Faculty of Informatics, Masaryk University

State Space Compression for the DiVinE Model Checker

Bachelor's thesis

Vladimír Štill

Brno, Spring 2013

Declaration

Advisor: doc. RNDr. Jiří Barnat, Ph.D.

Abstract

The main focus of this thesis is on a reduction of memory requirements of an explicit-state LTL model-checker by use of tree compression. Presented technique is successfully applied to model checking of real world threaded C and C++ programs as well as some modeling formalisms dedicated for model checking.

Keywords

Model checking, DiVinE, Implementation, Tree compression, State-space explosion

Acknowledgements

First I would like to thank the Parallel and Distributed Systems Laboratory. It is great place to learn new things and to work on interesting problems, with nice people ready to help to a newcomer like me. Namely, I would like to thank doc. RNDr. Jiří Barnat, Ph.D. for inviting me to laboratory and advising this thesis, RNDr. Petr Ročkai for giving me insights into C++ and DiVinE and RNDr. Jana Tůmová for supporting and motivating my journey for knowledge.

I would also like to thank my family and my friends for supporting me and having patience with me.

Contents

Chapter 1

Introduction

Explicit-state model checking[1] is a well-established technique for verification of concurrent asynchronous processes. As multithreaded programs are common nowadays, the need to test and verify concurrent processes is increasing. Common testing methods, such as unit testing, are insufficient in this context, since asynchronous concurrency introduces nondeterminism. This nondeterminism is caused by interleaving of independently scheduled processes and makes testing inefficient: for example, unit tests may terminate successfully even though a failing run exists (it may in fact be less likely than succeeding runs). Explicit-state model checking can address this issue – it can verify that no run can violate a given property (such as an assertion violation or an LTL property).

However, model checking suffers from a problem called state-space explosion, which derives from the need to verify all runs of a system. Despite permanently growing sizes of random access memory available in contemporary 64-bit computers, the state-space explosion is still a major limitation for explicit-state model checking.

This limitation may become even more severe when verifying unmodified programs (in languages such as C and C++), as in the case described in [2]. Although sophisticated reduction techniques (such as presented in [14]) can be applied to reduce the state-state explosion caused by interleaving of multithreaded programs, state-spaces of real-world programs are still generally large, for example due to use of dynamic memory structures, recursion stack or due to the use of unnecessarily large integral data types.

Several state-space compression methods that could be used to handle this problem were introduced and applied to explicit-state model checking, for example Huffman compression, Collapse [8], and modelling-language-independent tree compression [11].

[1]Introduction to explicit-state model checking and automata based approach to LTL verification can be found in [6]. In addition, section 1.1 gives an introduction on model checking that is necessary for this thesis.

1.1 Model checking

Model checking is a formal verification method that, for a given system, checks that it satisfies a given property. Once the system and the property are stated the verification can be performed automatically by a model-checker, that is, a software tool for model checking.

The system can be viewed as a computer program in some programming language, traditionally a special language is designed for the purpose of model checking, even though model checking can also be performed on programs in general-purpose languages such as C or C++.

The property could be a formula in a temporal logic. Commonly used logics to describe properties are LTL, CTL and CTL*. Properties often used in model checking include assertion safety, deadlock freedom, response properties, etc.

1.2 Explicit-state model checking

Explicit-state model checking uses the fact that a system in any given moment of its execution can be fully described by the memory it is using – its state. Therefore, a run of the system can be viewed as a sequence of such states. As verified systems are usually non-deterministic – either as a result of asynchronous parallelism or as a feature of the programming language – all possible runs of such systems can be naturally expressed as a graph of all their reachable states, connected by directed transitions when one state can change into another without any intermediate state observed. This graph is then called the state-space graph of the system.

An explicit-state model-checker builds this graph and searches it for runs violating the property. Like many other graph traversing algorithms, a set of already visited states (the so-called closed set) needs to be used. As the state-space graph can be vast, storage of this set gives the problem of the state-space explosion – that is, memory requirements of model checking are much higher than memory requirements of a single execution of the system.

1.3 Aims and contributions of this work

The aim of this work is to introduce a version of tree compression capable of a substantial reduction in memory requirements of large state-spaces during model checking. This technique is general, in the sense that it can be applied independently of the modelling language, still resulting in good memory savings.

At the same time, we augmented the tree compression technique with an interface to the state-space generator, allowing optimisations based on the specific needs of a modelling language, providing even better compression.

The proposed tree compression technique can also be applied to LTL model checking algorithms, including parallel ones.

In this work we describe tree compression and provide implementation of it in the context of the parallel LTL model checker DiVinE, which is being developed in the ParaDiSe laboratory, at Faculty of Informatics Masaryk University. At the time of writing of this thesis the implementation was already integrated in development version of DiVinE[2] and is ready to be released with DiVinE 3.1, planned for summer 2013. The implementation is also available in an archive of Information System of Masaryk University, together with the text of this thesis.

Finally, this work also includes a benchmark of our tree compression technique on a wide range of models, including real-world C and C++ programs. Tree compression allows us to perform verification with up to 42 times less memory compared to the original, introducing virtually no time overhead. This shows great usability of the method.

[2]Development version of DiVinE can be obtained from its homepage, at https://divine.fi.muni.cz/download.html, either from the hydra build system, or from a darcs repository.

Chapter 2

Existing state-space reduction methods

As state-spaces for explicit-state model checking of asynchronous parallel communicating processes can be vast, several methods for reducing its memory consumption were introduced. This chapter reviews some of these methods, their relationship to tree compression and some notable implementations.

2.1 Partial order reduction

Partial order reduction exploits the fact that many of system executions are equivalent with respect to the verified property [1]. Using this observation, the state-space can be reduced by omitting some states in a way which preserves the property satisfaction.

Since the reduction achieved by partial order reduction is orthogonal to the savings from tree compression they can be combined to provide even better state-space reduction.

DiVinE contains an implementation of parallel partial order reduction [1].

2.2 Compact state-space representation

This section presents some techniques used to decrease the memory consumption incurred by storing the state-space of a verified program in an explicit-state model-checker, without the reduction of the number of processed states. Most of these techniques aim to represent the closed set compactly, sometimes the same technique can also be applied for openset representation.

The only of the following methods implemented in DiVinE is hash compaction. Please note that not all methods can be directly applied to DiVinE as some of them only provide insertion and membership test, while DiVinE also requires that additional information can be associated with a state and fetched for a given state (such the information may include, for example, a predecessor count).

2.2.1 Automata representation

As presented in [10], minimized deterministic automata can be used to represent the entire state-space of a system. This approach treats the state-space as a set of words over alphabet Σ, that is, $S \subseteq \Sigma^*$.

The approach presented in aforementioned paper expects fixed state sizes, and as such, it can not be directly applied to DiVinE (which does not use fixed-length states, e.g. in LLVM verification). For this reason, a different encoding method would be needed.

Furthermore, algorithms implemented in DiVinE require closed-set to behave as an associative map, which is difficult to represent by an automaton.

2.2.2 Huffman compression

This well-known generic method of compression can be applied to a state-space, either using statically defined compression tables or in a setup with learning runs.

This method was already implemented in DiVinE in the past [15], but it is not supported in DiVinE 3.0. It was also implemented in the Spin model-checker [9] with even better results [8].

Despite its expected memory savings, we decided not to implement Huffman compression for DiVinE as tree compression promises similar memory saving, a superior speed and easier integration in our environment.

2.2.3 Collapse and recursive indexing

Several compression methods are described in [8] and evaluated in the context of the Spin model-checker, most notably two versions of lossless Collapse compression method.

