Faculty of Informatics,
Masaryk University

State space compression for the DiVinE model checker

Bachelor's thesis

Vladimír Štill

Brno, Spring 2013

Declaration

Thereby I declare that this thesis is my original work, which I have created on my own. All sources and literature used in writing the thesis, as well as any quoted material, are properly cited, including full reference to its source.

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

Abstract

The main focus of this thesis is reducing memory requirements of explicit LTL model checking by use of tree compression.

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 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 model checking[1] is a well established technique used for verification of concurrent asynchronous processes. As multithreaded environments are common in those days the need to test and verify concurrent processes is increasing. Common testing methods such as unit testing are failing in this case as asynchronous concurrency comes with nondetermin-

[1] Introduction to explicit model checking and automata based approach to LTL verification can be found in [6]. Also section 1.1 gives information about model checking that is necessary for this thesis.

ism (which is caused by independently scheduled processes). Therefore their testing is inefficient, for example unit tests may terminate successfully even though failing run exists (and it may be less probable than succeeding runs). Explicit model checking has ability to solve this issue – it can verify that there does not exist run violating given property (such as assertion violation or LTL property).

However model checking comes with problem called state-space explosion, cased by need to verify all runs of system. Despite permanently growing sizes of random access memory available for contemporary 64-bit computers, state-space explosion is still a major limitation to explicit model checking.

This limitation may become even more severe when verifying unmodified programs (in languages such as C and C++), as in case described by [2]. Although sophisticated reduction techniques (such as those in [14]) can be applied to reduce state-state explosion caused by interleaving in multithreaded programs, state-space of real-world programs is still generally large, for example due to dynamic memory structures and recursion stack or due to usage 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 model checking, for example Huffman compression, COLLAPSE [8], and modeling language independent tree compression [11].

The aim of this work is to introduce version of tree compression capable of substantially reducing memory requirements of large state-spaces. This technique is general, in can be applied in modeling-language-independent setting with good memory saving. At the same time it can be optimized for specific need of modeling language to provide even better results. Proposed tree compression technique can also be applied to parallel LTL model checking.

In this work we will describe tree compression and provide implementation of it in terms of parallel LTL model checker DIVINE, which is being developed in ParaDiSe laboratory of Faculty of Informatics Masaryk University. Finally this work also includes experimental results of our tree compression.

1.1 Model checking

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

The system can be viewed as computer program in some programming language, traditionally there exist languages designed specially for model checking, but model checking can be performed even on programs in some general-purpose language such as C and C++.

Property is a formula in temporal logic. Commonly used logics to describe properties are LTL, CTL and CTL*. Typical properties used in model checking are for example assertion safety, deadlock freedom, response properties and many other.

1.2 Explicit model checking

Explicit model checking uses the fact that system in given moment of execution can be fully described by the memory it is using – its state. Therefore run of

system can be viewed as sequence of such states. As verified systems are usually non-deterministic – either as a result of asynchronous parallelism or as a feature of programming language – all possible runs of such system can be naturally expressed by graph of all reachable states, connected by oriented transitions whenever one state can change to another without any intermediate state which can be observed as different by program. This graph is called state-space graph of system.

Explicit model-checker builds this graph and searches it for runs violating property. Similarly to many other graph traversing algorithms, set of already visited states (so called closed-set) need to be used. As state-space graph can be vast, storing this set gives the problem of state space explosion – that is memory requirements of model checking are much larger than memory requirements of single run of system.

Chapter 2

Existing state-space reduction methods

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

2.1 Partial order reduction

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

Since reductions achieved by partial order reduction are orthogonal to reductions of tree compression it can be combined to provide even better state-space reduction.

DiVinE has implementation of parallel partial order reduction [1].

2.2 Compact state-space representation

In this section I want to present some techniques used to decrease memory consumption incurred by storing state-space in explicit model-checkers without reducing number of processed states. Most of this techniques are trying to represent closed-set compactly, sometimes the same representation can also be used

for open-set.

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

2.2.1 Automata representation

As presented in [10], minimized deterministic automata can be used to represent whole state-space of explicit model-checker. This approach views state-space as set of words over alphabet Σ, that is $S \subseteq \Sigma^*$.

