consists of marking all terms reachable via some protection set, and removing all others.

In the parallel setting changing reference counts or inserting/deleting addresses in protection sets must be sequentially consistent meaning that they cannot be rearranged in the programs. Changing reference counts must be atomic and can lead to cache contention as the reference counts are accessible by all threads. Operations on the protection sets are far more complex than changing a reference count, but they are always local in a thread, and depending on the style of programming need to be executed far less often than changing a reference count. From the benchmarks we derive that protection sets are preferable.

If we only create terms, this can be done in parallel as well. We use a dedicated hash table with a bucket list in the form of a linked list to check whether a term already exists. If the term does not exist, it is added using a compare and swap operation to the bucket list of the appropriate entry of the hash table. If in the mean time another thread creates the same term, the compare and swap fails, informing the thread that it has to inspect the hash table again to find out whether the term came into existence. This is Treiber's stack, which is sufficient since terms are not simultaneously deleted from the bucket lists. Deletion only occurs during garbage collection, and during garbage collection no new terms are allowed to be constructed.

Accessing terms during garbage collection and rehashing is perfectly safe. But it is not allowed to create or copy terms while garbage collection or rehashing is going on. This requires a mutual exclusion protocol where either multiple threads can create and copy terms simultaneously, which we call the *shared* tasks, or a single thread can be involved in garbage collection or rehashing, which is called the *exclusive* task.

This is the same as a readers-writer lock [26] where multiple readers or at most one writer can access a shared resource. Reading is the shared task, and writing is exclusive. As we observed that creating and copying terms is done very frequently compared to garbage collection, shared access must be cheap and exclusive access can be expensive. Most standard readers-writer locks require at least one access to a common mutex variable for shared access which is so costly that parallel implementations based on the readers-writer lock run on multiple processors failed to outperform the sequential implementation. This observation is supported by the benchmarks. We developed a completely new protocol, called the *busy-forbidden* protocol serving our needs, which is described in the next section.

Using the busy-forbidden protocol, a compare and swap to insert terms in bucket lists for the hash table, the implementation of thread-safe Term Library is pretty straightforward but delicate. Table 1 contains the code for creating and destroying terms. In this code `enter_shared`, `leave_shared`, `enter_exclusive` and `leave_exclusive` are part of the busy-forbidden protocol described in the next section. The function `h` is a hash function that takes a function symbol f, and subterms $t_1, \ldots, t_n$, and calculates a possibly non-unique hash. The func-

Table 1. Pseudocode description of the thread-safe Term Library.

```
1  create(thread p, symbol f, subterms t_1, ..., t_n)
2     enter_shared(p);
3     hash := h(f, t_1, ..., t_n);
4     bucket := buckets[hash];
5     t := insert(bucket, f, t_1, ..., t_n);
6     protect(p, t);
7     leave_shared(p);
8     return t;
9
10 insert(bucket b, symbol f, subterms t_1, ..., t_n)
11    old_head, node := b.top;
12    do
13       if node.head represents f(t_1, ..., t_n)
14          return node.head;
15       node := node.tail;
16    while (node ≠ NULL);
17    t := construct f(t_1, ..., t_n);
18    if not cmpswap(b.top, old_head, Node(t, old_head))
19       destruct t;
20       return insert(b, f, t_1, ..., t_n);
21    return t;
```

```
destroy(thread p, term t)
   unprotect(p);
   possibly do GC(p)

GC(thread p)
   enter_exclusive(p);
   forall t ∈ hash_table
      if not protected(t)
         remove t;
   leave_exclusive(p);
```

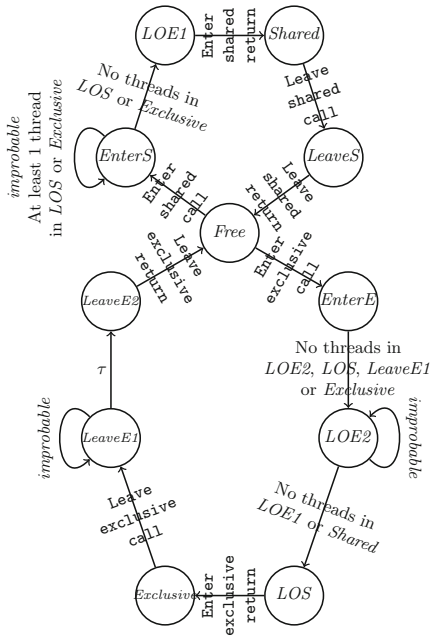
tions `protect`, `unprotect` and `protected` refer to the protection mechanisms described earlier, in which `protected`(t) will return true if and only if the term t is protected by some thread. Besides this, each bucket b in the hash table contains an atomic pointer $b.top$ that allows atomic loads and an atomic compare-and-swap operation `cmpswap`, which returns true if and only if successful. The call `GC`(p) stands for doing a garbage collect by thread p.

Using an mCRL2 model of the behaviour of the Term Library, the behavioural properties mentioned in Sect. 2.2 have been model checked. This is described in Sect. 4.

3 The Busy-Forbidden Protocol

The busy-forbidden protocol is of independent interest. This protocol guarantees that at most one thread can be in state *Exclusive* and if a thread is in state *Exclusive*, no thread is in state *Shared*, and vice versa, if there are threads in state *Shared*, then there is no thread in the state *Exclusive*. It behaves in a similar way as a readers-writer lock [26], called a shared mutex in `C++`.

The busy-forbidden protocol is designed for the situation where shared access is frequent whereas exclusive access is infrequent.



LOE1	There are no threads in or able to enter *Exclusive*.
Shared	Shared access. No concurrent access to *Exclusive* possible.
EnterS	Entering shared.
LeaveS	Leaving shared.
Free	The thread is outside any exclusive or shared section.
LeaveE2	Leaving exclusive.
EnterE	Entering exclusive.
LeaveE1	Leaving exclusive. No threads in or able to enter *Exclusive*.
LOE2	There are no threads in or able to enter *Exclusive*.
Exclusive	Exclusive access. There are no threads in or able to enter *Exclusive* or *Shared*.
LOS	No threads in or able to enter *Exclusive* or *Shared*.

Fig. 2. The external behaviour of the busy-forbidden protocol.

3.1 The External Behaviour of the Busy-Forbidden Protocol

We first look at the external behaviour of this protocol. As indicated above, threads can request for shared or exclusive access by calling one of the two functions `enter_shared` and `enter_exclusive`. The functions starting with `leave` are used to indicate that access is no longer required.

We make the external behaviour more precise by modelling it as a state automaton, actually obtained by the specification in mCRL2 used for verification. From the perspective of a single thread, the behaviour is depicted in Fig. 2. The calls are modelled by actions `Enter`/`Leave` `shared`/`exclusive` `call`. Returning from the function is modelled by actions ending in `return`.

The centre state, marked *Free*, indicates that the thread is not involved in the protocol. It is outside the shared and exclusive sections. Following the arrows in a clockwise fashion, a thread obtains access. In the state *EnterS* the thread requested shared access, and it will get it when there are no threads in the states *LOS* or *Exclusive*. From the figure it is quite easy to see that the protocol indeed satisfies the mutual exclusion constraints mentioned above.

We went to great length to ensure that the behaviour of Fig. 2 for multiple threads is divergence-preserving branching bisimilar to the implementation below [11,12]. This equivalence is equal to branching bisimulation, but it does not remove τ-loops, *i.e.*, loops of internal actions. It preserves not only safety

but also liveness properties, and allows us to use this specification to verify the Term Library.

The loop at *EnterS* occurs typically when another thread is in state *Exclusive* for a lengthy period. The loop at *LOE2* occurs when another thread is in *Shared* and refuses to leave. The loop in *LeaveE1* is required to obtain a concise equivalent external behaviour. When the busy protocol is used as intended, *i.e.*, threads only use common accesses for a short time, and the implementation uses the right internal scheduling, these loops rarely occur. They are therefore marked *improbable*.

3.2 The Implementation of the Busy-Forbidden Protocol

The code for entering and leaving the exclusive sections is described in Table 2. The busy-forbidden protocol is implemented by assigning to each thread two atomic flags, called *busy* and *forbidden*. The flag *busy* indicates that the current thread is in its shared section and can only be written to by this thread. The flag *forbidden* indicates that some thread is having exclusive access.

Table 2. Pseudocode description of the busy-forbidden protocol.

```
1  enter_shared(thread p)
2      p.busy := true;
3      while p.forbidden
4          p.busy := false;
5          if mutex.timed_lock()
6              mutex.unlock();
7          p.busy := true;
8
```

```
1  enter_exclusive(thread p)
2      mutex.lock();
3      while exists thread q with
4              ¬q.forbidden
5          select thread r
6          r.forbidden := true;
7          if r.busy or sometimes
8              r.forbidden := false;
```

```
1  leave_shared(thread p)
2      p.busy := false;
3
4
5
6
7
8
9
```

```
1  leave_exclusive(thread p)
2      while exists thread q with
3              q.forbidden
4          select thread r
5          usually do
6              r.forbidden := false;
7          sometimes do
8              r.forbidden := true
9      mutex.unlock();
```

Besides the flags there is one generic mutual exclusion variable, called *mutex*. The variable *mutex* can not only be locked and unlocked, but also provides a timed lock operation *timed_lock*(). It tries to lock the mutex, and if that fails after a certain time, it returns false without locking it. The timed lock is only important for performance, and can be replaced by a wait instruction or even be omitted altogether.

When entering the shared section, a thread generally only accesses its own *busy* and *forbidden* flags as *forbidden* is almost always false. These flags are

only rarely accessed by other threads and therefore virtually always available in the local cache of the processor executing the thread. In the rare case when the *forbidden* flag is set, this thread backs off using *mutex* to try again later. In principle the while-loop can be iterated indefinitely, giving rise to the internal loop in state *EnterS* in the specification. Leaving the shared section consists of only setting the *busy* flag of the thread to false.

Accessing the exclusive section is far more expensive. By using *mutex*, mutual access to the exclusive section is obtained. Subsequently, the *forbidden* flag for each thread p is set to true, unless the *busy* flag of thread p is set, as in this case the *forbidden* flag must be set to false again.

There is a non immediately obvious scenario where one thread refuses to leave the shared section, and two other threads p_2 and p_3 want to access the shared and exclusive section, respectively. Thread p_3 cannot obtain exclusive access, but hence should not indefinitely block shared access for p_2. Hence, p_3 must set the *forbidden* flag of p_2 to false if *busy* of p_1 is true.

Without the **sometimes** part, which represents an arbitrary heuristic which only rarely holds, the implementation is not divergence-preserving bisimilar to the specification, as reading $r.busy = false$ in line 9, once all other forbidden flags have been set, leads to a state without an internal loop, which does not occur in the specification. Without the **sometimes** part, a matching specification would become substantially more complex exhibiting exactly when each *forbidden* flag is set, rendering the specification far less abstract and hence making it less useful.

When leaving the exclusive section a thread resets all *forbidden* flags of the other threads. If this is done in a predetermined sequence the divergence-preserving branching bisimilar external behaviour becomes very complex, as this sequence has an influence on the precise sequence other threads can enter the shared section. By resetting and sometimes even setting the *forbidden* flag, a comprehensible provably equal external behaviour is obtained, although it leads to another τ-loop in the specification. Practically, re-resetting the flag is hardly ever needed, certainly not for the Term Library. However, it is interesting to further investigate the optimal use of the timing of *mutex* in `enter_shared`, as well as the optimal rate of occurrence of the **sometimes** instructions for generic uses of the busy-forbidden protocol.

We modelled the specification and implementation of the busy-forbidden protocol in mCRL2 (see Sect. 4) and proved them divergence-preserving branching bisimilar.

3.3 Behavioural Properties of the Busy-Forbidden Protocol

As an extra check we also formulate a number of natural requirements that should hold for this protocol. These requirements have been verified by formulating them as modal properties.

1. There should never be more than one thread present in the exclusive section.
2. There should never be a thread present in the exclusive section while one or more threads are present in the shared section.