The first method suggests identifying components of state vector, such as processes and communication channels, and storing them in separate hashtables. State itself is then represented as global data plus indices of the separately-stored components. As stated in the aforementioned paper, a shortcoming of this method is that an upper-bound on the largest index for the state components must be known. Also, an implementation of this method with a growing hashtable would pose further challenges.

Additionally, an improved version of the Collapse, called recursive indexing, is discussed and evaluated in the same paper. This improved version allows index sizes to vary between states by saving index sizes in the global component of a state, which is now stored indirectly by index too. While recursive application of this method to processes and communicating channels is suggested in [8], it is not benchmarked there.

Since the COLLAPSE technique uses knowledge of a state layout to achieve its results, it requires an interface between state-space generator and the storage module. This may be a complication in cases where this interface is not present, for example when using an external module to generate the state-space, as with DIVINE CESMI.

2.2.4 Tree compression

Paper [11] presents recursive state compression (or tree compression) together with its evaluation in the LTSmin model-checker [12]. This technique is basically a modification of COLLAPSE method in which states are partitioned recursively. In the LTSmin implementation, the state is partitioned into slots of fixed size. Tuples of slots are then stored in fixed size hashtable, forming leaves of the tree, while references to those tuples are again grouped as tuples and stored, and so on.

The method we propose in this work combines this approach with COLLAPSE's ability to use the state layout and with the ability to use growing hashtables, which are required in DIVINE.

2.2.5 Hash compaction

Hash compaction, presented for example in [3], is a method which sacrifices completeness of the model checking procedure for reduced memory requirements. Instead of storing full states in the closed set, it stores only hashes of visited states and, if necessary, some associated information. If implemented carefully, it can lead to an algorithm which does not give false-negative answers, that is, if the algorithm finds a counterexample than this counterexample witnesses violation of the verified property. However, the algorithm can miss some counterexamples as some states (those with equal hashes) will be merged during verification.

Since hash compaction tackles the state-space explosion by omitting state information it can be viewed as a lossy compression method, and as such it cannot be used together with tree compression. Moreover, since successors are generated from the open set, hash compaction cannot be used for open set compression.

2.3 Modeling language aware methods

In some cases, better reductions can be achieved using methods which exploit specific nature of a modeling formalism. Such a reductions are used for example in LLVM interpreter in DIVINE and presented in [14].

Similarly to partial order reduction, those techniques usually reduce the number of states, are therefore orthogonal to tree compression and can be combined well with it.

2.4 Distributed verification

Network-connected workstations can be used to bring more space and computing power for verification of large systems. This approach was actually the original motivation for the first version of DiVinE and is still supported in DiVinE 3.0, even though the availability of 64-bit multicore processors, large amounts of affordable RAM and vast computing power in a single machine allow verification of big systems even without this extension.

As distributed memory access is much slower than local (shared) memory access, it is necessary to apply tree compression only locally, on each workstation separately. Communication between workstations uses uncompressed states. Although tree compression will get less efficient in a distributed memory environment, it is still possible to combine those approaches. However, this combination is not part of this thesis.

Chapter 3

DiVinE

The DiVinE model-checker [4] is an explicit-state model-checker designed to utilize parallelism in both shared memory and distributed memory setting. It supports safety and liveness LTL properties and supports multiple input formats, including DVE, LLVM bitcode (which can be automatically created from C or C++ source by DiVinE using the CLang compiler), UppAal timed automata format, and CoIn. It also provides a Cesmi loader which allows models to be represented as shared libraries, loaded by DiVinE at runtime and used to generate the state-space – this allows easy integration of new input formalisms, possibly by external developers.

The source code of DiVinE is freely available from a version control repository[1] and from the web page of DiVinE [5], under BSD licence.

3.1 Architecture

DiVinE is written in C++ and since version 3.0 it utilises C++11 language features. Its architecture is modular, with modules connected together using C++ templates, in contrast to the common approach of using inheritance and virtual calls. This design allows tighter integration of components at compile time, therefore, more code can be inlined which results in faster execution.

DiVinE consists of several algorithms, each represented as a separate module, building on other modules, such as visitor modules and store modules. Visitor modules implement several graph traversal algorithms, such as DFS, BFS, and pseudo-BFS. Pseudo-BFS is used in parallel verification algorithms. It does not guarantee particular order of traversal and is not deterministic. Store modules are used by visitor modules and algorithms to represent the closed set in graph traversal. Each store is a hashtable-like module with support for insertion and retrieval of states and various support operations.

Since implementation of tree compression required some changes in the architecture of DiVinE, several technical aspects of DiVinE 3.0 will now

[1]available at http://divine.fi.muni.cz/darcs/mainline/

be presented. Those aspects are necessary for understanding the changes required by tree compression.

All the following sections are based on sources of DiVinE 3.0 if not explicitly stated otherwise. Sources are available in a version control repository[2].

3.1.1 Generators

As input formats for DiVinE are usually programs in some formalism, it is necessary to generate the state-space for explicit-state model checking from them. This generation is performed on the fly by a graph generator, starting from an initial state. With the exception of the Cesmi generator[3] all generators are input-format-specific. They provide a common interface, most notably the functions `successors` and `isAccepting` which are responsible for generating successors of given state and deciding whether a state is accepting, respectively. Currently all input formats are implicit, that is, the state-space has to be generated by those functions, and is not stored as part of the input.

Each state (that is, a vertex of the state-space graph) is represented by an object of type `Blob`, which points to a flat piece of memory that can be read from and written to, and whose size can be obtained. State memory is separated into two parts (where the first is optional): slack and the system state. While slack is used only by algorithms and never by the generator, the opposite is true for the system state. Only the system state is hashed when the state is stored, and it never changes once the state is generated.

3.1.2 Parallelization of algorithms

In DiVinE, algorithms are parallelized by using visitors which provide pseudo-BFS traversal of graph to algorithms. Currently, two types of parallel visitors are available: the partitioned visitor and the shared visitor.

The partitioned visitor works with a static partitioning of states among worker threads, based on the hash of each state. Most operations are then performed in a thread-local fashion, for example each thread has its own hashtable. In this setting, threads communicate using IPC queues of edges (queues transfer *from* state and *to* state of an edge). This approach can be easily extended to multiple machines using MPI. The disadvantage of this approach is that static partitioning may cause different loads for different threads.

The shared visitor is a new experimental feature of DiVinE which is under heavy development [16]. It uses a shared queue and a shared hashtable, providing faster parallel pseudo BFS exploration. It currently cannot be combined with MPI.

[2]DiVinE 3.0 repository is located at http://divine.fi.muni.cz/darcs/branch-3.0/

[3]The Cesmi generator is used as an interface to external generator provided by a shared library.

3.1.3 The interface between visitors and algorithms

Since all algorithms implemented in DiVinE share the basic approach to graph traversal, that is for particular state they are looking at its outgoing edges and then (if necessary) at the states these edges lead to, this is abstracted in an interface between a visitor and an algorithm. This interface works as follows: the graph is traversed in an order specified by the visitor (DFS, BFS or pseudo-BFS order) and each edge (leading from an already processed state *from* to a target state *to*) is processed:

1. edge is processed by the function `transitionHint`, provided usually by the visitor itself,
2. if the edge is not ignored, the *to* state is fetched from the store (if it is already stored),
3. the edge is processed by the function `transition`, provided by the algorithm,
4. if the transition is not ignored, the *to* state is stored in the store,
5. if the transition is to be followed, the *to* state is passed to the `expansion` function provided by the algorithm,
6. finally, the slack of the *to* state is updated in a hashtable, if necessary (used by hash compaction).