The approach presented in aforementioned paper expects fixed state sizes, therefore can not be directly applied to DiVinE which does not have fixed length of states (in LLVM verification). For this reason, another method of encoding would be needed.

Furthermore algorithms implemented in DiVinE require closed-set to behave as associative map, which is not easy to represent by automata.

2.2.2 Huffman compression

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

This method was already implemented in DiVinE in past [15], but it is not supported in DiVinE 3.0. It was also implemented in 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 advantage with better speed and easier integration in our environment.

2.2.3 Collapse and recursive indexing

In [8] several compression methods are described and evaluated with respect to Spin model-checker. 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 hash-table. State itself is then represented as global data plus indices of separately-stored components. As stated in aforementioned paper a shortcoming of this method is that an upper-bound on the largest index for the state components must be known. Also implementing this method with growing hash-table would pose further challenges.

Improved version of the COLLAPSE method is also discussed and evaluated in the aforementioned work, this method is also called recursive indexing. This improved version is allowing index sizes to vary between states by saving index sizes in global component of 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 here.

Please note that COLLAPSE is using knowledge of state layout to achieve its results, therefore it requires interface between state-space generator and storage. This may be complication in cases where this interface is not present, for example when using external generating algorithm as in DIVINE CESMI.

2.2.4 Tree compression

In [11] recursive state compression or tree compression is presented together with evaluation in LTSmin [12]. This technique is basically modification of COLLAPSE method with states being partitioned recursively. In their implementation state is partitioned into slots of fixed size. Tuples of slots are then stored in fixed size hash-table, forming leaves of 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 state layout and with growing hash-tables used in DIVINE.

2.2.5 Hash compaction

Hash compaction, presented for example in [3] is method which trades completeness of model checking for memory. Instead of storing full states in hash-set it stores only hashes of visited states and, if necessary, associated information. If implemented carefully it can lead to algorithm which does not give false-negative answers, that is if algorithm finds counterexample than

this counterexample witnesses violation of verified property. However algorithm can miss some counterexample as some states (with equal hash) can be merged during verification.

Since hash compaction tackles state-space explosion by omitting state information it can be viewed as lossy compression and as such it cannot be used together with tree compression. Please note that since successors are generated from 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 exploits specific nature of 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 are usually reducing number of states and therefore are orthogonal to tree compression and can be well combined.

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 first version of DiVinE and is still supported in DiVinE 3.0, despite the fact that availability of 64 bit multicore processors and therefore computers with large RAM and vast computing power in one machine allowed verification of bigger systems without this extension.

As distributed memory access is much slower than local (shared) memory access it is necessary to apply tree compression only inside one workstation. Communication between workstations then occurs with uncompressed states. Although tree compression will get less efficient in 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 explicit model-checker designed to utilize parallelism of in both shared memory and distributed memory setting. It supports both safety and LTL properties and currently supports several input formats such as DVE, LLVM bytecode (which can be automatically created from C or C++ source by DiVinE using CLang compiler), UppAal timed automata format, and CoIn. It also provides Cesmi interface which allows representing models as shared library which is loaded by DiVinE at runtime and used to generate state-space – this allows easy integration of

new input formalisms, possibly by external developers.

Source codes of DiVinE are freely available from DiVinE repository[1] and from web page of DiVinE [5], under BSD licence.

3.1 Architecture

The DiVinE is written in C++ and since version 3.0 it utilises C++11 language features. Architecture is modular, with modules connected together using C++ templates as opozit to quite common approach using inheritance and virtual calls. This design allows tighter integration of components that happens during compile time, allowing more code to be inlined which results in faster operation.

The DiVinE consists of several algorithms, each represented as separate module, itself using other modules such as visitor modules and store modules. Visitor modules are implementing several graph traversal algorithms such as DFS, BFS, and pseudo BFS. Pseudo BFS is used for parallel verification algorithms, it does

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

not guarantee particular order of traversal and is not deterministic. Store modules are used by visitor modules and algorithms to represent closed set in graph traversal. Each store is hash-table-like module, supporting insertion and retrieval of states and several supportive operations.