3. When a thread requests to enter the shared section, it will be granted access within a bounded number of steps, unless there is another thread in the exclusive section.
4. When a thread requests to enter the exclusive section, it will be granted access within a bounded number of steps, unless there is another thread in the shared or in the exclusive section.
5. When a thread requests to leave the exclusive/shared section, it will leave it within a bounded number of steps.
6. A thread not in the exclusive or shared section can instantly start to enter the exclusive or shared section.

For properties 3, 4, and 5 granting access and leaving can be indefinitely postponed if other threads are entering and leaving exclusive and shared sections, or when other threads are in the while loops, continuously writing forbidden and busy flags. This means that the algorithm relies on fair scheduling of threads.

3.4 Existing Readers-Writer Locks

Common readers-writer locks, such as `std::shared_mutex` in `C++17` in MSVC, use a mutual exclusion variable when entering and leaving the shared/reader section. This leads to poor scalability and is one of the reasons why the usage of readers-writer locks is often discouraged [7].

The readers-writer lock by Mellor-Crumney and Scott [23] reduces resource contention by using a counter to keep track of the amount of current readers and introducing a queue system in which threads only have to notify the thread next in line when they leave the lock. This lock is further improved by Krieger et al. [19] by reducing the amount of shared variables to a single pointer and using a double-linked list instead of a queue such that reader threads can leave the lock without having to wait for neighbouring readers to also be done reading. However, the single shared pointer needs to be updated using a costly compare-and-swap operation every time a thread enters the lock, which becomes a bottleneck when multiple threads try to enter at the same time.

Lev et al. [21] provide several readers-writer lock algorithms aimed at improving scalability by significantly reducing resource contention through the use of a tree-like data structure called C-SNZI. Lev et al. show that their algorithms outperform other readers-writer locks when the majority of accesses, *i.e.*, $\geq 80\%$, are read accesses. Concurrent read-accesses however still have resource contention with a 100% read workload, whereas the busy-forbidden protocol has none. Thus, this implies that the busy-forbidden protocol outperforms the C-SNZI based algorithms for our use case.

3.5 Performance of the Busy-Forbidden Protocol

We have implemented the busy-forbidden protocol in `C++` and assess its scalability compared to that of the `std::shared_mutex` by having threads repeatedly enter and leave the shared/reader section or the exclusive/writer section. Each

thread uses a random number generator to decide on which section to enter and then leave, with a preset probability of 99.99% of a thread deciding to enter and leave the shared/reader section. This probability was chosen as it corresponds to that of a typical use case such as state space generation.

Figure 3 shows the wall clock time of *#threads* threads entering and then leaving a random section 10^9 times per thread. The values displayed are the averages of 5 different runs. The measurements were taken on an Intel i7-7700HQ processor and the `C++` code was compiled using the MSVC19 compiler with the `-O2` flag enabled. The wall clock time of `std::shared_mutex` for more than 4 threads is omitted from the graph as it is too large to nicely display.

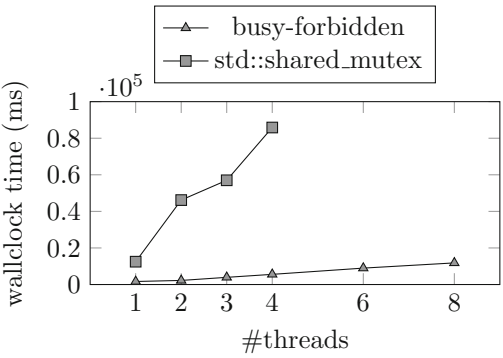


Fig. 3. Readers-writer lock benchmarks.

We observe in Fig. 3 that the busy-forbidden protocol performs significantly better than the `std::shared_mutex` and costs only a minimal amount of overhead. This can also be seen in Fig. 5 (a), discussed in Sect. 5, in which the busy-forbidden protocol and `std::shared_mutex` are used in combination with our implementation of the parallel Term Library.

4 Modelling and Verifying the Algorithms

As parallel algorithms are hard to get correct, we made models of the busy-forbidden protocol, of both specification and implementation, and the thread-safe Term Library in the process modelling language mCRL2 and verified the properties by formulating them in the modal mu-calculus [13]. The specification model is a direct reflection of the external behaviour shown in Fig. 2. The resulting implementation models are a direct reflection of the pseudocode in Table 1 and 2. The formulas are a one to one translation of the requirements listed in this article. For this reason, and for the reason of space, the models and formulas, are not included in this article[1].

Due to the nature of model checking, we only verify the models for finite instances. We repeatedly found that when protocols or distributed systems are erroneous, the problems already reveal themselves in small instances [14]. Used in this way, model checking is so efficient that it can effectively be used within the workflow of constructing software. The busy-forbidden protocol was modelled and proven, before implementation commenced, and we did not run into any problem with it during implementation.

[1] All models and formulas can be found in Appendices B and C, respectively.

A general equivalence proof has since been given for the specification and implementation of the busy-forbidden protocol [27], using an extension of the Cones and Foci method [8,15].

The correctness of the protocol and library have not been proven in general for any number of threads and terms. Unfortunately, we do not know of any effective method to prove modal formulas on models with a complexity such as ours, either automatically or manually, for any number of threads and terms, and consider this an important direction of research.

The model of the busy-forbidden protocol does not include the *mutex.timed_lock*() statement as it is only important for performance. The **sometimes** keywords are modelled as non-deterministic choices. The specification and implementation are proven to be divergence preserving branching bisimulation equivalent, for up to 7 concurrent threads.

We transformed the six requirements discussed in Sect. 3.3 into modal logic formulas, and verified them both on the specification and the implementation, although the latter was not really necessary due to their equivalency. The equivalence and properties were verified, both on the specification as well as on the implementation model, for up to 7 threads. We uncovered a number of issues and obtained various insights while doing the verification for 2 and 3 threads. The verification with more threads, although increasingly time consuming, did not lead to any additional insight.

The model is primarily concerned with the thread-safe creation and garbage collection of terms, and therefore the typical term structure, where terms contain subterms, is also not part of the model, and as such only uses terms with arity 0. We use the equivalent specification model of the busy-forbidden protocol instead of its implementation model, as it is significantly smaller. Furthermore, the model of the Term Library does not include buckets or a hashing function. Instead the hash table is modelled as a simple associative array, with atomic *contains* and *insert* operations.

The four properties discussed in Sect. 2.2 are also translated into modal logic and verified for finite instances. We have verified these properties for up to 3 threads, using 3 different terms and 4 possible addresses, giving us reasonable certainty that the thread-safe Term Library works as intended. We were unable to verify our properties on larger state spaces as they became too big to verify automatically. For example the state space of the aforementioned setup with 4 threads instead of 3 has 129 billion states.

5 Performance Evaluation

We have implemented a sequential and a thread-safe version of the Term Library. Both of these implementations are almost identical except for the synchronisation primitives added to the thread-safe version where necessary, including the busy-forbidden protocol. Furthermore, we have implemented both reference counting and address protection sets as garbage collection strategies in both implementations for comparison. We compare these with the sequential ATerm

library as used in the mCRL2 toolset [13] and with a thread-safe Java implementation [20] of the Term Library used in tools such as Spoofax [18], which was the only other thread-safe term library that we could find. All reported measurements are the average of five runs with an AMD EPYC 7452 32-Core processor, unless stated otherwise.

The results are listed in the plots in Figs. 4 and 5. In these plots the y-axis indicates the wall clock time in seconds and the x-axis the number of threads (*#threads*). The triangles are the thread-safe reference count implementation and the squares the thread-safe set protection implementation. For the sequential versions we have circles for the reference count version, diamonds for the protection set version and plusses for the original implementation. The results for the sequential implementations are extended horizontally for easier comparison. Finally, the dashed line indicates the thread-safe Java implementation and the dotted line is our thread-safe implementation where the busy-forbidden protocol has been replaced by a `std::shared_mutex`. This last implementation uses protection sets.

In Fig. 4 we report three experiments, one per row, designed to obtain insight in how the new thread-safe library performs for specific tasks. In the left column all threads access the same shared term, whereas in the right column each thread operates on its own term, but these distinct terms are stored in common data structures and accessed via the hash table common for all threads.

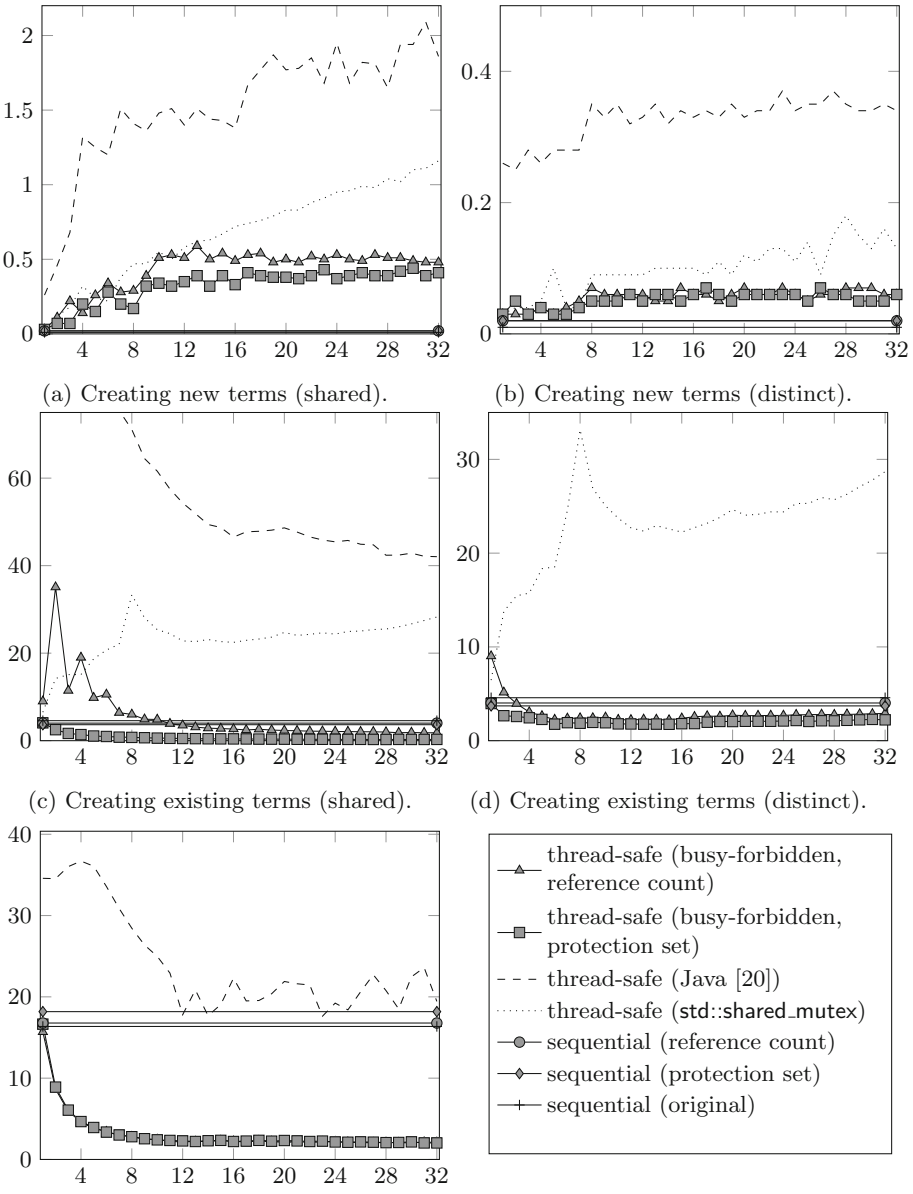
In Fig. 4 (a) we measure how expensive it is to create a term in parallel. The threads create a term $t_{400\,000}$ defined as follows. The term t_0 is equal to a constant c and t_i is $f(t_{i-1}, t_{i-1})$ for a function symbol f of arity two, which is the most common arity used in practice. Note that due to sharing, this term consists of 400 001 term nodes. In (b) each thread creates the term $t_{400\,000/\#threads}$ instead. With each term starting with a unique constant per thread, creating a total of $400\,000 + \#threads$ term nodes.