Each of the aforementioned functions is a static function of an algorithm or a visitor and a working instance of the algorithm is passed as the first argument to all algorithm calls.

This mechanism is implemented using C++ templates, allowing both maximal code reuse at design time and maximal optimization at compile time.

3.1.4 Memory management

Memory management in DiVinE 3.0 is quite straight-forward, each state of graph is either temporary or permanent and this is tracked by a single bit inside `Blob`'s header. A state is generated as temporary and it becomes permanent once stored in the store. Permanent states are never deallocated during a run of DiVinE and their location in memory never changes.

This memory management is simple, imposes virtually no overhead and is sufficient for most use cases as of version 3.0. The fact that permanent states are freed only on termination does not incur any overhead, since a single run of DiVinE can verify only one property of one model.

3.1.5 Stores

DiVinE 3.0 supports three store types: a partitioned store, a shared store and a hash-compacted store. The partitioned store is used by the partitioned visitor. It has one hashtable for each thread and stores full states. The shared store is an experimental version used by the shared visitor. It is optimized for shared memory, using only one hashtable which is shared across all threads. The hash-compacted store is derived from the partitioned store and implements hash compaction [3], a lossy compression method that stores only hashes of states in the closed set, making memory requirements of explicit-state model checking lower. Hash-compacted store does not support counterexample generation and is currently only suitable for reachability analysis.

3.1.6 Counterexample generation

Counterexamples are (in DiVinE 3.0) generated using parent pointers saved into the slack part of a state by algorithms. When an algorithm terminates and detects a property violation, it starts counterexample generation, passing either accepting state (in case of reachability analysis) or state on an accepting cycle (in case of LTL verification) to a counterexample-generating algorithm.

In the case of reachability (where counterexample is just a path to a property-violating state), it is sufficient to track parent pointers back to the initial state and save each state into the counterexample.

In algorithms for LTL verification, counterexamples have a lasso shape containing a cycle going through an accepting state of the product automaton and a path from initial state to this cycle (as implemented in DiVinE, the path leads to an accepting state on this cycle). Those counterexamples are generated in two phases. First, the path from the accepting state to the initial state is traced as in the reachability case. Then, parallel BFS is run again from the accepting state and parent pointers are updated. Finally, the parent pointers are traced from the accepting state back to itself, generating the cycle part of the counterexample.

Note that this approach requires parent pointers to be valid when searching for a counterexample. This assumption holds for traditional uncompressed stores present in DiVinE 3.0, as *from* states are already saved in a hashtable (and therefore permanent) when their pointer is saved as the parent pointer in the *to* state.

Chapter 4

Tree compression

With traditional hashtable approach to explicit-state model checking, full states are saved. However, this is not necessary, as in most cases only small part of state changes from between a state and its successors. Several methods to take advantage of this fact were introduced, for example Collapse in Spin model-checker [8] and tree compression with a binary tree in LTSmin model-checker [11].

The method described in this thesis is inspired by both aforementioned methods, while at the same time it integrates well with DiVinE and is optimized for very large state-spaces.

4.1 Representation of states

Instead of the traditional method where states are represented as byte vectors, tree compression represents them as a tree with parts of the state vector in leaves. As both leaves and internal nodes of the tree are saved in a hashtable, they can be naturally reused, leading to a memory efficient representation of the state-space, where leaves and internal nodes of the tree can be shared among trees of different states (or even inside state).

In our implementation, the tree representation of a state can have arbitrary branching and different trees can store states with different sizes. The latter is required, among others, by the LLVM interpreter used in DiVinE.

Figure 4.1 shows the layout of a single state represented as a tree. We store roots of the tree, the internal nodes and the leaves in distinct tables, marked as roots, forks and leaves in the figure. This allows us to unambiguously connect each state to its corresponding root and store the state's associated information in the root. The associated information cannot be saved separately from the root, as correctness of DiVinE's algorithms relies on this information being state-local (and information is updated during a verification run). Size of a state is not saved in the root as that would be redundant, since all leaves must know their size and leaves can be unambiguously identified in the tree.

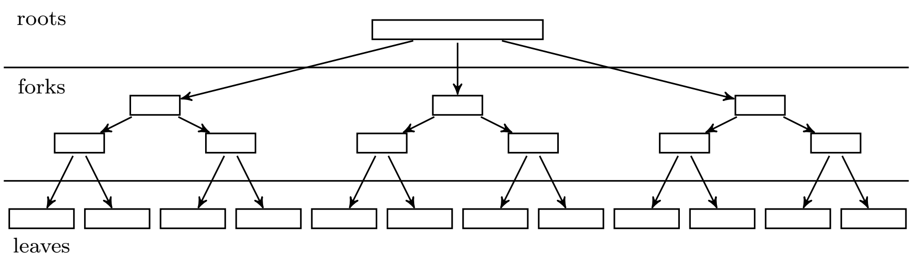


Figure 4.1: Tree representation of state.

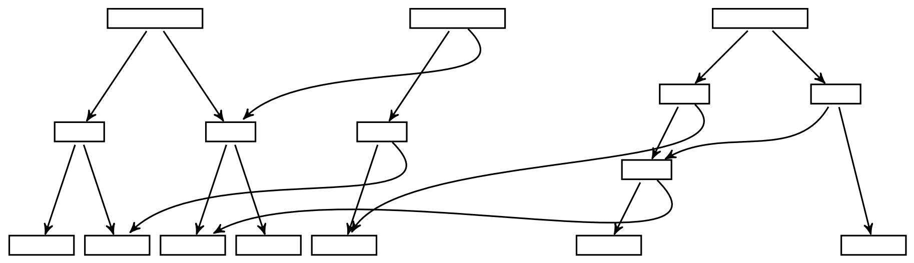


Figure 4.2: Subtree sharing in tree compression.

Figure 4.2 illustrates subtree sharing between states. Note that there are no limits for subtree reusal in our tree compression, subtrees can be reused even inside one state or across states with different sizes (therefore tree compression is actually a somewhat misleading name, since states are in fact represented as directed acyclic graphs).

4.2 Requirements

In DiVinE, it was a requirement that tree compression will integrate well with most already supported features such as parallel verification, counterexample generation or partial order reduction. Also, it is expected to work without special settings coming from the user (with the sole exception of enabling it). Therefore, it is important that tree compression must allow closed set to grow, as DiVinE supports hashtables that grow during parallel verification, in contrast to some model-checkers (such as Spin or LTSmin) who require the user to provide an upper bound on the number of states.

When compared to the implementation of tree compression in LTSmin, we wanted to relax its behavior by allowing trees to be not only binary but with arbitrary branching. Furthermore, we wanted support for generators to provide hints how to partition states well (for example cutting states on process boundaries), whereas the abovementioned solution splits states into fixed-size chunks.

It must also be possible to integrate tree compression with verification using a shared, growing hashtable, which has been developed in parallel with tree compression (more about these shared growing hashtables in DiVinE can be found in [16]).

Finally, tree compression in DiVinE is required to work well with very large state-spaces (that is, tens of millions of states, or even billions and more). Such big state-spaces cannot be usually (without compression) explicitly verified using just one single-cpu workstation since they require hundreds of gigabytes of RAM.

4.3 Design and implementation

Because of the aforementioned requirements, tree compression was written from scratch, not using an existing implementation of a similar technique.