Since implementation of tree compression required some changes in the architecture of DiVinE, several technical aspects of DiVinE 3.0 will now be presented. Those aspects are necessary for understanding of 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 DiVinE 3.0 repository[2].

3.1.1 Generators

As input formats for DiVinE are usually programs in some formalism, it is necessary to generate state-space for explicit model checking from them. This generation is performed on-the-fly from initial state by graph

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

generator. With exception of CESMI generator[3] all generators are input format specific. They are providing common interface, notably functions `successors` and `isAccepting` which are responsible for generating successors of given state and deciding whether state is accepting respectively. Currently all input formats are implicit, that is state-space has to be generated by those functions, it is not stored as part of input.

Each state (that is vertex of state-space graph) is represented by object of type `Blob`, which is flat piece of memory that can be read and written on given location, and size can be detected. State memory is separated in two parts (where the first is optional): slack and state of model. While slack is used only by algorithms and never touched by generator, the opposite holds for state of model. Also only state of model is hashed when storing state. State of model never changes once the state is generated.

[3] CESMI generator is used as interface to external generator provided by shared library.

3.1.2 Parallelization of algorithms

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

Partitioned visitor is working with static partitioning of states to threads using hash of state. Most operations are then performed in thread local fashion, for example each thread has its own hash-table. In this setting threads are communicating using IPC queues of edges (queues are transferring *from* state and *to* state of edge). This setting can be extended to multiple machines using MPI. The disadvantage of this approach is that static partitioning may cause different loads for different threads.

Shared visitor is new experimental feature of DiVinE which is under heavy development [16]. It facilitates shared queue and shared hash-table to allow faster parallel pseudo BFS exploration. Currently it cannot be combined with MPI.

3.1.3 Interface between visitor and algorithm

Since all algorithms implemented in DiVinE share common basic way of graph traversal, that is they are, for particular states, looking at outgoing edges and then (if necessary) to states on the other side of edge, this usage is abstracted in interface between visitor and algorithm. This interface works as follows: graph is being traversed in order specified by used visitor (DFS, BFS or pseudo BFS order) and each edge (leading from already processed state denoted by *from* to its target state denoted by *to*) is processed:

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

necessary (used by hash compaction).

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

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

3.1.4 Memory management

Memory management in DiVinE 3.0 is quite straightforward, each state of graph is either temporary or permanent and its state is marked by single bit inside `Blob`. State is generated as temporary and it becomes permanent once stored in store. Permanent states are never deallocated during run of DiVinE and their location in memory never changes.

This memory management is simple, imposes virtually no overhead and was 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 as single run of DiVinE can verify only one property of one model.

3.1.5 Stores

DiVinE 3.0 supports three types of store: partitioned store, shared store and hash-compacted store. The partitioned store is original version used by partitioned visitor. It has one hash-table for each thread and storing is full states. Shared store is experimental version used by shared visitor. It is optimized for shared memory, with one hash-table which is shared across threads. Hash-compacted store is derived from partitioned store and is implementing hash compaction [3], the method of lowering space requirements of explicit model checking by storing only hashes of states in closed set. 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 slack of state by algorithms. When algorithm finishes and detects property violation it starts counterexample generation, passing either accepting state (in case of reachability analysis)

or state on accepting cycle (in case of LTL verification) to counterexample generating algorithm.

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

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

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

Chapter 4

Tree compression

With traditional hash-table approach to explicit model checking, full states are saved. However this is not necessary as in most cases only small part of state changes from one state to its successors. Several methods to take advantage of this fact were introduces, for example COLLAPSE in SPIN [8] and tree compression with binary tree in LTSmin [11].

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 vast state-spaces.

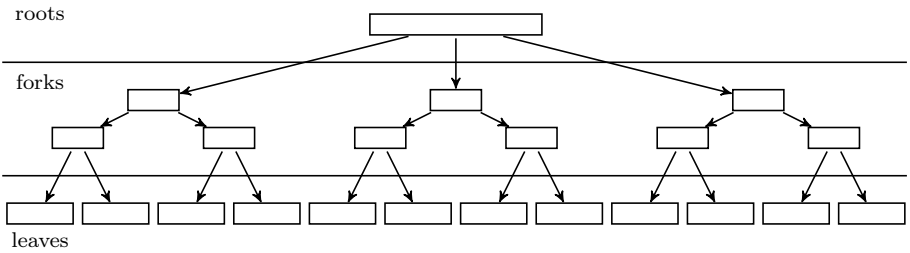


Figure 4.1: Tree representation of state.