In Figs. 4 (c) and (d) we measure the time it takes to create $1000/\#threads$ instances of the terms used in respectively (a) and (b). This measures the time to create terms that are already present in the term library, and this essentially boils down to a hash table lookup. In diagram (d) the Java results are left out as Java consistently requires more than 100 s. In the lower diagram, *i.e.*(e), we measure the time to perform $1000/\#threads$ breadth-first traversals on a term t_{20}.

The traversals do not employ the shared structure, hence $2^{21} - 1$ terms are visited per traversal. We observe that for this benchmark there is no difference in timings between traversing the term t_{20} and traversing a unique term per thread.

We conclude that our term library completely outperforms the Java implementation. For creating terms, the `std::shared_mutex` is slower. For traversing terms no locking is required, and therefore, no difference is observed. The dotted line is hidden under the line with the boxes[2]. Except for creating new terms, the term library clearly benefits from the extra processors, outperforming the

[2] All benchmark results are listed in Appendix C.



(a) Creating new terms (shared).

(b) Creating new terms (distinct).

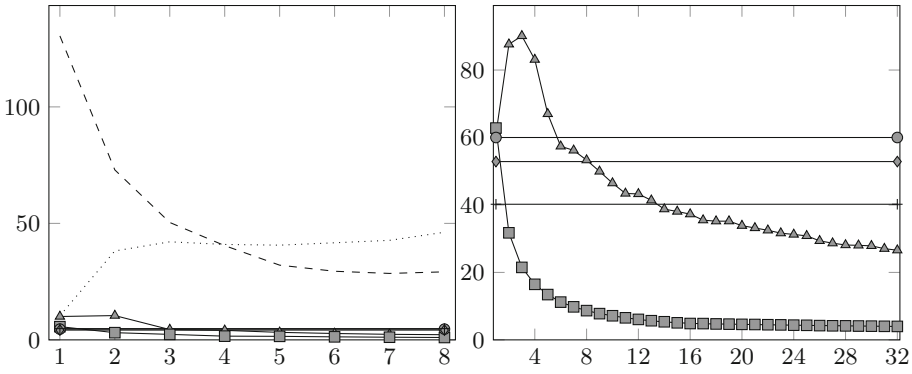
(c) Creating existing terms (shared).

(d) Creating existing terms (distinct).

(e) Traversing terms (shared).

(f) Legend for all plots in Figures 4 and 5.

Fig. 4. Execution time (in seconds) plotted against number of threads.



(a) Creating existing terms (shared, Intel). (b) State space exploration.

Fig. 5. Additional experiments comparing execution times versus threads.

sequential libraries with two processors using protection sets. When creating new terms, scaling goes reasonably well when beyond 12 processors. We observe in Fig. 4 (c) that the reference counting implementation for a few threads is unexpectedly inefficient. In order to understand this, we retried the experiments on an Intel i7-7700HQ processor, reported in Fig. 5 (a). Here, none of the anomalies occur, and notably, Java even outperforms the `std::shared_mutex` implementation with more than four threads. This is in line with our many other experiments that compiler and processor have a large influence on such benchmarks.

The dedicated benchmarks are promising, but in order to get insight in the behaviour of the Term Library in practical situations, we incorporated the Term Library in the mCRL2 toolset and used it to generate the state space of the 1394 firewire protocol [22]. Essentially each thread picks an unexplored state from a common state buffer, and using term rewriting, generates all states reachable from this state, putting them back in the buffer. With protection sets, two threads are already sufficient to outperform all sequential implementations, and scaling is very good, where with 16 threads, the state space is generated more than 12 times faster. Reference counting is clearly a less viable option, which is most likely due to the fact that often the same terms, such as *true* and *false*, are accessed when calculating next states, leading to atomically changing the same reference count often. Note that in this prototype, nothing has been done yet to optimise thread access to the common state buffer, being simply protected by a mutex.

A The mCRL2 Language and Modal Formulas

The models are written in mCRL2. This is a modelling language based on CCS (Calculus of Communicating Processes) [24] and ACP (Algebra of Communicating Processes) [1]. It is based on atomic actions. Every occurrence of an atomic

action causes a state change. Typically, calling a function, or returning from a function, setting or reading a global variable are modelled by atomic actions. Each action consists of a label and a possible set of data parameters, *e.g.*, the $lock(p)$ action has p as a data parameter. The tau or hidden action τ has a special status, as it is an action of which the occurrence cannot be observed directly.

Actions can be sequentially composed using the dot ('$\cdot$') operator. Alternative composition, where nondeterministically one of the options can be chosen, is denoted using a plus ('+'). The $\sum_{e:S}$ operator denotes the application of the ('+') operator over all elements of some set S. The if-then-else is written as $c \rightarrow p \diamond q$ where p is executed if c is true, otherwise the process q takes place.

Parallel composition is denoted by $\|$ and allows the actions of two processes to both interleave and occur simultaneously. Using the **comm** and **allow** operators, we can enforce that only specific actions can and must occur simultaneously. For example, $\mathbf{comm}(\{a,b\} \rightarrow c\}, \mathbf{allow}(\{c,d,e\}, (d \cdot a)||(b \cdot e)))$ enforces that the actions d and e can not occur simultaneously, while a and b must occur simultaneously with any other action, signified with c. As a result, $d \cdot c \cdot e$ is the only possible sequence of actions that can occur.

Recursive behaviour is denoted using equations, typically of the form $X = p$, *e.g.*, $X = a \cdot X$ is the process that can perform an infinite number of a's. Similar to actions, the process variables X can contain data parameters. A counter can thus be described as $C(n{:}\mathbb{N}) = up \cdot X(n{+}1)$. An important type of data parameter that we use is a function. For example, the process variable $Y(m : \mathbb{N} \rightarrow \mathbb{B})$ uses a mapping m from natural numbers to booleans. The function update $m[n \mapsto b]$ specifies that $m[n \mapsto b](k)$ equals b if $k = n$, and otherwise, it equals $m(k)$.

The safety and liveness properties that we verify, are written in the modal mu-calculus. These consist of conjunctions ($\wedge$), disjunctions ($\vee$), implications ($\rightarrow$), negations ($\neg$), quantification ($\exists, \forall$) and *true* and *false*, each with their usual meaning. Besides this there is a modality $\langle a \rangle \phi$ that is valid if we can take an action a after which ϕ holds. Similarly, the modality $[a]\phi$ holds iff after every possible a action ϕ holds. The action a inside these modalities can also consist of possibly multiple actions. This can be done through sequential composition ($\cdot$), choice ($\cup$), intersection ($\cap$) and complement ($\overline{a}$). For example, the formula $\langle \overline{a} \cup \overline{b} \rangle true$ only holds if we can do some action that is neither a or b. The expression *true* in a modality represents the set of all actions. Using Kleene's star on a set of actions, all sequences over the action in this set are expressed. An often occurring pattern is $[true^*]\phi$ expressing that ϕ must hold in all states reachable via a sequence of actions.

We can also write recursive formulas using the minimal fixed point operator $\mu X.\phi$ and the maximal fixed point operator $\nu X.\phi$. For example, the formula $\nu X.\langle a \rangle X$ expresses that we must be able to perform action a after which the same formula still holds. Thus this formula only holds if we can perform an infinite amount of a actions. The difference between a minimal and a maximal fixed point is that iteration through the fixed point variable must be bounded in a minimal fixed point.

A fixed point construction used in several properties is

$$\nu X.\mu Y.([\overline{succes \cup interrupt}]Y \wedge [interrupt]X \wedge \langle true^*.succes\rangle true)$$

This says that an action *succes* will always occur within a finite amount of steps, unless an action *interrupt* continuously occurs. But even in that case *succes* must remain possible. This construction is useful for properties in which we state that something must eventually happen given fair scheduling.

The fixed point operators also allow us to pass on parameters in the same way we can do for process variables. This allows us, for example, to keep track of the number of times that a given action has occurred. We discuss one such fixed point operator in Appendix B.

B mCRL2 Specifications for the Busy-Forbidden Protocol

In this section we give the formal mCRL2 specifications of the implementation and the external behaviour, *i.e.*, the specification, of the busy-forbidden protocol that are used to perform the model and equivalence checking. The process specification given in Table 3 exactly matches the external behaviour shown in Fig. 2. We define P to be the (finite) set of threads and we define S to be a data set representing the set of states:

$$S = \{Free, EnterS, LOE1, Shared, LeaveS, \\ EnterE, LOE2, LOS, Exclusive, LeaveE1, LeaveE2\}$$

The mapping s maps each thread to their current state. Initially, $s(p) = Free$ for all $p \in P$. The conditions for performing transitions are the same as the conditions in the diagram of the external behaviour.

Observe that we use a typewriter font (for example `enter_shared_call`) to indicate visible actions and an italics font (for example $store_p$) to indicate internal actions that will be hidden for divergence-preserving branching bisimulation reductions.

The mCRL2 specification of the implementation is separated per function. Entering the shared section is specified in Table 4 and leaving it in Table 5. Entering the exclusive section is specified in Table 6 and leaving it in Table 7.

Note that we use actions to model the assignments to variables. For example $store_p(Busy(p), true, p)$ corresponds to the assignment of *true* to $p.busy$ in the implementation pseudocode. The process algebra has no global variables and we use an additional process and actions to read from and write to these variables. For the atomic flags we introduce a struct F that is defined below to declare a busy and a forbidden flag per thread.

$$\texttt{sort}\ F = \texttt{struct}\ Busy(P) \mid Forbidden(P)$$

Table 3. Specification of the busy-forbidden protocol corresponding to Fig. 2.

$$
\begin{aligned}
&BF(s : P \to S) = \textstyle\sum_{p:P} .(\\
&\quad (s(p) \approx \mathit{Free}) \\
&\quad \to \texttt{enter\_shared\_call}(p).BF(s[p \mapsto \mathit{EnterS}]) \\
&\quad + (s(p) \approx \mathit{EnterS}) \\
&\quad \to ((\neg\exists_{p':P}.\ s(p') \in \{\mathit{LOS}, \mathit{Exclusive}\}) \to \tau.BF(s[p \mapsto \mathit{LOE1}]) \\
&\quad \diamond\ \mathit{improbable}.BF(s)) \\
&\quad + (s(p) \approx \mathit{LOE1}) \\
&\quad \to \texttt{enter\_shared\_return}(p).BF(s[p \mapsto \mathit{Shared}]) \\
&\quad + (s(p) \approx \mathit{Shared}) \\
&\quad \to \texttt{leave\_shared\_call}(p).BF(s[p \mapsto \mathit{LeaveS}]) \\
&\quad + (s(p) \approx \mathit{LeaveS}) \\
&\quad \to \texttt{leave\_shared\_return}(p).BF(s[p \mapsto \mathit{Free}]) \\
&\quad + (s(p) \approx \mathit{Free}) \\
&\quad \to \texttt{enter\_exclusive\_call}(p).BF(s[p \mapsto \mathit{EnterE}]) \\
&\quad + (s(p) \approx \mathit{EnterE} \wedge \neg\exists_{p':P}.\ s(p') \in \{\mathit{LOE2}, \mathit{LOS}, \mathit{Exclusive}\}) \\
&\quad \to \tau.BF(s[p \mapsto \mathit{LOE2}]) \\
&\quad + (s(p) \approx \mathit{LOE2}) \\
&\quad \to \mathit{improbable}.BF(s) \\
&\quad + (s(p) \approx \mathit{LOE2} \wedge \neg\exists_{p':P}.\ s(p') \in \{\mathit{LOE1}, \mathit{Shared}\}) \\
&\quad \to \tau.BF(s[p \mapsto \mathit{LOS}]) \\
&\quad + (s(p) \approx \mathit{LOS}) \\
&\quad \to \texttt{enter\_exclusive\_return}(p).BF(s[p \mapsto \mathit{Exclusive}]) \\
&\quad + (s(p) \approx \mathit{Exclusive}) \\
&\quad \to \texttt{leave\_exclusive\_call}(p).BF(s[p \mapsto \mathit{LeaveE1}]) \\
&\quad + (s(p) \approx \mathit{LeaveE1}) \\
&\quad \to \mathit{improbable}.BF(s) \\
&\quad + (s(p) \approx \mathit{LeaveE1}) \\
&\quad \to \tau.BF(s[p \mapsto \mathit{LeaveE2}]) \\
&\quad + (s(p) \approx \mathit{LeaveE2}) \\
&\quad \to \texttt{leave\_exclusive\_return}(p).BF(s[p \mapsto \mathit{Free}])\ )
\end{aligned}
$$

Table 8 shows the behaviour of the *Busy* and *Forbidden* flags for every thread and the *mutex* variable. We model the 'while' construction in the pseudocode by recursion and have added the *improbable* action to ensure equivalence modulo divergence-preserving branching bisimulation. When entering the exclusive section, we use a set *forbidden* (and for leaving *allowed*) to keep track of the threads whose forbidden flag have already been set to *true* (*false* when leaving).