The compression support itself consists of two main parts, a tree-compressed hashset and a tree-compressed store. The tree-compressed hashset is itself based on an arbitrary hashset which provides the `insertHinted` and `getHinted` operations. This design choice allowed me both to fully re-use the current implementation of a hashset used for partitioned verification, as well as to easily support experimental versions of the shared hashset.

Following sections summarize the implementation of tree compression in DiVinE 3.1. As the changes leading towards tree compression support were performed inside DiVinE's main repository (so-called mainline[1]), which is constantly changing, I created a branch capturing the state of this repository at the time of writing of this thesis. It is included with this thesis and available online[2].

Implementation of tree compression showed some weaknesses in the current workflow in DiVinE. Namely, memory management had to be improved to facilitate the changes in counterexample generation. Those changes allowed generation of counterexamples with both tree compression and hash compaction. As a by-product of those changes, the system of stores was reworked. Similarly, the integration of tree compression into the instantiation of algorithms reached limits of the pre-existing system and therefore a new, more flexible system of instantiation was created[3].

[1]DiVinE mainline repository is available at http://divine.fi.muni.cz/darcs/mainline/

[2]DiVinE 3.1 Darcs repository with tree compression implemented as of this thesis is available on http://paradise.fi.muni.cz/~xstill/darcs/divine31_thesis/

[3]Since instantiation of algorithms is largely unrelated to tree compression it will not be described here. Curious readers will find instantiation in `divine/instantiate/` directory inside mainline repository (most notably `divine/instances/instantiate.h` and `divine/instances/definitions.h`, previous versions can be found in history of repository).

4.3.1 Tree-compressed hashset

Tree-compressed hashset, represented by the class `NTreeHashSet`[4] is the core of tree compression. It implements the compression itself while exporting an interface similar to that of hashsets already present in DiVinE.

The header of the `NTreeHashSet` declaration looks as follows:

```
template< template< typename, typename >
                   class HashSet,
             typename Item, typename Hasher >
struct NTreeHashSet;
```

It is parametrized by the underlying hashset, by the type of the item being stored and by a class providing hashing and equality tests for objects of type `Item`. It should be noted that the `Item` type must provide several methods, currently supported only by the `Blob` type. Those methods are `size` and `data` returning size of an object in bytes and its (continuous) content as a pointer of the type `char*`, respectively.

Tree-compressed hashset utilizes three hashsets (of the base type given in the `HashSet` parameter). These hashsets store different parts of the tree – roots of the tree including the slack are stored in the *roots* set, internal nodes are stored in the *forks* set and finally, leaves of the tree are in the *leaves* set. This setup allows maximal reuse of internal nodes and leaves already created by previous inserts, while at the same time gives tighter type control (those tables store pointer to different types, since each tree node type requires different metainfomation).

Each state is stored as a tree identified by an instance of class `Root`. Instances of this class support direct access to the slack (subsection 3.1.1) and they can be reassembled without further support from an `NTreeHashSet` instance. For internal purposes, roots also support enumeration of the leaves of the entire tree, an operation which is currently unused outsize of `NTreeHashSet`, but could be used, for example, for serialization of compressed states over MPI.

If the system state is shorter than a given lower limit for compression (provided by the state-space generator), it is stored entirely in roots table – in this situation, tree compression actually incurs a slight space overhead, as it has to store additional information in the root of the tree, as well as the state itself in an uncompressed form. In practice, this situation will usually imply that model itself is relatively small and can be easily verified without tree compression, while at the same time the overhead of tree compression will not matter (also, those models are not the primary interest in tree compression, as mentioned earlier).

In all the other cases, a model state is partitioned and stored using all three hashtables. States are inserted using the function `insertHinted`, with

[4] `divine/toolkit/ntreehashset.h`

the following prototype:

```
template < typename Generator >
std::tuple< Root*, bool > insertHinted( Item item,
    hash_t hash, Generator& generator );
```

This function requires the state itself, its hash, and a generator responsible for the state; it returns a tuple of the (possibly new) root and a boolean flag indicating whether the state was freshly inserted or already present. The generator (subsection 3.1.1) provides hints on a desired partitioning of the state. The interface between the generator and `NTreeHashSet` is provided by the following functions implemented within the generator:

```
template< typename Yield >
void splitHint( Node n, Yield yield );
template< typename Yield >
void splitHint( Node n, int form, int length,
                Yield yield );
```

The first of these function overloads is basically a shortcut for top-level splitting of the entire model state, therefore we can focus on the second form. The `splitHint` function uses the generator pattern, expecting its `yield` parameter to be a callable object[5]. A range of bytes of the state `n`, together with a flag indicating whether this range is to be stored as a leaf and the number of remaining partitions is passed to `yield`. All other arguments of `splitHint` are quite self-explanatory. The default implementation of `splitHint` is provided in `divine/graph/graph.h`; this implementation creates a balanced binary tree and is generator-agnostic. Specialized versions can be implemented in a particular generator and will be used automatically if available.

The implementation of `insertHinted` calls `splitHint` and a lambda function is passed as the yield argument, allowing easy recursive generation of n-ary tree, according to a shape defined by the generator. An advantage of this approach is that the tree is created on-the-fly, without any intermediate data structures (the overhead of recursive calls is presumably small, as C++ compilers perform massive inlining and depth of the tree should be at most logarithmic in the size of the state in any reasonable implementation of `splitHint`). Using recursion, `splitHint` is called until it reaches the bottom of the desired tree shape. At this moment, a `Leaf` instance is allocated and the given range is copied to it from the state. The leaf is then inserted into the leaf set, which returns an equivalent, but permanent, leaf. This permanent leaf is then used on a higher level of recursion, to construct a fork (forks store pointers to permanent leaves, or to other forks), and forks are saved into the forks table in a similar fashion. As the recursion backtracks to the root, the entire tree is created, and finally the root is inserted. Each time an element

[5]In C++11 callable objects are functions, objects that implement `operator ()`, lambda functions and a few others, such as results of calls to `std::bind`.

is inserted to one of the tables, the original element is discarded if it was already present in the table (if the element is not present prior to insertion, it is equivalent to a permanent element and therefore cannot be freed). Finally, a permanent root element is returned together with a flag indicating whether it was already present (a state is present in `NTreeHashSet` if and only if a corresponding root is present).

As an optimization, roots are stored using the hash of the entire state, not of the binary representation of the `Root` object. This allows fetching roots from the `NTreeHashSet` corresponding to a given state, without repeating the aforementioned construction of the tree, which is allocation-heavy (a state can be compared to a tree representation almost without allocation – only a stack for tree traversal has to be allocated – but a tree is constructed from a state with allocations at least as big as the state itself).

As already mentioned, the vertices of the tree are connected using pointers, leading in direction from root to leaves. This approach has obvious disadvantage of memory overhead incurred by 64 bit pointers (as DiVinE is usually run in 64 bit environment). On the other hand, it allows easy integration of tree compression with growing hashtables, as well as reconstruction of a state from a tree without the assistance of the hashtable. Using indices to the hashtable would require trees to be recreated each time any of the tables is resized, slowing down resizing and complicating the design (tree compression would have to be tightly integrate with the hashtable, whereas in our current approach, any table with the given interface can be used). Furthermore, leveraging the advantage of shorter indices (as compared to pointer width) would require the trees to be parametrized by the size of the index. Finally (as mentioned in section 4.2), we are aiming at very large state-spaces, possibly with billions of states; this would either require indices to be stored in variables whose size is not a power of two (which is quite impractical) or in 64 bit variables, therefore gaining nothing on memory efficiency.