4.1 Representation of states

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

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

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

Figure 4.2 is illustrating 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 misleading name as states are represented as acyclic oriented graphs).

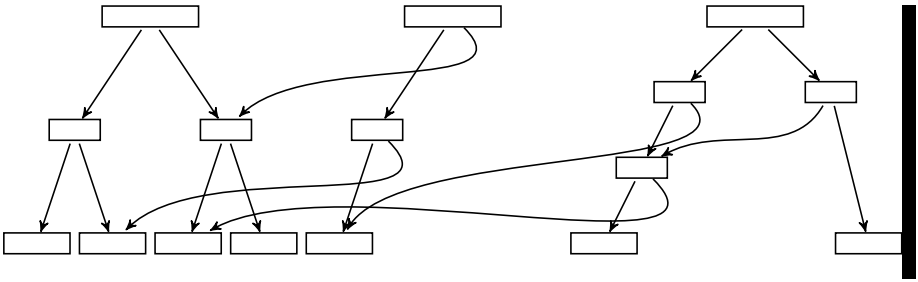


Figure 4.2: Subtree sharing in tree compression.

4.2 Requirements

In DiVinE it was required that tree compression will integrate well with most already supported features such as parallel verification, counterexample generation or partial order reduction. Also it is supposed to work without special setting by user (with exception of enabling it). Therefore it is important that tree compression must allow closed set to be growing, as DiVinE supports growing hash-sets in parallel verification as opposed to some model-checkers such as Spin who are requiring user to provide upper bound of number of states.

When compared to 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 bounds), whereas above mentioned solution partitioned states to fixed size chunks.

It must also be possible to integrate tree compression with verification using shared growing hash-table which is developed in the same time as tree compression (more about shared growing hash-table in DiVinE can be found in [16]).

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

4.3 Design and implementation

Because of aforementioned requirements tree compression was written from scratch, not using any implementation of similar technique.

The compression support itself consists of two main parts, the tree-compressed hash-set and tree-compressed store. Tree-compressed hash-set is itself based on arbitrary hash-set which is providing `insertHinted` and `getHinted` operations. This design choose allowed me both to fully reuse current implementation of hash-set used for partitioned verification as well as easy support for experimental version of shared hash-set.

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

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

[2] DiVinE 3.1 Darcs repository with tree compression implemented as of this thesis is available on

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

4.3.1 Tree-compressed hash-set

Tree-compressed hash-set, represented by class `NTreeHashSet`[4] is the core of tree compression. It implements com-

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]`divine/toolkit/ntreehashset.h`

pression itself exporting interface similar to hash-sets already present in DiVinE.

The header of `NTreeHashSet` declaration looks as follows:

```
1 template< template< typename, typename >
2                   class HashSet,
3             typename Item, typename Hasher >
4 struct NTreeHashSet;
```

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

Tree-compressed hash-set utilizes three hash-sets (of base type given by its `HashSet` parameter). Those hash-sets are used to store different parts of tree – roots of tree including slack are stored in *roots* set, internal nodes are stored in *forks* set and finally leaves of tree are in *leaves* set. This setting allows maximal reusal of internal nodes and leaves already created by previ-

ous inserts, while at the same time gives tighter type control (those tables are storing pointer to different types as each of type of tree nodes requires different metainfomation).

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

If by chance the model state is shorter than given lower limit for compression (given by generator), it is stored entirely in roots table – in this situation tree compression is actually incurring space overhead as it has to store additional information in root of tree as well as the state itself in uncompressed way. In practice this situation will usually imply that model itself is quite small and can be easily verified without tree compression and the overhead of tree compression enabled will not matter (also those models are not primary interest of tree compression as mentioned above).