Table 9 shows the specification for the behaviour of a thread. Each thread repeatedly chooses (non-deterministically) to enter and leave either the shared or

exclusive section. Finally, Table 10 contains the complete mCRL2 specification of the various processes in a parallel composition and the necessary communication to deal with the atomic flags and mutex.

As the next step, we transformed the six requirements discussed in Sect. 3.3 into modal logic formulas, and verified them on the specification. Note that these properties are preserved by divergence-preserving branching bisimulation, so verifying the implementation is not necessary. We discuss property 2 in detail as an illustration of what such formulas look like. The informal description of the property reads:

2. There should never be a thread present in the exclusive section while one or more threads are present in the shared section.

The corresponding modal formula is shown in Table 12. We use a maximal fixpoint with two data parameters, namely n_{shared} and $n_{exclusive}$, both initially 0. The argument n_{shared} indicates the number of threads present in the shared section, and $n_{exclusive}$ the number in the exclusive section. At lines 2 through 5, we keep track of the amount of threads present in each section, updating the variables after each respective action. At lines 6 through 11, we state that our variables stay the same, after any action that is not one of the four aforementioned actions. Finally, at line 12, we state that threads are only allowed to be either present in exclusive or are present in shared.

The formula for property 1 is shown in Table 11 and states that when a thread enters the exclusive section, no other thread may enter that section till it leaves the section. The formulas for properties 3 and 4 are presented in Tables 13 and 15 and use data parameters to count the number of threads in the exclusive section or in any section respectively. Note that these are two subformulas with identical structure for shared and exclusive sections respectively. Finally, properties 4 and 6 presented in Tables 14 and 16 use boolean parameters to keep track of whether any thread is in the shared or exclusive sections, respectively. This is more efficient than keeping track of the exact amount of threads.

The properties were verified for up to 7 threads. The specification model has about three million states and the implementation model about 11 billion states.

Table 4. mCRL2 specification for the implementation of `enter_shared`.

$$
\begin{array}{l}
\mathit{EnterShared}(p : P) = \\
\quad \texttt{enter\_shared\_call}(p)\ . \\
\quad \mathit{TryBothFlags}(p)\ . \\
\quad \texttt{enter\_shared\_return}(p) \\
\\
\mathit{TryBothFlags}(p : P) = \\
\quad \mathit{store}_p(\mathit{Busy}(p), \mathit{true}, p).(\\
\quad\quad \mathit{load}_p(\mathit{Forbidden}(p),\ \mathit{true},\ p)\ . \\
\quad\quad \mathit{store}_p(\mathit{Busy}(p), \mathit{false}, p).\mathit{improbable}.\mathit{TryBothFlags}(p) \\
\quad + \ \mathit{load}_p(\mathit{Forbidden}(p), \mathit{false}, p)\ )
\end{array}
$$

Table 5. mCRL2 specification for the implementation of `leave_shared`.

$$
\begin{array}{l}
\mathit{LeaveShared}(p : P) = \\
\quad \texttt{leave\_shared\_call}(p)\ . \\
\quad \mathit{store}_p(\mathit{Busy}(p), \mathit{false}, p)\ . \\
\quad \texttt{leave\_shared\_return}(p)
\end{array}
$$

Table 6. mCRL2 specification for the `enter_exclusive` function in Table 2.

$$
\begin{array}{l}
\mathit{EnterExclusive}(p : P) = \\
\quad \texttt{enter\_exclusive\_call}(p) \\
\quad \mathit{lock}_p(p)\ . \\
\quad \mathit{SetAllForbiddenFlags}(p,\ \emptyset)\ . \\
\quad \texttt{enter\_exclusive\_return}(p) \\
\\
\mathit{SetAllForbiddenFlags}(p : P,\ \mathit{forbidden} : \mathit{Set}(P)) = \\
\quad (\forall_{p':P}.p \in \mathit{forbidden}) \\
\quad \rightarrow \mathit{internal} \\
\quad \diamond \ \textstyle\sum_{p':P} .\mathit{store}_p(\mathit{Forbidden}(p'), \mathit{true}, p).(\\
\quad\quad \mathit{load}_p(\mathit{Busy}(p'), \mathit{false}, p)\ . \\
\quad\quad \mathit{SetAllForbiddenFlags}(p,\ \mathit{forbidden} \cup \{p'\}) \\
\quad\quad + \ \mathit{load}_p(\mathit{Busy}(p'), \mathit{true}, p)\ . \\
\quad\quad \mathit{store}_p(\mathit{Forbidden}(p'), \mathit{false}, p).\mathit{improbable}\ . \\
\quad\quad \mathit{SetAllForbiddenFlags}(p,\ \mathit{forbidden} \setminus \{p'\}) \\
\quad\quad + \ \mathit{store}_p(\mathit{Forbidden}(p'), \mathit{false}, p).\mathit{improbable}\ . \\
\quad\quad \mathit{SetAllForbiddenFlags}(p,\ \mathit{forbidden} \setminus \{p'\})\ )
\end{array}
$$

Table 7. mCRL2 specification for the `leave_exclusive` function in Table 2.

$$
\begin{array}{l}
\mathit{LeaveExclusive}(p : P) = \\
\quad \texttt{leave\_exclusive\_call}(p)\ . \\
\quad \mathit{AllowAllThreads}(p, \emptyset)\ . \\
\quad \mathit{unlock}_p(p)\ . \\
\quad \texttt{leave\_exclusive\_return}(p) \\
\\
\mathit{AllowAllThreads}(p : P,\ \mathit{allowed} : \mathit{Set}(P)) = \\
\quad (\forall_{q:P}.q \in \mathit{allowed}) \\
\quad \rightarrow \mathit{internal} \\
\quad \diamond\ \textstyle\sum_{p':P} .(\\
\quad\quad \mathit{store}_p(\mathit{Forbidden}(p'), \mathit{false}, p)\ . \\
\quad\quad \mathit{AllowAllThreads}(p,\ \mathit{allowed} \cup \{p'\}) \\
\quad + \ \mathit{store}_p(\mathit{Forbidden}(p'), \mathit{true}, p).\mathit{improbable} \\
\quad\quad \mathit{AllowAllThreads}(p,\ \mathit{allowed} \setminus \{p'\}\ ))
\end{array}
$$

Table 8. mCRL2 specifications for the atomic flags and the mutex.

$$
\begin{array}{l}
\mathit{Flags}(\mathit{flags} : F \rightarrow \mathit{Bool}) = \\
\textstyle\sum_{f:F,p:P} .(\\
\quad \textstyle\sum_{b:\mathit{Bool}} .\mathit{store}_f(f, b, p).\mathit{Flag}(\mathit{flags}[f \mapsto b]) \\
\quad + \ \mathit{load}_f(f, \mathit{flags}(f), p).\mathit{Flag}(\mathit{flags}) \\
\quad) \\
\\
\mathit{Mutex}(\mathit{locked} : \mathit{Bool}) = \\
\textstyle\sum_{p:P} .(\\
\quad \mathit{locked} \\
\quad \rightarrow \mathit{lock}_m(p).\mathit{Mutex}(\mathit{true}) \\
\quad \diamond\ \mathit{unlock}_m(p).\mathit{Mutex}(\mathit{false})\ )
\end{array}
$$

Table 9. mCRl2 specification for a thread p interacting with the protocol.

$$
\begin{array}{l}
\mathit{Thread}(p : P) = \\
\quad \mathit{EnterShared}(p)\ . \\
\quad \mathit{LeaveShared}(p)\ . \\
\quad \mathit{Thread}(p) \\
+ \ \mathit{EnterExclusive}(p)\ . \\
\quad \mathit{LeaveExclusive}(p)\ . \\
\quad \mathit{Thread}(p)
\end{array}
$$

Table 10. mCRL2 specification for the busy-forbidden protocol.

```
allow({
        store, load,
        lock, unlock,
        internal, improbable,
        enter_shared_call, enter_shared_return,
        leave_shared_call, leave_shared_return,
        enter_exclusive_call, enter_exclusive_return,
        leave_exclusive_call, leave_exclusive_return
        }, comm({
            store_f|store_p → store,
            load_f|load_p → load,
            lock_m|lock_p → lock,
            unlock_m|unlock_p → unlock
            },
            Thread(p_1) ||
                ⋮
            Thread(p_|P|) ||
            Flags(λf:F.false) ||
            Mutex(false) ) )
```

Table 11. Modal formula for property 1: "There should never be more than one thread present in the exclusive section".

$$
\begin{array}{l}
[true^*] \\
[\exists_{p \in P} : \texttt{enter\_exclusive\_return}(p)] \\
[\overline{\exists_{p \in P} : \texttt{leave\_exclusive\_call}(p)}^{\,*}] \\
[\exists_{p \in P} : \texttt{enter\_exclusive\_return}(p)] \\
false
\end{array}
$$

Table 12. The modal formula for property 2: "There should never be a thread present in the exclusive section while one or more threads are present in the shared section".

$$
\begin{array}{ll}
\nu X(n_{shared} : Nat = 0,\ n_{exclusive} : Nat = 0). & \\
\quad (\forall_{p:P}.[\texttt{enter\_shared\_return}(p)]X(n_{shared}+1, n_{exclusive})\ ) & \\
\wedge\ (\forall_{p:P}.[\texttt{enter\_exclusive\_return}(p)]X(n_{shared}, n_{exclusive}+1)\ ) & \\
\wedge\ (\forall_{p:P}.[\texttt{leave\_shared\_call}(p)]X(n_{shared}-1, n_{exclusive})\ ) & \\
\wedge\ (\forall_{p:P}.[\texttt{leave\_exclusive\_call}(p)]X(n_{shared}, n_{exclusive}-1)\ ) & \\
\wedge\ [\ \overline{(\exists_{p:P}.\texttt{enter\_shared\_return}(p)\ )} & \\
\quad \cap\ \overline{(\exists_{p:P}.\texttt{enter\_exclusive\_return}(p)\ )} & \\
\quad \cap\ \overline{(\exists_{p:P}.\texttt{leave\_shared\_call}(p)\ )} & \\
\quad \cap\ \overline{(\exists_{p:P}.\texttt{leave\_exclusive\_call}(p)\ )} & \\
\quad]X(n_{shared}, n_{exclusive}) & \\
\wedge\ \neg(n_{exclusive} > 0 \wedge n_{shared} > 0) &
\end{array}
$$

Table 13. Modal formula for property 3: "When a thread requests to enter the shared section, it will be granted access within a bounded number of steps, unless there is another thread in the exclusive section".