4.3.2 Memory management and stores

As mentioned in subsection 3.1.4, memory management in DiVinE 3.0 is quite simple. However, it makes some assumptions that no longer hold for compressed states. In particular, it is assumed that when a *from* state is processed by an algorithm in `transition` (item 3 in subsection 3.1.3), it is permanent and its memory location can be saved inside *to* state of the transition for later use in counterexample generation. This assumption no longer holds for compressed states, since the state is no longer saved in one unchanged piece, but it is still processed in this form by algorithms and visitors (therefore states visible in algorithms are temporary). Note that this is also the problem which caused hash compaction to be unable to provide counterexamples in DiVinE 3.0.

To solve these issues and to allow easy integration of new types of compression, two new types that wrap a state were introduced. These types depend on the type of the store and the generator and are implemented as nested types of the store. They have a common interface, independent of a store type.

`Vertex` represents both a (possibly temporary) full state suitable for successor generation and a permanent state saved in a hashtable. It is passed to algorithms when processing transitions and states of model. All uses of slack must access the stack in a permanent state. Objects of type `Vertex` must be convertible to `VertexId`.

`VertexId` represents a permanent state. It contains slack, and provides access to it. `VertexId`s are processed by algorithms when iterating over an entire hashtable (used in OWCTY and MAP). Objects of the type `VertexId` are generally not suitable for successor generation, but support for their conversion to `Vertex` may be provided (if compression is lossless).

Implementation of the aforementioned objects for traditional uncompressed store is trivial, they are simple wrappers around the state type provided by the generator.

For tree compression, `VertexId` is a wrapper around a pointer to the root of the tree (which is stored in the roots table inside `NTreeHashSet`, and provides conversion to `Vertex` since tree compression is lossless). `Vertex` then contains both an uncompressed state as provided by the generator and a pointer to the root of the tree.

For hash compaction, `VertexId` is an object which contains just the slack and a hash of the state, while `Vertex` additionally keeps the state in full. As hash compaction is lossy, conversion from `VertexId` to `Vertex` cannot be provided for hash compaction.

Algorithms were modified so that they always access the slack saved in the hashtable and, when they need to save an identifier of another state (for example for predecessor tracking), they must use `VertexId` for this purpose. This incurs no memory overhead as `VertexId` contains only a single pointer – same as `Blob`, which is used to represent states in all current generators.

4.3.3 Compressed queues

Compression of the closed set as mentioned above would certainly help to reduce memory requirements, but left alone, much space would still be wasted.

This is caused by queues which represent the open set for most algorithms. Those queues can get quite long (the number of distinct states in them is only bounded by the number of states, or more precisely, by width of the state-space graph, although a state can be present multiple times in a queue).

Queues originally saved full states, but in presence of tree compression this is not necessary. For IPC queues, the *from* state can be compressed and local queues can be fully compressed (as they store only *from* states).

Queue compression is activated by store, which defines the `QueueVertex` type, which is a type alias for either `Vertex` or `VertexId`. It is required that `QueueVertex` can be converted to `Vertex` and vice versa. For tree compression `QueueVertex` is defined as an alias to `VertexId` as it can be decompressed. For hash compaction it must be defined as an alias to `Vertex`, because hash compaction is lossy. Decompression is then performed on-the-fy in the queue.

Prior to this optimization, queues (in presence of tree compression) contained full states which were not saved in stores, leading to high memory overhead. Now queues store just `VertexId`s, each the size of a single pointer.

Please note that this optimization is easily applicable to BFS based exploration strategies (which are used by all algorithms in DiVinE with the exception of NestedDFS). Application to DFS would require successors to be saved in the closed set before they are pushed to the stack which would require modification of the DFS algorithm used in DiVinE.

In the case of distributed verification, it would be necessary to decompress `QueueVertex` values when sending it to different machines, as the compressed representation is only a pointer, and would be invalid when interpreted on a different machine.

4.3.4 The interface between visitors and algorithms

Slight modification of the interface between visitors and algorithms was required, as algorithms must be able to access `VertexId` for both *from* and *to* states of a transition. The new workflow looks like this:

1. the transition is processed by the function `transitionFilter`, provided by the visitor,
2. if the transition is marked to be ignored, its processing is abandoned,
3. the *to* state is inserted into the store, the store returns a `Vertex` associated with this state and a boolean flag indicating whether the state was already present,
4. the transition is processed by the function `transition` provided by the algorithm (it gets a `Vertex` objects for both *from* and *to* states),
5. if the transition is to be followed, `Vertex` associated with the *to* state is passed to the function `expansion`, provided by the algorithm.

The main difference is that the *to* state is stored before the `transition` function is called. Also, only one call to the store is now necessary, as the `store` procedure returns an up-to-date version of the `Vertex` and all modifications of the slack are performed directly in the version stored in the table.

4.3.5 Counterexample generation

As mentioned in subsection 3.1.6, counterexample generation in DiVinE 3.0 made some assumptions, which no longer hold for a compressed state-space.

The new approach, implemented for DiVinE 3.1, takes advantage of `VertexId`s as permanent identifiers of states, as means to track parents of states.

The algorithm has to cope with the fact that a counterexample is generated from a sequence of full states, whereas `VertexId`s are just identifiers for those states. It would be possible to use only a slightly modified version of the original algorithm which would first reassemble `VertexId` to a `Vertex` and than continue with full state. However, this approach would not be suitable for hash compaction.

The new algorithm generates each trace in two phases. First, it traces `VertexId`s back to the initial state (they can be traced as parent `VertexId` is saved in slack, which is accessible via `VertexId` alone). Comparison to the initial state must be performed in the thread responsible for given state, i.e. the thread which stores it in its hashtable. The output of this first phase is a sequence of `VertexId`s from initial state to target state. In the second phase, a counterexample built from full states is generated using this sequence. This can be done by starting in the initial state (which can be always generated again) and following the path encoded in the sequence of `VertexId`s. Each (possible) successor in the trace is compared to a matching `VertexId` and successors leading outside the trace are ignored. Please note that comparison to a `VertexId` must again happen on the thread which owns the state.

Cycle traces can be generated in a similar fashion, the main difference is that accepting state is used in place of the initial state of the graph (this is again preceded by parallel BFS which updates parent pointers, as in original algorithm).

4.4 Integration with parallel visitors

As mentioned above, the standard approach to parallel verification in DiVinE uses a partitioned setup, i.e. states are statically assigned to threads (using their hash); this setup can be optionally extended to a network of workstation.

In this setup, tree compression can work only on a per-thread basis, compressing each partitioned hashtable independently (access to hashtables is not threadsafe in this setup). This means tree compression will become less efficient as the number of threads used for verification rises.

Additional memory overhead will be incurred by IPC queues which cannot be compressed fully, as *to* states are not stored yet (and they belong to different thread and therefore cannot be saved before they are sent to queue).