In all other cases model state is partitioned and

stored using all three hash-tables. States are inserted using function `insertHinted` with following header:

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

This function requires state, its hash, and generator responsible for state and it is returning tuple of root and boolean flag denoting whether state was freshly inserted or already present. The generator (subsection 3.1.1) is used to provide hints on desired partitioning of state. The interface between generator and `NTreeHashSet` is provided by following functions implemented by generator:

```
1 template< typename Yield >
2 void splitHint( Node n, Yield yield );
3 template< typename Yield >
4 void splitHint( Node n, int form, int length,
5                 Yield yield );
```

The first of those function overloads is basically shortcut for top-level splitting of entire model state, therefore we can focus only on the second one. The `splitHint` function is using generator pattern, expecting its `yield`

parameter to be callable object[5]. A range from `n` together with flag indicating whether this range is to be stored as leaf and number of remaining partitions is passed to `yield`. All other arguments of `splitHint` are pretty self-explanatory. Default implementation of `splitHint` is provided in `divine/graph/graph.h`, this implementation is creating balanced binary tree and is generator-agnostic. Specialized versions can be implemented in particular generator and will be used automatically if available.

Inside `insertHinted`, `splitHint` is called and lambda function is passed as yield argument, allowing easy recursive generation of n-ary tree defined by generator. Advantage of this approach is that tree is created on-the-fly, without any supportive data structure (the overhead of recursive calls is presumably small as C++ compilers are performing massive inlining and depth of tree should be at most logarithmic to size of state in good implementation of `splitHint`). Using recursion, `splitHint` is called until it reaches bottom of desired

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

tree. In this moment `Leaf` instance is allocated and given range is copied to it from state. Leaf is then inserted into leaf set, which returns equivalent permanent leaf. This permanent leaf is used on upper level of recursion to construct fork (pointers to permanent leaves or forks are saved in forks), and forks are saved into forks table in similar fashion. As recursion backtracks to root entire tree is created and finally root is inserted. Each time element is inserted to one of the tables, the original element is discarded if it was already present in the table (if element is not present prior insertion it is equivalent to permanent element and therefore cannot be freed). Finally permanent root element is returned together with flag indicating whether it was already present (state is present in `NTreeHashSet` if and only if corresponding root is present here).

As an optimization roots are saved using hash of entire model state, not the binary representation of `Root` object. This allows fetching roots from `NTreeHashSet` by state, without repeating aforementioned construction of tree which is allocation heavy (state can be compared to tree representation almost without allocation – only stack for tree traversal has to be allocated

– but tree is constructed from state with allocation at least as big as state itself).

As already mentioned vertices of tree are connected with 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 hash-tables as well as reconstruction of state from tree without assistance of hash-table. Using indices to hash-table would require trees to be recreated each time any of tables is resized, slowing down resizing and complicating design (tree compression would had to be tightly integrate with hash-table, whereas in this approach any table with given interface can be used). Furthermore leveraging advantage of shorter indices compared to pointers would require trees to be parametrized by size of index. Finally (as mentioned in section 4.2) we are aiming at vast state-spaces, possibly with billions of states, this would either require indices to be stored in variables whose size is not 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 is using some assumptions that no longer holds for compressed states. In particular it is assumed that when *from* state is processed by algorithm in `transition` (item 3 in subsection 3.1.3) it is permanent and its memory location can be saved inside *to* state of transition for further use in counterexample generation. This assumption no longer holds for compressed states as state is no longer saved in one unchanged piece but is still processed this way by algorithms and visitor (therefore state visible in algorithms is temporary). Note that this is the problem which caused hash compaction to be unable to provide counterexamples in DiVinE 3.0.

To solve those issues and allow easy integration of new types of compression two new types wrapping state were introduced. Those types are dependent upon type of store and generator and they are implemented as nested types of store. They are having common interface independent of store type.

`Vertex` represents both (possibly temporary) full state

suitable for successor generation and permanent state saved in hash-table. It is passed to algorithms when processing transitions and states of model. All uses of slack must be accessing stack in permanent part of state. Objects of type `Vertex` must be convertible to `VertexId`.