$$
\begin{array}{l}
\nu X(n_{exclusive} : Nat = 0). \\
\quad [\exists_{p:P}.\ \texttt{enter\_exclusive\_call}(p)]X(n_{exclusive} + 1) \\
\wedge\ [\exists_{p:P}.\ \texttt{leave\_exclusive\_return}(p)]X(n_{exclusive} - 1) \\
\wedge\ [\ \overline{(\exists_{p:P}.\ \texttt{enter\_exclusive\_call}(p))} \\
\quad \cap\ \overline{(\exists_{p:P}.\ \texttt{leave\_exclusive\_return}(p))} \\
\quad]\ X(n_{exclusive}) \\
\wedge\ \forall_{p:P}.[\texttt{enter\_shared\_call}(p)] \\
\quad \nu Y(n'_{exclusive} : Nat = n_{exclusive}).\mu Z(n''_{exclusive} : Nat = n'_{exclusive}).\ (\\
\qquad [\ \overline{\texttt{enter\_shared\_return}(p)} \\
\qquad \cap\ \overline{(\exists_{p':P}.\ \texttt{enter\_shared\_call}(p'))} \\
\qquad \cap\ \overline{(\exists_{p':P}.\ \texttt{enter\_exclusive\_call}(p'))} \\
\qquad \cap\ \overline{(\exists_{p':P}.\ \texttt{leave\_exclusive\_return}(p'))} \\
\qquad \cap\ \overline{improbable} \\
\qquad]\ (\ ((n''_{exclusive} \approx 0)\ \Longrightarrow\ Z(n''_{exclusive})) \\
\qquad \wedge\ ((n''_{exclusive} > 0)\ \Longrightarrow\ Y(n''_{exclusive}))\ ) \\
\quad \wedge\ [\exists_{p':P}.\ \texttt{enter\_shared\_call}(p')]Y(n''_{exclusive}) \\
\quad \wedge\ [\exists_{p':P}.\ \texttt{enter\_exclusive\_call}(p')]Y(n''_{exclusive} + 1) \\
\quad \wedge\ [\exists_{p':P}.\ \texttt{leave\_exclusive\_return}(p')]Y(n''_{exclusive} - 1) \\
\quad \wedge\ [improbable]Y(n''_{exclusive}) \\
\quad \wedge\ \langle true^*.\texttt{enter\_shared\_return}(p)\rangle true\ )
\end{array}
$$

Table 14. Modal formula for property 5: "When a thread requests to leave the exclusive/shared section, it will leave it within a bounded number of steps".

$$
\begin{array}{l}
[true^*]\ \forall_{p:P}.(\\
\quad [\texttt{leave\_shared\_call}(p)]\nu X.\mu Y.(\\
\qquad [\ \overline{\texttt{leave\_shared\_return}(p)} \\
\qquad \cap\ \overline{(\exists_{p':P}.\ \texttt{enter\_exclusive\_call}(p'))} \\
\qquad \cap\ \overline{(\exists_{p':P}.\ \texttt{enter\_shared\_call}(p'))} \\
\qquad \cap\ \overline{improbable}\ ]\ Y \\
\quad \wedge\ [\ (\exists_{p':P}.\ \texttt{enter\_exclusive\_call}(p')) \\
\qquad \cup\ (\exists_{p':P}.\ \texttt{enter\_shared\_call}(p')) \\
\qquad \cup\ (improbable)\ ]\ X \\
\quad \wedge\ \langle true^*.\texttt{leave\_shared\_return}(p)\rangle true\ ) \\
\wedge\ [\texttt{leave\_exclusive\_call}(p)]\nu X.\mu Y.(\\
\qquad [\ \overline{\texttt{leave\_exclusive\_return}(p)} \\
\qquad \cap\ \overline{(\exists_{p':P}.\ \texttt{enter\_exclusive\_call}(p'))} \\
\qquad \cap\ \overline{(\exists_{p':P}.\ \texttt{enter\_shared\_call}(p'))} \\
\qquad \cap\ \overline{improbable}\ ]\ Y \\
\quad \wedge\ [\ (\exists_{p':P}.\ \texttt{enter\_exclusive\_call}(p')) \\
\qquad \cup\ (\exists_{p':P}.\ \texttt{enter\_shared\_call}(p')) \\
\qquad \cup\ (improbable)\ ]\ X \\
\quad \wedge\ \langle true^*.\texttt{leave\_exclusive\_return}(p)\rangle true\ )\ )
\end{array}
$$

Table 15. Modal formula for property 4: "When a thread requests to enter the exclusive section, it will be granted access within a bounded number of steps, unless there is another thread in the shared or in the exclusive section".

$$
\begin{array}{l}
\nu X(n_{blocking} : Nat = 0). \\
\quad [\exists_{p:P}.\ \texttt{enter\_exclusive\_call}(p)]X(n_{blocking} + 1) \\
\wedge\ [\exists_{p:P}.\ \texttt{enter\_shared\_call}(p)]X(n_{blocking} + 1) \\
\wedge\ [\exists_{p:P}.\ \texttt{leave\_shared\_return}(p)]X(n_{blocking} - 1) \\
\wedge\ [\exists_{p:P}.\ \texttt{leave\_exclusive\_return}(p)]X(n_{blocking} - 1) \\
\wedge\ [\ \overline{(\exists_{p:P}.\ \texttt{enter\_exclusive\_call}(p))} \\
\quad \cap\ \overline{(\exists_{p:P}.\ \texttt{leave\_exclusive\_return}(p))} \\
\quad \cap\ \overline{(\exists_{p:P}.\ \texttt{enter\_shared\_call}(p))} \\
\quad \cap\ \overline{(\exists_{p:P}.\ \texttt{leave\_shared\_return}(p))} \\
\quad]\ X(n_{exclusive}) \\
\wedge\ \forall_{p:P}.[\texttt{enter\_exclusive\_call}(p)] \\
\quad \nu Y(n'_{blocking} : Nat = n_{blocking}).\mu Z(n''_{blocking} : Nat = n'_{blocking}).\ (\\
\quad\quad [\ \overline{\texttt{enter\_exclusive\_return}(p)} \\
\quad\quad \cap\ \overline{(\exists_{p':P}.\ \texttt{enter\_shared\_call}(p))} \\
\quad\quad \cap\ \overline{(\exists_{p':P}.\ \texttt{leave\_shared\_return}(p))} \\
\quad\quad \cap\ \overline{(\exists_{p':P}.\ \texttt{enter\_exclusive\_call}(p'))} \\
\quad\quad \cap\ \overline{(\exists_{p':P}.\ \texttt{leave\_exclusive\_return}(p'))} \\
\quad\quad \cap\ \overline{improbable} \\
\quad\quad]\ (\ ((n''_{blocking} \approx 0)\ \Longrightarrow\ Z(n''_{blocking})) \\
\quad\quad \wedge\ ((n''_{blocking} > 0)\ \Longrightarrow\ Y(n''_{blocking}))\ ) \\
\quad \wedge\ [\exists_{p':P}.\ \texttt{enter\_shared\_call}(p')]Y(n''_{blocking} + 1) \\
\quad \wedge\ [\exists_{p':P}.\ \texttt{leave\_shared\_return}(p')]Y(n''_{blocking} - 1) \\
\quad \wedge\ [\exists_{p':P}.\ \texttt{enter\_exclusive\_call}(p')]Y(n''_{blocking} + 1) \\
\quad \wedge\ [\exists_{p':P}.\ (p' \not\approx p)\ \wedge\ \texttt{enter\_exclusive\_return}(p')]Y(n''_{blocking} - 1) \\
\quad \wedge\ [improbable]Y(n''_{exclusive}) \\
\quad \wedge\ \langle true^*.\texttt{enter\_exclusive\_return}(p)\rangle true\ )
\end{array}
$$

Table 16. Modal formula for property 6: "A thread not in the exclusive or shared section can instantly start to enter the exclusive or shared section".

$$
\begin{array}{l}
\forall_{p:P}.\nu X(b_{shared} : Bool = false,\ b_{exclusive} : Bool = false). \\
\quad [\texttt{enter\_shared\_call}(p)]X(true, b_{exclusive}) \\
\wedge\ [\texttt{leave\_shared\_return}(p)]X(false, b_{exclusive}) \\
\wedge\ [\texttt{enter\_exclusive\_call}(p)]X(b_{shared}, true) \\
\wedge\ [\texttt{leave\_exclusive\_return}(p)]X(b_{shared}, false) \\
\wedge\ [\ \overline{\texttt{enter\_shared\_call}(p)} \\
\quad \cap\ \overline{\texttt{leave\_shared\_return}(p)} \\
\quad \cap\ \overline{\texttt{enter\_exclusive\_call}(p)} \\
\quad \cap\ \overline{\texttt{leave\_exclusive\_return}(p)} \\
\quad]\ X(n_{shared}, n_{exclusive}) \\
\wedge\ ((\neg n_{shared} \wedge \neg n_{exclusive})\ \Longrightarrow\ (\\
\quad\quad \langle \texttt{enter\_exclusive\_call(p)}\rangle true \\
\quad \wedge\ \langle \texttt{enter\_shared\_call(p)}\rangle true)\ )
\end{array}
$$

C mCRL2 Specifications for the Term Library

In this section we give the formal mCRL2 specifications of the implementation of the term library that is used to perform model checking. Creating a term is specified in Table 17 and destroying a term in Table 18. In this model, the set P corresponds to the set containing all threads, T to the set containing all terms and A to the set containing all memory addresses. The set $A_\perp = A \cup \{\perp\}$ with $\perp \notin A$ contains the extra element $\perp$ meaning no address or a NULL pointer. To ensure finiteness and reduce the complexity of the model, the set T only contains a finite amount of constants, *i.e.*, terms of arity zero.

Table 17. mCRL2 specification for the `create` function shown in Table 1.

$$
\begin{aligned}
&\mathit{Create}(p : P,\ t : T,\ lm : T \to A_\perp) = \\
&\qquad \texttt{create\_call}(p,\ t)\ . \\
&\qquad \mathit{EnterShared}(p)\ . \\
&\qquad \mathit{Create}_2(p,\ t,\ lm) \\
&\\
&\mathit{Create}_2(p : P,\ t : T,\ lm : T \to A_\perp) = \\
&\textstyle\sum_{a:A_\perp} .(\\
&\qquad \mathit{contains}_p(t, a, p)\ . \\
&\qquad (a \approx \perp) \\
&\qquad \to \textstyle\sum_{a':A} \cdot\ (\\
&\qquad\qquad \mathit{construct\_term}_p(t\ , a'\ , p).(\\
&\qquad\qquad\quad \mathit{insert}_p(t\ , a'\ , \mathit{true}\ , p)\ . \\
&\qquad\qquad\quad \mathit{Create}_3(p,\ t,\ lm,\ a') \\
&\qquad\qquad + \ \mathit{insert}_p(t,\ a',\ \mathit{false},\ p)\ . \\
&\qquad\qquad\quad \mathit{destruct\_term}_p(t,\ a',\ p)\ . \\
&\qquad\qquad\quad \mathit{Create}_2(p,\ t,\ lm)\ )\ ) \\
&\qquad \diamond\ \ \mathit{Create}_3(p,\ t,\ lm,\ a)\ ) \\
&\\
&\mathit{Create}_3(p : P, t : T,\ lm : T \to A_\perp,\ a : A) = \\
&\qquad \mathit{protect}_p(t,\ a,\ p)\ . \\
&\qquad \mathit{LeaveShared}(p)\ . \\
&\qquad \texttt{create\_return}(p,\ t,\ a)\ . \\
&\qquad \mathit{Thread}(p,\ lm[t \mapsto a])
\end{aligned}
$$

First of all, we introduce processes *EnterShared*, *LeaveShared*, *EnterExclusive* and *LeaveExclusive* to interact with the busy-forbidden specification *BF* specified in Table 3. To distinguish between the term library and the protocol all actions such as `enter_shared_call` are split into action $\texttt{enter\_shared\_call}_{bf}$

for the protocol and $\texttt{enter\_shared\_call}_p$ for the term library. Finally, we have the process *MainMemory* to model the main memory by keeping track of *used* memory addresses, the process *HashTable* to model a hash table as an associative array, and process *ReferenceCounter* to track a reference counter for every address (or term). Destroying a term is specified in Table 18, which uses the same other processes as the creation function. Again, there are two separate processes to model the behaviour of the while loop. In Table 19 the behaviour of the main memory, the hash table and the reference counter are specified.