This disadvantage will be eliminated in future releases of DiVinE as a new mode of verification (based on hashtables shared by all threads on a

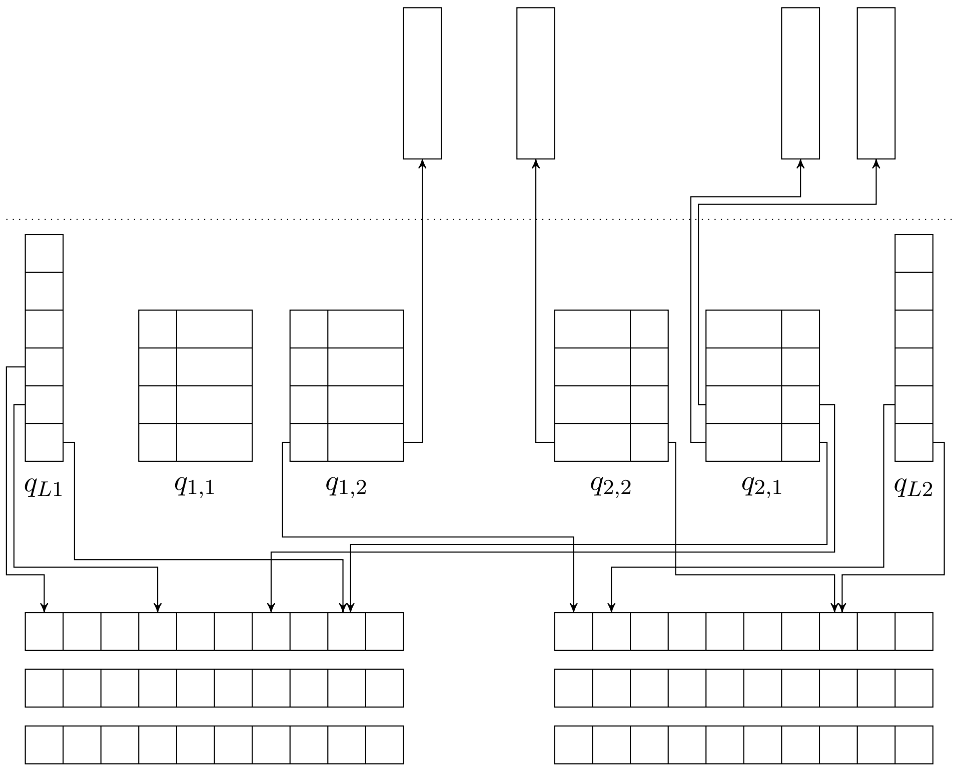


Figure 4.3: Partitioned visitor with tree compression.

given machine) becomes the default [16]. Experimental implementation of the shared hashtable was already integrated with tree compression.

Figure 4.3 shows a scheme of two worker threads running, using a partitioned visitor. Each thread has its own local queue and it also has an IPC queue for each other worker, including itself. While the local queue contains only *from* states, IPC queue contains entire transitions, that is both a *from* and a *to* state. Arrows in the figure show pointers to data[6], at the top – above the dotted line – are temporary data, representing the overhead of using a partitioned visitor. Tree-compressed hashtables are at the bottom of the figure, while queues are in between. q_{L1} and q_{L2} are local queues and $q_{1,1}, q_{1,2}, q_{2,1}$ and $q_{2,2}$ are IPC queues. Note that *from* states in an IPC queue belong to the hashtable of their owner – for example *from* states in $q_{2,1}$ are processed by worker 2 but they are compressed and saved by worker 1. Therefore, it must be possible to decompress state without access to a hashtables of other workers (otherwise all workers would need to have access to all hashtables).

[6]There is actually small simplification as hashtable does not store data directly and compressed queues points not to hashtable but to memory location of data itself, but it can be viewed as accessible through hashtable as there exist at most one instance of each compressed state in each worker.

Chapter 5

Experiments

5.1 Settings

Several measurements were performed comparing tree compression implementation in DiVinE to the default verification mode of DiVinE, that is without any compression. All memory usages are peak virtual memory as reported by DiVinE in the *Memory-Used* field of the `-r` report; this is the maximal amount of memory addressable during a given run. Time is *Wall-Time* value from the report, that is the real time of the entire run. If a test did not finish within given limits, it is marked by a dash in the result table.

If not stated otherwise, DiVinE was executed as follows

```
$ divine verify <algorithm> -r --max-memory=<M> -w 1 \
  [ --compression=ntree ] <model>
```

where algorithm is one of the supported algorithms (`--reachability`, `--owcty`, `--map`, `--nested-dfs`); `M` is a memory bound for the run and `-w 1` signifies that DiVinE is running with one thread (to show maximal compression, more about threading with tree compression in section 4.4).

Models for experimental evaluation came from several sources:

DiVinE distribution tarball several examples are includes in DiVinE itself; the LLVM and timed automata examples were used,

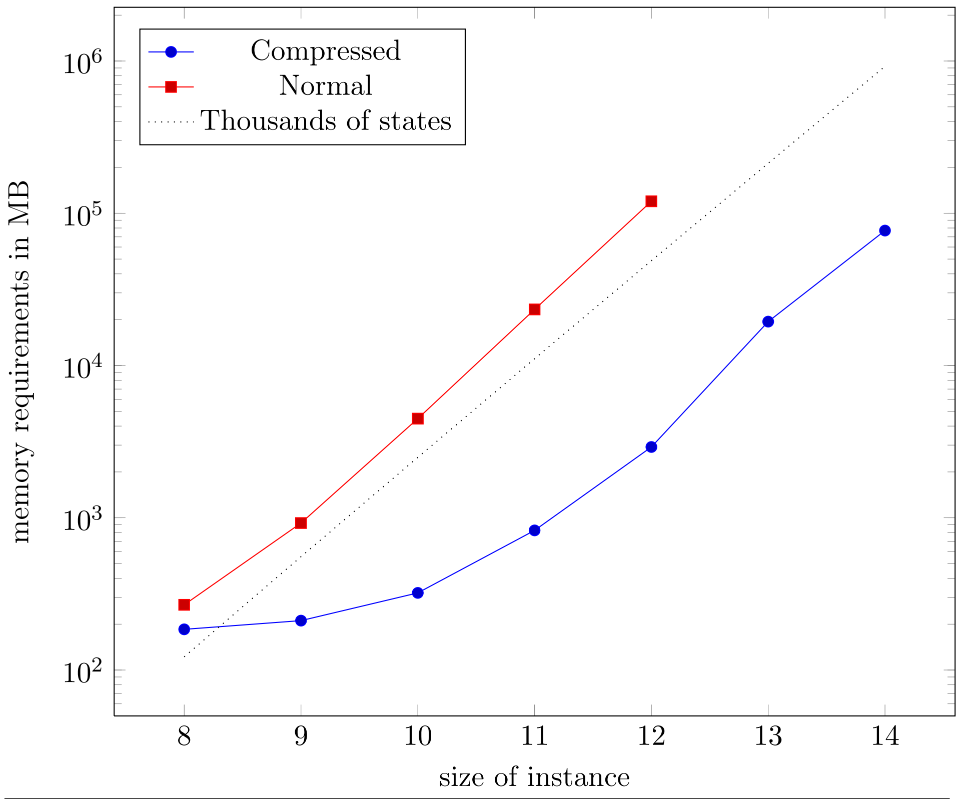
Beem database a large database of DVE models of different sizes [13], some of the bigger models were used,

Modifications of above in some cases, models could be modified to create bigger instances

5.2 Timed automata

Timed automata in the UppAal format are supported by DiVinE as described in [7]. Jan Havlíček also kindly provided an implementation of `splitHint` for timed automata, providing optimized partitioning of states.

Figure 5.1 shows impact of tree compression on the model of Fischer's mutual exclusion protocol relative to a number of processes. Measurements were performed on aura.fi.muni.cz with 440GB of memory, the limit was set to 250GB.



size of instance		8	9	10	11	12	13	14
memory [MB]	normal	268	923	4478	23352	119842	–	–
	compressed	185	211	321	826	2916	19420	76960
millions of states		0.12	0.55	2.5	11.1	48.8	212.9	922.2
compression ratio		69%	23%	7.2%	3.5%	2.4%	–	–

Figure 5.1: Memory requirements for verification of Fischer's mutual exclusion protocol. relative to number of processes.