`VertexId` represents permanent state. It contains slack, and provides access to it. `VertexId`s are processed by algorithms when iterating over whole hash-table (used by OWCTY and MAP). Objects of type `VertexId` are generally not suitable for successor generation but support for their conversion to `Vertex` may be provided (if compression is lossless). Objects of this type must be unique and therefore comparable by identity.

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

For tree compression, `VertexId` is wrapper around pointer to root of tree (which is stored in roots table inside `NTreeHashSet` and it provides conversion to `Vertex` as tree compression is lossless). `Vertex` is

then containing both uncompressed state as provided by generator and pointer to root of tree.

For hash compaction, `VertexId` is object containing just slack and hash of node, while `Vertex` adds full state to it. As hash compaction is lossy compression, conversion from `VertexId` to `Vertex` cannot be provided for hash compaction versions.

Algorithms were modified so that they are always accessing slack via version saved in hash-table and, when they need to save identifier of other state (for example for predecessor tracking), they must save `VertexId`. This incurs no memory overhead as `VertexId` is containing single pointer – same as `Blob` object used to represent state in all current generators.

4.3.3 Compressed queues

Compression of closed-set as mentioned above would certainly help to reduce memory requirements. But left alone there would still be much space wasted.

This is caused by queues used as open-set for most algorithms. Those queues can get quite long as they can be upper bounded by number of states, or more precisely by width of state-space graph. Queues were

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

Compression is activated by store, which defines type `QueueVertex`, which is 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 alias of `VertexId` as it can be decompressed. For hash compaction it must be defined as alias of `Vertex` because hash compaction is lossy compression. Decompression is performed on-the-fy in queue.

Prior to this optimization queues were (in presence of tree compression) containing full states which were not saved in store leading to large memory overhead. Now queues are storing just `VertexId`s, each of size of single pointer.

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

gorithm used in DiVinE.

In presence of distributed verification using MPI it would be necessary to decompress `QueueVertex` when sending it to other machine as compressed representation is just a pointer and therefore would be invalid when sent to different machine.

4.3.4 Interface between visitor and algorithm

Slight modification of interface between visitor and algorithm was required as algorithms must have access to `VertexId` for both *from* and *to* states of transition. The new workflow looks like this:

1. edge is processed by function `transitionFilter` provided by visitor,
2. if transition is marked to be ignored its processing is abandoned,
3. *to* state is inserted to store, store returns `Vertex` associated with this state and boolean flag denoting if vertex was already present,
4. edge is processed by function `transition` provided by algorithm (passing inside `Vertex` object

for both *from* and *to* states),

5. if transition is to be followed, `Vertex` associated with *to* state is passed to `expansion` function provided by algorithm.

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

4.3.5 Counterexample generation

As mentioned in subsection 3.1.6 counterexample generation algorithm in DiVinE 3.0 used some assumptions, which no longer holds for compressed state-space.

The new algorithm created for DiVinE 3.1 takes advantage of `VertexId`s as guaranteed permanent identifiers of state which are saved in place of parent pointers.

The algorithm has to cope with fact that counterexample is generated from sequence of full states, whereas `VertexId`s are just identifiers of those states. It would be possible to use only slightly modified version of original algorithm which would first reassemble

`VertexId` to `Vertex` and than continue with full state. However this approach would not be suitable for hash compaction.

The new algorithm is generating each trace in two phases. First it traces `VertexId`s back to initial state (they can be traced as parent's `VertexId` is saved inside slack which is accessible from `VertexId`). Comparison to initial state must be performed on thread responsible for given state, that is the thread which has it saved in its hash-table. Output of this first phase is sequence of `VertexId`s from initial state to target state. In second phase counterexample consisting of full states is generated using this sequence. This can be done by starting in the initial state (which can be always generated again by generator) and following path marked by sequence of `VertexId`s. Each successor in trace is compared to matching `VertexId` and successors leading outside trace are ignored. Please note that comparison to `VertexId` must again happen on thread owning given state.

Cycle traces can be generated in similar fashion, main difference is that accepting state is used instead of initial vertex of graph (this is again preceded by parallel BFS with update of parent pointers as in original

algorithm).