The specification in Table 20 models the behaviour of each thread. Each thread repeatedly tries to either creates a term it does not yet know, or it destroys a known term. Finally, Table 21 shows the complete specification including the communication between various processes used to model the thread-safe term library.

Table 18. mCRL2 specification for the destroy function shown in Table 1.

$$
\begin{array}{l}
\mathit{Destroy}(p : P,\ t : T,\ \mathit{lm} : T \to A_\bot) = \\
\quad \texttt{destroy\_call}(p,\ t)\ . \\
\quad \mathit{unprotect}_p(t,\ \mathit{lm}(t),\ p).(\\
\quad\quad \mathit{skip} \\
\quad + \ \mathit{skip}.\mathit{GC}(p)\ )\ . \\
\quad \texttt{destroy\_return}(p)\ . \\
\quad \mathit{Thread}(p,\ \mathit{lm}[t \mapsto \bot]) \\
\\
\mathit{GC}(p : P) = \\
\quad \mathit{EnterExclusive}(p)\ . \\
\quad \mathit{GC}_2(p,\ \emptyset) \\
\\
\mathit{GC}_2(p : P,\ \mathit{checked} : \mathit{FSet}(T)) = \\
\quad (\forall_{t:T}.t \in \mathit{checked}) \\
\quad \to \mathit{LeaveExclusive}(p) \\
\quad \diamond \ \textstyle\sum_{t:T}.(t \notin \mathit{checked}) \to (\\
\quad\quad \mathit{contains}_p(t,\ \bot,\ p)\ . \\
\quad\quad \mathit{GC}_2(p,\ \mathit{checked} \cup \{t\} \\
\quad\quad + \ \textstyle\sum_{a:A}.\mathit{contains}_p(t,\ a,\ p).(\\
\quad\quad\quad \mathit{protected}_p(a,\ \mathit{true},\ p)\ . \\
\quad\quad\quad \mathit{GC}_2(p,\ \mathit{checked} \cup \{t\}) \\
\quad\quad\quad + \ \mathit{protected}_p(a,\ \mathit{false},\ p)\ . \\
\quad\quad\quad \mathit{destruct\_term}_p(t,\ a,\ p)\ . \\
\quad\quad\quad \mathit{delete}_p(t,\ p)\ . \\
\quad\quad\quad \mathit{GC}_2(p,\ \mathit{checked} \cup \{t\})\ )\ )
\end{array}
$$

To verify the model of the thread-safe term library we again specify a number of modal formulas for the properties described in Sect. 2.2. The modal formula for property 1 specified in Table 22 uses a mapping a from addresses to terms and the finite set *owners* containing all threads that own/protect term t as data parameters. If at any point in time a `create(t)` returns a different address than the current address, then the term must not be owned by any thread. The modal formula in Table 23 for property 2 uses the same constructs to check whether terms on the same address are also equivalent.

The formula for property 3 shown in Table 24 uses a boolean parameter *busy* to keep track of whether the thread p is creating (or destroying) a term. Furthermore, the parameter *known* is a finite set containing all terms that thread p knows. If at any point in time *busy* is false, then the process must be able to start destroying any term in *known* and start creating any term not currently in *known*. Finally, for property 4 the formula shown in Table 25 uses again the construction which (under fairness) indicates that term creation and destruction will finish within a finite number of steps. Note that the subformulas for creation and destruction have an identical structure.

Table 19. mCRL2 specifications of the main memory, hash table and reference counters used in the term library specification.

$$
\begin{array}{l}
\mathit{MainMemory}(\mathit{used} : \mathit{FSet}(A)) = \\
\sum_{p:P,t:T,a:A} .((a \notin \mathit{used}) \\
\qquad \rightarrow \mathit{construct\_term}_{mm}(t,a,p).\mathit{MainMemory}(\mathit{used} \cup \{a\}) \\
\qquad \diamond \ \ \mathit{destruct\_term}_{mm}(t,a,p).\mathit{MainMemory}(\mathit{used} \setminus \{a\})\ ) \\
\\
\mathit{HashTable}(m : T \rightarrow A_{\bot}) = \\
\sum_{t:T,p:P} .(\\
\qquad \mathit{contains}_{ht}(t, m(e), p).\mathit{HashTable}(m) \\
\quad + \ \sum_{a:A} .(m(e) \approx \bot) \\
\qquad \rightarrow \mathit{insert}_{ht}(t,a,\mathit{true},p).\mathit{HashTable}(m[e \mapsto a) \\
\qquad \diamond \ \ \mathit{insert}_{ht}(t,a,\mathit{false},p).\mathit{HashTable}(m) \\
\quad + \ \mathit{delete}_{ht}(t,p).\mathit{HashTable}(m[e \mapsto \bot])\ ) \\
\\
\mathit{ReferenceCounter}(\mathit{counter} : A \rightarrow \mathit{Nat}) = \\
\qquad \sum_{t:T,p:P} .\ \mathit{protect}_{rc}(t,a,p)\ . \\
\qquad \mathit{ReferenceCounter}(\mathit{counter}[a \mapsto \mathit{counter}(a)+1] \\
\quad + \ \sum_{t:T,p:P} .\ \mathit{unprotect}_{rc}(t,a,p)\ . \\
\qquad \mathit{ReferenceCounter}(\mathit{counter}[a \mapsto \mathit{counter}(a)-1] \\
\quad + \ \sum_{t:T,p:P} .\ \mathit{protected}_{rc}(t,a,\mathit{counter}(a) \not\approx 0,p)\ . \\
\qquad \mathit{ReferenceCounter}(\mathit{counter})
\end{array}
$$

Table 20. mCRL2 specification of a thread p interacting with the term library.

$$
\begin{aligned}
&Thread(p:P,\ lm:T \to A_\bot) = \\
&\quad (\textstyle\sum_{t:T}.(lm(t) \approx \bot) \to Create(p,t,lm)) \\
&\quad + (\textstyle\sum_{t:T}.(lm(t) \not\approx \bot) \to Destroy(p,t,lm))
\end{aligned}
$$

Table 21. mCRL2 specification for the thread-safe term library.

$$
\begin{aligned}
&\textbf{allow}(\{ \\
&\quad construct\_term, destruct\_term, \\
&\quad contains, insert, delete, \\
&\quad protect, unprotect, protected, \\
&\quad skip,\ improbable, \\
&\quad enter\_shared\_call, enter\_shared\_return, \\
&\quad leave\_shared\_call, leave\_shared\_return, \\
&\quad enter\_exclusive\_call, enter\_exclusive\_return, \\
&\quad leave\_exclusive\_call, leave\_exclusive\_return, \\
&\quad \texttt{create\_call}, \texttt{create\_return}, \\
&\quad \texttt{destroy\_call}, \texttt{destroy\_return} \\
&\quad \}, \textbf{comm}(\{ \\
&\qquad construct\_term_{mm} | construct\_term_p \to construct\_term, \\
&\qquad destruct\_term_{mm} | destruct\_term_p \to destruct\_term, \\
&\qquad contains_{ht} | contains_p \to contains, \\
&\qquad insert_{ht} | insert_p \to insert, \\
&\qquad delete_{ht} | delete_p \to delete, \\
&\qquad protect_{rc} | protect_p \to protect, \\
&\qquad unprotect_{rc} | unprotect_p \to unprotect, \\
&\qquad protected_{rc} | protected_p \to protected, \\
&\qquad enter\_shared\_call_{bf} | enter\_shared\_call_p \to enter\_shared\_call, \\
&\qquad enter\_shared\_return_{bf} | enter\_shared\_return_p \to enter\_shared\_return, \\
&\qquad leave\_shared\_call_{bf} | leave\_shared\_call_p \to leave\_shared\_call, \\
&\qquad leave\_shared\_return_{bf} | leave\_shared\_return_p \to leave\_shared\_return, \\
&\qquad enter\_exclusive\_call_{bf} | enter\_exclusive\_call_p \to enter\_exclusive\_call, \\
&\qquad enter\_exclusive\_return_{bf} | enter\_exclusive\_return_p \to enter\_exclusive\_return, \\
&\qquad leave\_exclusive\_call_{bf} | leave\_exclusive\_call_p \to leave\_exclusive\_call, \\
&\qquad leave\_exclusive\_return_{bf} | leave\_exclusive\_return_p \to leave\_exclusive\_return \}, \\
&\qquad Thread(p_1) \mid\mid \\
&\qquad\quad \vdots \\
&\qquad Thread(p_{|P|}) \mid\mid \\
&\qquad MainMemory(\emptyset) \mid\mid \\
&\qquad HashTable(\lambda t{:}T.\bot) \mid\mid \\
&\qquad ReferenceCounter(\lambda a{:}A.\ 0) \mid\mid \\
&\qquad BF(\lambda p{:}P.Free)\ )\ )
\end{aligned}
$$

Table 22. Formulation of property 1: "A term and all its subterms remain in existence at exactly the same address, with unchanged function symbol and arguments, as long as it is not destroyed".

$$
\begin{array}{l}
\forall_{t:T}.\nu X(a : A_\perp = \perp,\ owners : FSet(P) = \emptyset). \\
\quad (\forall_{p:P,a':A}\cdot \\
\quad\quad [\texttt{create\_return}(p,t,a')]\ (\\
\quad\quad\quad X(a', owners \cup \{p\}) \\
\quad\quad\quad \wedge\ (a \not\approx a' \implies owners \approx \emptyset)\ )\ ) \\
\wedge\ (\forall p : P.\ [\texttt{destroy\_call}(p,t)]X(a, owners \setminus \{p\})) \\
\wedge\ [\ \ \overline{\exists_{p:P,a':A}.\texttt{create\_return}(p,t,a')} \\
\quad \cap\ \overline{\exists_{p:P}.\texttt{destroy\_call}(p,t)} \\
\quad]\ X(a, owners)
\end{array}
$$

Table 23. Modal formula for property 2: "Two stored terms t_1 and t_2 always have the same non-null address iff they are equal".

$$
\begin{array}{l}
\forall_{a:A,t_1:T}.\nu X(t : T = t_1, owners : FSet(P) = \emptyset). \\
\quad (\forall_{p:P,t_2:T}\cdot \\
\quad\quad [\texttt{create\_return}(p,t_2,a)]\ (\\
\quad\quad\quad X(t_2, owners \cup \{p\}) \\
\quad\quad\quad \wedge\ (t \not\approx t_2 \implies owners \approx \emptyset)\ )\ ) \\
\wedge\ (\forall_{p:P}.\ [\texttt{destroy\_call}(p,t)]X(t, owners \setminus \{p\})) \\
\wedge\ [\ \ \overline{\exists_{p:P,t':T}.\texttt{create\_return}(p,t',a)} \\
\quad \cap\ \overline{\exists_{p:P}.\texttt{destroy\_call}(p,t)} \\
\quad]\ X(t, owners)
\end{array}
$$

Table 24. Modal formula for property 3: "Any thread that is not busy creating or destroying a term, can always initiate the construction of a new term or the destruction of an owned term, *i.e.*, a term that this thread has exclusive access to".

$$
\begin{array}{l}
\forall_{p:P}.\ \nu X(busy : Bool = false, known : FSet(T) = \emptyset). \\
\quad (\neg busy) \rightarrow (\\
\quad\quad (\forall_{t:T}.(t \notin known) \implies [\tau^*]\langle\tau^*.\texttt{create\_call}(p,t)\rangle true) \\
\quad\quad \wedge\ (\forall_{t:T}.(t \in known) \implies [\tau^*]\langle\tau^*.\texttt{destroy\_call}(p,t)\rangle true)\ ) \\
\wedge\ [\ \ (\exists_{t:T}.\texttt{create\_call}(p,t)) \\
\quad \cup\ (\exists_{t:T}.\texttt{destroy\_call}(p,t)) \\
\quad]\ X(true, owned) \\
\wedge\ (\forall_{t:T}.\ [\exists_{a:A}.\texttt{create\_return}(p,t,a)]X(false, owned \cup \{t\})) \\
\wedge\ (\forall_{t:T}.\ [\texttt{destroy\_return}(p,t)]X(false, owned \setminus \{t\})) \\
\wedge\ [\ \ \overline{(\exists_{t:T}.\texttt{create\_call}(p,t))} \\
\quad \cap\ \overline{(\exists_{t:T}.\texttt{destroy\_call}(p,t))} \\
\quad \cap\ \overline{(\exists_{t:T,a:A}.\texttt{create\_return}(p,t,a))} \\
\quad \cap\ \overline{(\exists_{t:T}.\texttt{destroy\_return}(p,t))} \\
\quad]\ X(busy, owned)
\end{array}
$$