Figure 5.2 shows measurements of reachability (in this case property is deadlock freedom) on computer with 8GB of RAM, memory limit was set to 7GB. For all finished examples with the exception of boxes.xml and fixer.xml, the property holds.

It can be seen that the impact of tree compression rises with state-space sizes. As timed automata have large states, the impact is significant for instances of roughly half a milion states and more; these instances can be checked with minimal time overhead.

	memory [MB]			time [s]			# of
	normal	ntree	ratio	normal	ntree	ratio	states
boxes	160	176	110.0%	0	0	113.3%	8.9K
bridge	156	172	110.3%	0	0	74.1%	206.0
fischer	156	172	110.3%	0	0	107.3%	5.8K
fischer8	296	188	63.4%	6	6	108.3%	122.2K
fischer9	954	228	23.9%	38	38	101.4%	555.1K
fischer10	4517	410	9.1%	201	212	105.7%	2.5M
fischer11	–	1174	–	–	1262	–	11.1M
fischer12	–	4163	–	–	7810	–	48.8M
fischer13	–	–	–	–	–	–	–
fixer	164	176	107.3%	1	2	135.4%	205.7K
train-gate8	445	256	57.6%	24	27	111.6%	726.9K
train-gate9	3420	1249	36.5%	218	289	132.7%	6.5M
train-gate10	–	–	–	–	–	–	–

Figure 5.2: Timed automata – compression with memory limit of 7GB.

Figure 5.3 shows measurements of the MAP algorithm for LTL verification. The MAP algorithm was used instead of OWCTY (which is asymptotically faster) because it currently better integrates with tree compression. With OWCTY, IPC queues are used even in a single-threaded case, in manner which can push the whole spate space into them (as *to* vertices, therefore uncompressed). Obviously, this cancels out any savings from tree compression. OWCTY running with a shared visitor does not have this drawback.

	memory [MB]			time [s]			# of
	normal	ntree	ratio	normal	ntree	ratio	states
bridge	129	164	126.9%	0	0	130.2%	1.6K
fischer9	2575	344	13.4%	1001	1693	169.1%	1.7M
fischer10	3599	345	9.6%	206	349	169.8%	31K–67K
fischer11	–	1044	–	–	2298	–	117.4K
fischer12	–	4005	–	–	12921	–	195.9K
fischer13	–	–	–	–	–	–	–
fixer	3604	3150	87.4%	446	644	144.6%	214K–99K

Figure 5.3: Timed automata – LTL verification with limit of 7GB, using MAP algorithm.

For the models bridge and fisher9, the property holds; for remaining models, most of the memory requirements is due to counterexample generation. The reason state-space sizes are different when compression is running can be tracked to the implementation of the MAP algorithm – it internally orders states of the product automaton by their location in memory, which is different if states are compressed.

Note that in DiVinE running LTL verification, the same system state is generated repeatedly with different state of property automaton, therefore compression ratio is better when verifying LTL properties, as overhead of multiple states is reduced to overhead of multiple instances of a tree root and the part of the state which contains the state of the property automaton.

5.3 LLVM – C and C++ programs with threads

The LLVM interpreter is used by DiVinE for verification of parallel C and C++ programs which are translated to LLVM bitcode by CLang compiler [14, 2]. Thanks to reductions described in [14], most LLVM models in the DiVinE distribution are too small to be considered as candidates for compression; nevertheless, all instances were tested and the table is divided into two parts, first shows models that cannot be verified in less than 1GB without compression, the remaining (smaller) models are in the second part.

	memory [MB]			time [s]			# of
	normal	ntree	ratio	normal	ntree	ratio	states
airlines	–	–	–	–	–	–	–
elevator	5191	772	14.9%	4238	4417	104.2%	825.2K
elevator2	75963	1792	2.4%	91575	91430	99.8%	10.5M
pt_barrier	9247	1126	12.2%	4583	5052	110.2%	1.4M
pt_rwlock	25191	1845	7.3%	19604	22811	116.4%	3.9M
anderson	218	269	123.7%	0	1	151.0%	247.0
at	414	285	68.8%	115	174	151.0%	42.0K
bakery	254	305	120.3%	2	3	133.7%	1.3K
bridge	160	205	127.8%	0	0	247.8%	226.0
collision	600	596	99.2%	97	196	203.2%	10.4K
cyclic_sched	607	427	70.2%	202	255	126.2%	48.1K
elevator_plan	170	207	121.9%	1	1	109.4%	4.8K
fifo	255	306	120.1%	2	2	111.6%	162.0
fischer	334	289	86.6%	74	121	163.0%	24.2K
global	193	243	125.6%	0	0	136.7%	44.0
lamport	824	315	38.2%	334	362	108.2%	131.1K
lamport_n1	247	299	120.8%	4	5	116.5%	2.2K
lamport_n2	215	267	123.9%	0	0	109.7%	164.0
lead-uni_b	481	433	90.0%	217	250	115.4%	10.1K
lead-uni_dkr	294	345	117.5%	2	2	116.5%	304.0
lead-uni_pt	458	452	98.6%	84	91	108.9%	6.5K
peterson	191	241	125.7%	1	1	119.7%	631.0
pt-showcase	465	517	111.0%	2	3	120.0%	1.2K
pt_cond_vars	361	412	114.2%	10	12	118.8%	5.5K
pt_mutex	203	255	125.3%	0	0	129.7%	67.0
ring	187	239	127.5%	3	3	115.9%	1.4K
szymanski	280	331	118.4%	3	4	116.2%	1.6K

Figure 5.4: LLVM compression.

Experiments showed that for LLVM, even models with tens of thousands of states can benefit from tree compression and with millions of states, the benefit is massive. Moreover, the time penalty is small. Please note that a state-space of an LLVM model is already massively reduced, as described in [14].

No particular limits were set for those tests, but the airlines model failed to terminate within a day, hence it was omitted (the tests were performed on an 8 socket server aura.fi.muni.cz with 440GB of RAM).

5.4 DVE

DVE models are models of parallel communicating finite automata; this formalism was designed by DiVinE authors. Abundance of DVE models can be found in Beem database. As Beem contains hundreds of models, most of them small, only a subset was used: first, large instances of models were chosen (instances with more than 5 workers if at least two of them were available, otherwise two biggest), then models were verified with memory limit of 7GB and without partial order reduction (`--no-reduce` option was passed to DiVinE). Models requiring at least 1GB of memory (in uncompressed verification) were finally benchmarked. Results can be seen in Figure 5.5.