4.4 Integration with parallel visitors

As mentioned above the standard way of parallel verification in DiVinE is using partitioned setup, that means states are statically partitioned to threads (using their hash), this setup can be optionally extended to network of workstation.

In this setup tree compression can work only on per-thread basis, compressing inside each partitioned hash-tables independently (as hash tables are not thread safe in this setup). This means tree compression will get less efficient as 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 prior to their sending to IPC queue).

This disadvantage will be eliminated in future re-

leases of DiVinE as new mode of verification using growing hash-table shared by all threads on machine is being developed in this time [16]. Experimental implementation of shared hash-table was integrated with tree compression for the purpose of experiments.

The Figure 4.3 shows schematics of two worker threads running partitioned visitor. Each of threads hast its own local queue and it also has IPC queue for each other worker including itself. While local queue contains only *from* states, IPC queue contains whole transitions, that is *from* and *to* edges. Arrows in figure are showing pointers to data[6], on the top there are temporary data, tree compression hash-tables are in bottom of 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 of IPC queue belongs to hash-table 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

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

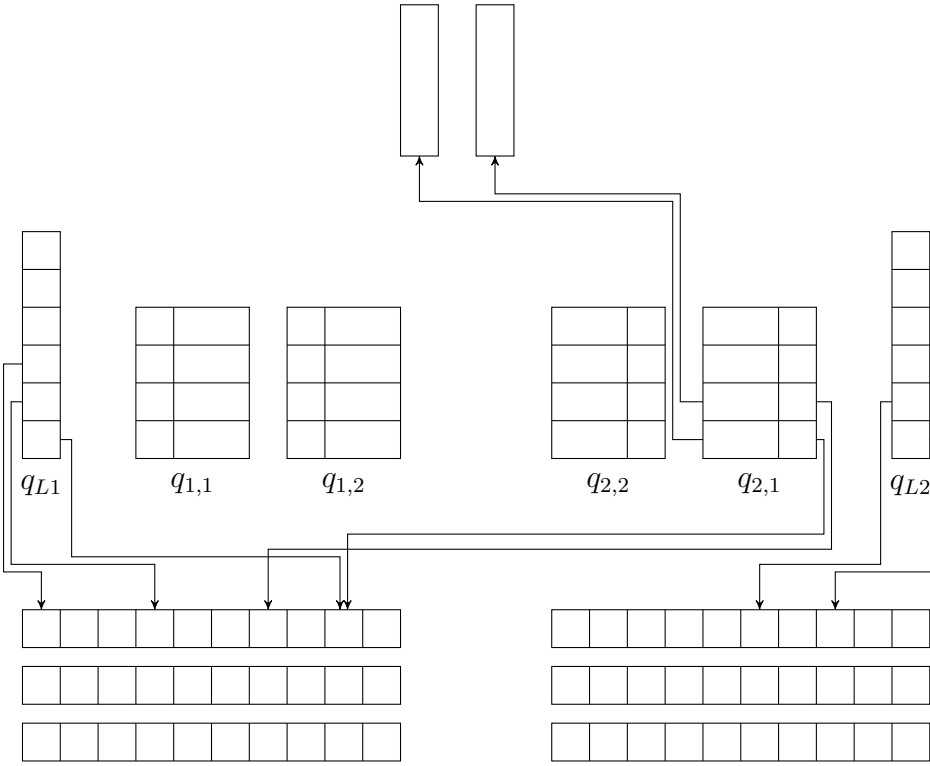


Figure 4.3: Partitioned visitor with tree compression

be possible to decompress state without direct access to hash-tables of other workers (otherwise all workers would need to have access to all hash-table, which would be complicating design).

Chapter 5

Experiments

5.1 Settings

Several measurements were performed comparing tree compression implementation in DiVinE to default verification mode of DiVinE, that is hash-table without compression. If not sated otherwise all memory usages are peak virtual memory as stated by DiVinE in *Memory-Used* of report, that is maximal amount of memory addressable from run of DiVinE on given model, and time is value of *Wall-Time* of report, that

is real time of running.