Table 25. Modal formula(s) for property 4: "Any thread that started creating a term or destroying a term, will eventually successfully finish this task provided there is enough memory to store one more term than those that are in use. But it is required that other threads behave fairly, in the sense that they will not continually create and destroy terms or stall other threads by busy waiting".

$$
\begin{array}{l}
([true^*]\forall_{p:P,t:T}.[\texttt{create\_call}(p,t)]\ \nu X_c.\ \mu Y_c.(\\
\quad \forall_{p':P}.(p \not\approx p') \implies \\
\quad\quad [\ \ (\exists_{t':T}.\ \texttt{create\_call}(p',t')) \\
\quad\quad \cup\ (\exists_{t':T}.\ \texttt{destroy\_call}(p',t')) \\
\quad\quad \cup\ \mathit{improbable}\ ]\ X_c \\
\quad \wedge\ [\ \ \overline{(\exists_{a:A}.\ \texttt{create\_return}(p,t,a))} \\
\quad\quad \cap\ \overline{(\exists_{p':P}.(p \not\approx p') \cap (\exists_{t':T}.\ \texttt{create\_call}(p',t')))} \\
\quad\quad \cap\ \overline{(\exists_{p':P}.(p \not\approx p') \cap (\exists_{t':T}.\ \texttt{destroy\_call}(p',t')))} \\
\quad\quad \cap\ \overline{\mathit{improbable}}\ ]\ Y_c \\
\quad \wedge\ \langle true^*.\exists_{a:A}.\ \texttt{create\_return}(p,t,a)\rangle true\ )\ ) \\
\wedge \\
([true^*]\forall_{p:P,t:T}.[\texttt{destroy\_call}(p,t)]\ \nu X_d.\ \mu Y_d.(\\
\quad \forall_{p':P}.(p \not\approx p') \implies \\
\quad\quad [\ \ (\exists_{t':T}.\ \texttt{create\_call}(p',t')) \\
\quad\quad \cup\ (\exists_{t':T}.\ \texttt{destroy\_call}(p',t')) \\
\quad\quad \cup\ \mathit{improbable}\ ]\ X_d \\
\quad \wedge\ [\ \ \overline{\texttt{destroy\_return}(p,t)} \\
\quad\quad \cap\ \overline{(\exists_{p':P}.(p \not\approx p') \cap (\exists_{t':T}.\ \texttt{create\_call}(p',t')))} \\
\quad\quad \cap\ \overline{(\exists_{p':P}.(p \not\approx p') \cap (\exists_{t':T}.\ \texttt{destroy\_call}(p',t')))} \\
\quad\quad \cap\ \overline{\mathit{improbable}}\ ]\ Y_d \\
\quad \wedge\ \langle true^*.\texttt{destroy\_return}(p,t)\rangle true\ )\ )
\end{array}
$$

D Benchmark Data

The benchmark tests and information shown in Figs. 4 and 5 are hard to read exactly. Therefore, we repeat the corresponding precise benchmark numbers in Table 28 up to and including Table 26. Each wall-clock time is measured in seconds.

The measurements in Table 27 came from benchmarking performed on an Intel i7-7700HQ processor. All other measurements were obtained through benchmarking on an AMD EPYC 7452 32-Core processor.

The benchmark results in Table 28 were obtained by having each thread create a term $t_{400\ 000}$, with t_0 being a constant, and t_{i+1} equal to $f(t_i, t_i)$. No garbage collection was performed during the benchmark. Note that only one copy of the term is actually stored in memory. So, most threads wanting to construct some term $f(t_i, t_i)$ detect that the term already exists, and only need to return its address, without actually creating it.

The benchmark results in Table 29 were obtained by having each thread create its own copy of the term $t_{400\ 000/\#threads}$, and measuring the wall-clock time. Note that although each thread creates its own term, all terms are stored in the data structures in an intermixed way. Note that as there is no sharing here, each thread stores a full copy of the term in memory.

The benchmark results in Table 30 and 31 were obtained by measuring the wall-clock time of creating $1000/\#threads$ instances of the terms used in Table 28 and 29. Before we start measuring the wall-clock times, the terms and subterms have already been inserted into the hash table, thus we are only measuring the cost of performing repeated lookups in our hash table. The experiment reported in Table 30 is the same as the one in Table 27, but the former is run on an AMD EPYC 7452 processor whereas the latter uses an Intel i7-7700HQ processor.

The benchmark results in Table 32 were obtained by having each thread perform $1000/\#threads$ breadth-first traversals of the term t_{20} and measuring the wall-clock time. The traversal does not make use of the shared structure of terms, meaning that approximately 10^9 term nodes are visited. Similarly, the benchmark results in Table 33 were obtained by having each thread perform $1000/\#threads$ breadth-first traversals of a term t_{20} that is unique for each thread.

We also measured the wall-clock time of the state space generation of the 1394 firewire protocol using a parallel prototype of the mCRL2 toolset. The results are listed in Table 26.

Table 26. Wall-clock time for state space exploration.

#Threads	1	2	3	4	5	6	7	8	9	10	11
parallel reference counter	61.0	87.6	90.1	83.1	67.0	57.4	56.2	53.3	50.0	46.5	43.4
parallel protection set	62.8	31.7	21.5	16.5	13.4	11.2	9.79	8.67	7.78	7.15	6.54
sequential reference counter	60.0										
sequential protection set	52.9										
original aterm library	40.2										
#Threads	12	13	14	15	16	17	18	19	20	21	22
parallel reference counter	43.3	41.4	38.8	38.1	37.3	35.5	35.2	35.2	33.9	33.2	32.5
parallel protection set	6.07	5.68	5.37	5.06	4.85	4.83	4.72	4.67	4.60	4.56	4.49
#Threads	23	24	25	26	27	28	29	30	31	32	
parallel reference counter	31.7	31.3	30.8	29.4	28.7	28.1	28.0	27.9	27.0	26.6	
parallel protection set	4.44	4.37	4.32	4.23	4.17	4.15	4.11	4.07	4.02	3.99	

Table 27. Wall-clock time for creating existing terms (shared, Intel).

#Threads	1	2	3	4	5	6	7	8
parallel reference counter	10.0	10.4	4.42	4.08	3.22	2.74	2.38	2.22
parallel protection set	5.68	3.09	2.36	1.60	1.52	1.30	1.14	1.02
sequential reference counter	4.61							
sequential protection set	4.20							
original aterm library	4.83							
parallel java	130	73.0	50.4	40.5	32.1	29.5	28.5	29.2
std::shared_mutex	10.3	38.1	42.1	41.0	40.7	41.7	42.7	46.2

Table 28. Wall-clock time for creating new terms (shared).

#Threads	1	2	3	4	5	6	7	8	9	10	11
parallel reference counter	0.03	0.11	0.22	0.14	0.26	0.34	0.28	0.29	0.39	0.51	0.53
parallel protection set	0.03	0.07	0.07	0.20	0.15	0.28	0.20	0.17	0.32	0.34	0.32
sequential reference counter	0.02										
sequential protection set	0.02										
original aterm library	0.01										
parallel java	0.26	0.46	0.68	1.32	1.25	1.20	1.51	1.41	1.36	1.48	1.51
std::shared_mutex	0.03	0.12	0.18	0.32	0.24	0.22	0.38	0.47	0.47	0.55	0.50
#Threads	12	13	14	15	16	17	18	19	20	21	22
parallel reference counter	0.51	0.59	0.50	0.54	0.49	0.53	0.54	0.48	0.5	0.48	0.52
parallel protection set	0.35	0.39	0.32	0.39	0.33	0.41	0.39	0.38	0.38	0.37	0.39
parallel java	1.40	1.51	1.44	1.43	1.38	1.67	1.77	1.87	1.77	1.78	1.85
std::shared_mutex	0.58	0.62	0.63	0.67	0.72	0.74	0.76	0.79	0.83	0.83	0.88
#Threads	23	24	25	26	27	28	29	30	31	32	
parallel reference counter	0.50	0.53	0.50	0.49	0.53	0.51	0.51	0.49	0.48	0.48	
parallel protection set	0.43	0.37	0.39	0.41	0.39	0.39	0.42	0.44	0.39	0.41	
parallel java	1.68	1.95	1.68	1.82	1.81	1.65	1.94	1.94	2.09	1.86	
std::shared_mutex	0.91	0.95	0.96	0.99	0.98	1.04	1.02	1.10	1.11	1.16	

Table 29. Wall-clock time for creating new terms (distinct).

#Threads	1	2	3	4	5	6	7	8	9	10	11
parallel reference counter	0.03	0.03	0.03	0.04	0.03	0.04	0.05	0.07	0.06	0.06	0.06
parallel protection set	0.03	0.05	0.03	0.04	0.03	0.03	0.04	0.05	0.05	0.05	0.06
sequential reference counter	0.02										
sequential protection set	0.02										
original aterm library	0.01										
parallel java	0.26	0.25	0.28	0.26	0.28	0.28	0.28	0.35	0.33	0.35	0.32
std::shared_mutex	0.03	0.04	0.04	0.05	0.10	0.04	0.04	0.09	0.09	0.09	0.09
#Threads	12	13	14	15	16	17	18	19	20	21	22
parallel reference counter	0.06	0.05	0.05	0.07	0.06	0.06	0.05	0.06	0.07	0.06	0.06
parallel protection set	0.05	0.06	0.06	0.05	0.06	0.07	0.06	0.05	0.06	0.06	0.06
parallel java	0.33	0.35	0.32	0.34	0.33	0.34	0.33	0.35	0.33	0.34	0.34
std::shared_mutex	0.09	0.10	0.10	0.10	0.10	0.09	0.11	0.09	0.12	0.11	0.13
#Threads	23	24	25	26	27	28	29	30	31	32	
parallel reference counter	0.07	0.06	0.05	0.06	0.06	0.07	0.07	0.07	0.06	0.06	
parallel protection set	0.06	0.06	0.05	0.07	0.06	0.06	0.05	0.05	0.05	0.06	
parallel java	0.37	0.34	0.35	0.35	0.37	0.35	0.34	0.34	0.35	0.34	
std::shared_mutex	0.13	0.11	0.14	0.09	0.15	0.18	0.15	0.13	0.16	0.13	

Table 30. Wall-clock time for creating existing terms (shared).

#Threads	1	2	3	4	5	6	7	8	9	10	11
parallel reference counter	9.01	35.1	11.5	19.0	9.81	10.6	6.40	6.00	4.90	4.85	3.88
parallel protection set	4.07	2.51	1.66	1.39	1.07	0.96	0.81	0.75	0.67	0.59	0.53
sequential reference counter	4.01										
sequential protection set	3.65										
original aterm library	4.57										
parallel java	104	136	106	103	91.7	84.2	75.5	71.1	64.5	61.6	57.5
std::shared_mutex	6.51	14.2	15.1	15.1	18.7	20.7	22.0	33.3	28.1	25.4	24.5
#Threads	12	13	14	15	16	17	18	19	20	21	22
parallel reference counter	3.51	3.10	2.86	2.77	2.66	2.45	2.61	2.43	2.33	2.27	2.17
parallel protection set	0.49	0.46	0.43	0.41	0.40	0.39	0.37	0.36	0.35	0.32	0.31
parallel java	54.3	51.9	49.4	48.7	46.5	47.7	47.9	48.1	48.6	47.7	46.6
std::shared_mutex	22.8	22.7	23.1	22.6	22.5	22.9	23.2	23.7	24.7	24.1	24.3
#Threads	23	24	25	26	27	28	29	30	31	32	
parallel reference counter	2.16	2.10	2.08	2.04	2.03	1.97	1.82	1.87	1.81	1.81	
parallel protection set	0.32	0.31	0.29	0.3	0.28	0.27	0.29	0.28	0.28	0.29	
parallel java	45.9	45.4	45.8	44.9	44.8	42.4	42.4	42.9	42.1	42.1	
std::shared_mutex	24.7	24.4	25.0	25.1	25.4	25.5	26.0	26.7	27.5	28.3	

Table 31. Wall-clock time for creating existing terms (distinct).