It can be seen that for DVE, tree compression does not achieve very good results. Overall, compression gives moderate improvement in memory consumption but incurs significant time overhead. This is caused by fact that DVE models have very small states. Moreover, the DVE generator is far faster than timed automata and LLVM generators. If we analyse the worst achieved result (for at.5) we can see that average memory requirement for one state is roughly 93 bytes in uncompressed verification (computed as ratio of overall memory and number of states). Of course this is an upper bound, including overhead of all data structures used by DiVinE (such as hashtable itself, queues and many other), the real size of state is 46 bytes in this case. Obviously, tree compression with default leaf size of 32 bytes cannot help much in this case and other methods such as Huffman compression would presumably give better results.

	memory [MB]			time [s]			# of
	normal	ntree	ratio	normal	ntree	ratio	states
anderson.5	5590	6847	122.5%	156	433	277.1%	51.5M
anderson.6	1484	1510	101.7%	66	197	299.5%	18.2M
anderson.7	–	–	–	–	–	–	–
at.5	2841	4285	150.8%	153	312	204.4%	32.0M
at.6	–	–	–	–	–	–	–
bakery.8	1706	1377	80.7%	86	199	232.2%	19.7M
exit.4	1506	1493	99.2%	67	139	207.5%	15.3M
firewire_tree.5	3151	834	26.5%	195	452	231.7%	3.8M
hanoi.3	1963	2920	148.7%	52	156	299.7%	14.3M
hanoi.4	–	–	–	–	–	–	–
iprotocol.5	3403	2489	73.1%	89	266	297.9%	31.1M
iprotocol.6	4008	2483	61.9%	139	339	244.1%	41.4M
iprotocol.7	6680	4810	72.0%	244	499	204.7%	59.8M
lamport.7	2939	3450	117.4%	122	367	299.4%	38.7M
leader_elect.5	1444	655	45.4%	44	173	391.7%	4.8M
leader_elect.6	–	5863	–	–	1661	–	35.8M
leader_filt.6	1588	1359	85.5%	44	133	304.8%	16.9M
mcs.5	5535	5262	95.1%	198	614	310.3%	60.6M
phils.6	1054	935	88.7%	34	145	429.0%	6.6M
phils.7	–	–	–	–	–	–	–
phils.8	2457	1702	69.3%	127	430	339.3%	19.4M
plc.4	1092	825	75.6%	23503	23647	100.6%	3.8M
prod_cell.6	1970	1428	72.5%	50	163	328.0%	14.5M
sokoban.3	–	4831	–	–	1215	–	72.4M
sorter.4	1674	1381	82.5%	182	227	125.0%	12.6M
szymanski.5	6449	4853	75.3%	334	1110	332.4%	79.5M
telephony.6	–	–	–	–	–	–	–
telephony.7	1658	1858	112.1%	137	301	220.3%	22.0M
telephony.8	–	–	–	–	–	–	–

Figure 5.5: DVE – compression with memory limit of 7GB.

Chapter 6

Conclusion

We proposed an improved version of tree compression, a method to mitigate state-space explosion in an explicit-state model checking. It works by replacing a standard hashtable with a compressed hashtable, which reuses parts of states that are already saved.

The aim was to provide a memory efficient storage for model checking of programs with large state-spaces, such as real-world programs in C and C++. Those state-spaces are big in both number of states as well as in size of a particular state, which may contain, for example, dynamically allocated data structures. Also, such a program has variably-sized states and we support this in our tree compression.

The proposed tree compression is generic, not bound to a particular modeling language or state-space generator. In fact, it could be used in different environments than model checking, as a general hashmap. At the same time, our implementation of tree compression can be easily integrated with specific state-space generators to further improve memory efficiency, specifying an optimal shape of the tree representation.

Tree compression was implemented in the DiVinE model-checker and integrated with most of its functionality (currently the only exception being distributed MPI verification). Tree compression is being used for compression of BFS queues as well, and for partial compression of a DFS stack.

Results of tree compression on LLVM (which is used for verification of C and C++) programs and timed automata are very good memory saving (best achieved compression ratio for LLVM examples is 2.4%), with minimal impact on verification time. As a result of this efficiency, feasibility of some LLVM examples shifted (memory-wise) from multi-socket servers or networks of workstations to a single desktop or laptop computer. In turn, this is a starting point for verification of instances for which the standard approach would require several terabytes of memory. Also the additional memory overhead of LTL verification can be efficiently reduced by tree compression.

Integration with parallel verification is supported, although it loses some efficiency using a partitioned setup traditional in DiVinE. This downside can be mitigated by using of new shared memory hashtables and queues [16].

6.1 Future work

In future, we would like to combine tree compression with distributed verification using MPI, allowing states to be compressed on each workstation to facilitate verification of even larger state-spaces.

Memory efficiency of tree compression could be further improved by saving some parts of a state explicitly in the root of a tree. Such parts would include short and heavily changing fragments, such as positions in property automaton when verifying LTL properties, or program counters in LLVM.

As the speed of verification with tree compression may become limiting factor in some cases, we would like to work on methods to improve time efficiency, while retaining its memory efficiency.

Finally, combination of tree compression with other compression techniques such as Huffman compression is an interesting field which may further push limits of the state-of-the art explicit-state LTL model checking.

Bibliography

[1] Jiří Barnat, Luboš Brim, and Petr Ročkai. “Parallel Partial Order Reduction with Topological Sort Proviso”. In: *Software Engineering and Formal Methods (SEFM 2010)*. IEEE Computer Society Press, 2010, pp. 222–231.

[2] Jiří Barnat, Luboš Brim, and Petr Ročkai. “Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs”. In: *NASA Formal Methods Symposium*. Vol. 7226. LNCS. Springer, 2012, pp. 252–267.

[3] Jiří Barnat, Jan Havlíček, and Petr Ročkai. “Distributed LTL Model Checking with Hash Compaction”. In: *To appear in proceedings of PASM/PDMC 2012*. 2013.

[4] Jiří Barnat et al. “DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs”. In: *To appear in Computer Aided Verification (CAV 2013)*. 2013, p. 6.

[5] Jiří Barnat et al. *DIVINE: Model Checking for Everyone*. 2013. URL: http://divine.fi.muni.cz/ (visited on 05/12/2013).

[6] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-03270-8.

[7] Jan Havlíček. “Untimed LTL Model Checking of Timed Automata [online]”. Master’s thesis. Masaryk University, Faculty of Informatics, 2013. URL: http://is.muni.cz/th/324943/fi_m/ (visited on 05/06/2013).

[8] Gerard J. Holzmann. “State Compression in SPIN: Recursive Indexing And Compression Training Runs”. In: *Proceedings of third international* SPIN *workshop*. 1997.

[9] Gerard J. Holzmann. “The model checker SPIN”. In: *IEEE Transactions on Software Engineering* 23 (1997), pp. 279–295.

[10] Gerard J. Holzmann and Anuj Puri. “A Minimized Automaton Representation of Reachable States”. In: *Software Tools for Technology Transfer* 2 (1999), pp. 270–278.

[11] A. W. Laarman, J. C. van de Pol, and M. Weber. “Parallel Recursive State Compression for Free”. In: *Proceedings of the 18th International SPIN Workshop, SPIN 2011, Snow Bird, Utah*. Ed. by A. Groce and M. Musuvathi. Vol. 6823. Lecture Notes in Computer Science. Snow Bird, Utah: Springer Verlag, July 2011, pp. 38–56.

[12] Alfons Laarman. *LTSmin*. 2013. URL: `http://fmt.cs.utwente.nl/tools/ltsmin/` (visited on 05/12/2013).

[13] Radek Pelánek. “BEEM: Benchmarks for explicit model checkers”. In: *In Proc. of SPIN Workshop, volume 4595 of LNCS*. Springer, 2007, pp. 263–267.

[14] Petr Ročkai, Jiří Barnat, and Luboš Brim. “Improved State Space Reductions for LTL Model Checking of C & C++ Programs”. In: *NASA Formal Methods (NFM 2013)*. Vol. 7871. LNCS. Springer, 2013, pp. 1–15.

[15] Jaroslav Šeděnka. *Huffmanovo kódování stavů v DiVinE [online]*. Bachelor’s thesis. 2007. URL: `http://is.muni.cz/th/143135/fi_b/` (visited on 04/29/2013).

[16] Jiří Weiser. *Dynamicky rostoucí sdílená hašovací tabulka pro DiVinE [online]*. Bachelor’s thesis. 2013. URL: `http://is.muni.cz/th/374154/fi_b/` (visited on 05/06/2013).