#Threads	1	2	3	4	5	6	7	8	9	10	11
parallel reference counter	9.02	5.15	3.95	3.05	2.64	2.25	2.38	2.41	2.42	2.49	2.24
parallel protection set	3.96	2.66	2.58	2.44	2.27	1.76	1.92	1.86	1.95	1.91	1.79
sequential reference counter	4.02										
sequential protection set	3.71										
original aterm library	4.58										
parallel java	106	212	218	227	260	266	274	276	295	272	287
std::shared_mutex	6.46	13.8	15.5	15.7	18.3	18.5	24.6	33.2	26.9	25.0	23.7
#Threads	12	13	14	15	16	17	18	19	20	21	22
parallel reference counter	2.22	2.27	2.26	2.20	2.39	2.55	2.63	2.58	2.72	2.67	2.67
parallel protection set	1.78	1.76	1.77	1.75	1.81	1.82	1.97	2.05	2.04	2.07	2.07
parallel java	275	287	296	2912	281	286	280	284	294	292	314
std::shared_mutex	22.7	22.4	22.9	22.6	22.2	22.7	23.2	23.8	24.7	24.1	24.1
#Threads	23	24	25	26	27	28	29	30	31	32	
parallel reference counter	2.69	2.69	2.77	2.76	2.8	2.77	2.82	2.84	2.86	2.92	
parallel protection set	2.07	2.10	2.15	2.05	2.12	2.11	2.17	2.22	2.27	2.22	
parallel java	311	308	315	316	324	330	339	332	343	352	
std::shared_mutex	24.4	24.4	25.3	25.3	25.9	25.7	26.3	27.1	27.8	28.7	

Table 32. Wall-clock time for traversing terms (shared).

#Threads	1	2	3	4	5	6	7	8	9	10	11
parallel reference counter	15.7	8.63	5.93	4.60	3.87	3.41	3.00	2.79	2.50	2.45	2.21
parallel protection set	16.7	8.90	6.07	4.66	3.93	3.37	3.01	2.80	2.55	2.41	2.34
sequential reference counter	16.8										
sequential protection set	18.2										
original aterm library	16.4										
parallel java	34.6	34.5	36.0	36.7	36.1	33.6	30.9	28.4	26.4	25.0	22.9
std::shared_mutex	16.2	8.71	5.95	4.54	3.86	3.34	3.01	2.74	2.53	2.40	2.29
#Threads	12	13	14	15	16	17	18	19	20	21	22
parallel reference counter	2.17	2.21	2.23	2.32	2.24	2.14	2.30	2.21	2.11	2.21	2.09
parallel protection set	2.28	2.21	2.30	2.35	2.21	2.26	2.35	2.25	2.33	2.28	2.21
parallel java	17.8	20.6	17.7	19.1	22.3	19.5	19.6	20.4	21.9	21.6	21.5
std::shared_mutex	2.25	2.33	2.28	2.24	2.21	2.22	2.29	2.15	2.24	2.04	2.24
#Threads	23	24	25	26	27	28	29	30	31	32	
parallel reference counter	2.17	2.18	2.13	2.07	2.06	2.09	2.06	2.05	2.13	2.10	
parallel protection set	2.26	2.15	2.12	2.16	2.13	2.07	2.09	2.16	2.03	2.03	
parallel java	17.6	19.2	18.4	20.6	22.7	20.8	18.5	22.5	23.6	19.4	
std::shared_mutex	2.24	2.12	2.18	2.05	2.05	2.2	2.16	2.17	2.02	2.07	

Table 33. Wall-clock time for traversing terms (distinct).

# Threads	1	2	3	4	5	6	7	8	9	10	11
parallel reference counter	18.4	9.61	6.41	4.93	4.10	3.56	3.14	2.88	2.63	2.51	2.34
parallel protection set	17.0	8.80	6.03	4.59	3.85	3.39	3.00	2.78	2.56	2.40	2.28
sequential reference counter	15.9										
sequential protection set	18.3										
original aterm library	17.4										
parallel java	34.5	34.2	35.8	37.0	35.5	33.8	30.1	28.3	27.1	23.4	21.9
std::shared_mutex	16.5	8.59	5.98	4.63	3.99	3.47	3.07	2.88	2.60	2.49	2.43
# Threads	12	13	14	15	16	17	18	19	20	21	22
parallel reference counter	2.36	2.33	2.33	2.28	2.24	2.19	2.26	2.22	2.16	2.20	2.11
parallel protection set	2.31	2.38	2.28	2.31	2.20	2.21	2.21	2.13	2.21	2.16	2.18
parallel java	21.1	17.5	20.9	17.3	18.7	20.8	23.0	18.4	21.6	23.0	22.8
std::shared_mutex	2.56	2.52	2.50	2.41	2.32	2.25	2.4	2.37	2.25	2.34	2.39
# Threads	23	24	25	26	27	28	29	30	31	32	
parallel reference counter	2.27	2.16	2.16	2.13	2.10	2.34	2.13	2.06	2.16	2.05	
parallel protection set	2.27	2.15	2.16	2.17	2.12	2.08	2.11	2.10	2.06	2.04	
parallel java	18.3	22.2	22.3	22.5	18.7	21.7	22.2	19.3	22.8	19.5	
std::shared_mutex	2.18	2.30	2.27	2.34	2.14	2.28	2.08	2.10	2.38	2.26	

References

1. Baeten, J.C.M., Weijland, W.P.: Process Algebra, Cambridge Tracts in Theoretical Computer Science, vol. 18. Cambridge University Press, Cambridge (1990)
2. Bergstra, J.A., Klint, P.: The ToolBus coordination architecture. In: Ciancarini, P., Hankin, C. (eds.) COORDINATION 1996. LNCS, vol. 1061, pp. 75–88. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61052-9_40
3. Blom, S., Lisser, B., van de Pol, J., Weber, M.: A database approach to distributed state-space generation. J. Log. Comput. **21**(1), 45–62 (2011). https://doi.org/10.1093/logcom/exp004
4. van den Brand, M., de Jong, H.A., Klint, P., Olivier, P.A.: Efficient annotated terms. Softw. Pract. Exp. **30**(3), 259–291 (2000)
5. van den Brand, M., Klint, P.: ATerms for manipulation and exchange of structured data: it's all about sharing. Inf. Softw. Technol. **49**(1), 55–64 (2007). https://doi.org/10.1016/j.infsof.2006.08.009
6. van den Brand, M., Moreau, P.E., Vinju, J.: Generator of efficient strongly typed abstract syntax trees in Java. IEE Proceedings - Software, vol. 152, pp. 70–78(8), April 2005
7. Cantrill, B., Bonwick, J.: Real-world concurrency. Commun. ACM **51**(11), 34–39 (2008). https://doi.org/10.1145/1400214.1400227
8. Fokkink, W., Pang, J., van de Pol, J.: Cones and foci: a mechanical framework for protocol verification. Formal Methods Syst. Des. **29**(1), 1–31 (2006). https://doi.org/10.1007/s10703-006-0004-3

9. Gao, H., Groote, J.F., Hesselink, W.: Lock-free dynamic hash tables with open addressing. Distrib. Comput. **18**(1), 21–42 (2005). https://doi.org/10.1007/s00446-004-0115-2
10. Gao, H., Groote, J.F., Hesselink, W.: Lock-free parallel and concurrent garbage collection by mark&sweep. Sci. Comput. Program. **64**(3), 341–374 (2007). https://doi.org/10.1016/j.scico.2006.10.001
11. van Glabbeek, R.J., Luttik, S.P., Trcka, N.: Branching bisimilarity with explicit divergence. Fundam. Informaticae **93**(4), 371–392 (2009). https://doi.org/10.3233/FI-2009-109
12. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. J. ACM **43**(3), 555–600 (1996). https://doi.org/10.1145/233551.233556
13. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press, Cambridge (2014). https://mitpress.mit.edu/books/modeling-and-analysis-communicating-systems
14. Groote, J.F., Keiren, J.J.A.: Tutorial: designing distributed software in mCRL2. In: Peters, K., Willemse, T.A.C. (eds.) FORTE 2021. LNCS, vol. 12719, pp. 226–243. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78089-0_15
15. Groote, J.F., Springintveld, J.: Focus points and convergent process operators: a proof strategy for protocol verification. J. Log. Algebr. Methods Program. **49**(1–2), 31–60 (2001). https://doi.org/10.1016/S1567-8326(01)00010-8
16. Hesselink, W., Groote, J.F.: Wait-free concurrent memory management by create and read until deletion (CaRuD). Distrib. Comput. **14**(1), 31–39 (2001). https://doi.org/10.1007/PL00008924
17. de Jong, H.A., Olivier, P.: Generation of abstract programming interfaces from syntax definitions. J. Logic Algebr. Program. **59**(1), 35–61 (2004). https://doi.org/10.1016/j.jlap.2003.12.002
18. Kats, L.C., Visser, E.: The spoofax language workbench: rules for declarative specification of languages and ides. SIGPLAN Not. **45**(10), 444–463 (2010). https://doi.org/10.1145/1932682.1869497
19. Krieger, O., Stumm, M., Unrau, R., Hanna, J.: A fair fast scalable rea, der-writer lock. In: 1993 International Conference on Parallel Processing - ICPP 1993, vol. 2, pp. 201–204 (1993). https://doi.org/10.1109/ICPP.1993.21
20. Lankamp, A.: https://github.com/cwi-swat/aterms/blob/master/shared-objects/src/shared/SharedObjectFactory.java. Accessed 2021; Last Changed 16 Dec 2009
21. Lev, Y., Luchangco, V., Olszewski, M.: Scalable reader-writer locks. In: Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures. SPAA 2009, pp. 101–110. Association for Computing Machinery, New York (2009). https://doi.org/10.1145/1583991.1584020
22. Luttik, S.P.: Description and formal specification of the link layer of P1394. In: Lovrek, I. (ed.) Proceedings of the 2nd International Workshop on Applied Formal Methods in System Design, pp. 43–56. University of Zagreb, Croatia (1997)
23. Mellor-Crummey, J.M., Scott, M.L.: Synchronization without contention. In: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS IV, pp. 269–278. Association for Computing Machinery, New York (1991). https://doi.org/10.1145/106972.106999
24. Milner, R. (ed.): A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10235-3

25. Prokopec, A., Bronson, N., Bagwell, P., Odersky, M.: Concurrent tries with efficient non-blocking snapshots. ACM SIGPLAN Not. **47**, 151–160 (2012). https://doi.org/10.1145/2145816.2145836
26. Raynal, M.: Concurrent Programming - Algorithms, Principles, and Foundations. Springer, Cham (2013). https://doi.org/10.1007/978-3-642-32027-9
27. van Spaendonck, P.H.M.: Verification of the busy-forbidden protocol (using an extension of the cones and foci framework) (2022). https://doi.org/10.48550/ARXIV.2208.05334
28. Sun, Y., Blelloch, G.: Implementing parallel and concurrent tree structures. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. PPoPP 2019, pp. 447–450. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3293883.3302576
29. Treiber, R.: Systems Programming: Coping with Parallelism. International Business Machines Incorporated, Thomas J. Watson Research (1986)
30. Umar, I., Anshus, O., Ha, P.: GreenBST: energy-efficient concurrent search tree. In: Dutot, P.-F., Trystram, D. (eds.) Euro-Par 2016. LNCS, vol. 9833, pp. 502–517. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43659-3_37