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 supported algorithms (`--reachability`, `--owcty`, `--map`, `--nested-dfs`), `M` is memory bound for 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 originated from several sources:

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

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

Modifications of above in most cases models could have been extended to bigger instances

5.2 Timed automata

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

Figure 5.1 shows impact of tree compression on model 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 exception of boxes.xml and fixer.xml property holds.

It can be seen that impact of tree compression rises with state space sizes. As Timed automata has quite large states, impact is significant for instances having roughly half a milion states and more, those instances can be checked with minimal time overhead.

Figure 5.3 shows measurements of MAP algorithm for LTL verification. MAP algorithm was used instead of OWCTY (which is asymptotically faster) because it integrates with tree compression better. With

OWCTY IPC queues are used even in single threaded case, in manner which pushes whole spate space into them (as *to* vertices, therefore uncompressed). Obviously this mitigates any savings of tree compression. OWCTY running with shared visitor does not have this drawback.

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

Note that in DiVinE running LTL verification, 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 or tree root and part of state which contains position in property automaton.

5.3 LLVM – C and C++ programs with threads

LLVM interpreter is used by DiVinE for verification of parallel C and C++ programs which are automatically translated to LLVM bytecode [14, 2]. Thanks to reductions described in [14], most LLVM models in DiVinE distribution are quite small to be considered as candidates for compression, despite that all instances were tested, table is divided into two parts, first shows models, that cannot be verified in less than 1GB when not compressed, smaller models are in second part.

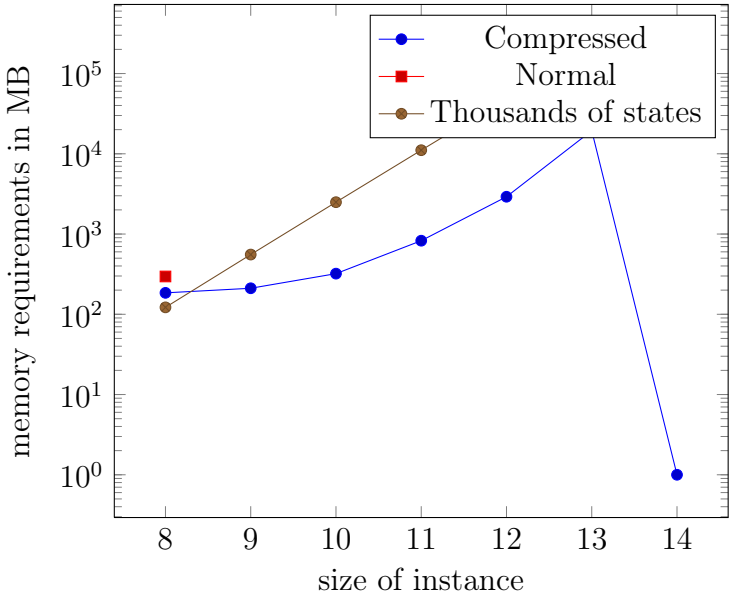
Experiments showed that for LLVM, even models with tens of thousands of states can benefit from tree compression and with millions of states the benefit is massive. Also time penalty is small. No particular limits were set for those tests, but airlines model failed to terminate within more than one day so it was omitted (tests were performed on 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, models were chosen in this way: first larger instances of models were chosen (instances with more than 5 workers if at least of them were available, otherwise two biggest), then models were verified with memory limit of 7GB without partial order reduction. 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 while having quite large tome overhead. This is caused by fact that DVE models have quite small states. Also 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 per one state is roughly 93 bytes in uncompressed verification (computed as ratio of overall mem-

ory and number of states). Of course this is just upper bound, including overhead of all data structures used by DiVinE (such as hash-table 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.



size of instance		8	9	10	11	12	13	14
memory [MB]	normal							
	compressed	185	211	321	826	2916	19420	
millions of states		0.12	0.55	2.4	11.1	48.8	212.9	

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

	memory [MB]			time [s]			#
	normal	ntree	ratio	normal	ntree	ratio	sta
boxes	160	176	110.0%	0	0	113.3%	8.9
bridge	156	172	110.3%	0	0	74.1%	206
fischer	156	172	110.3%	0	0	107.3%	5.8
fischer8	296	188	63.4%	6	6	108.3%	122.
fischer9	954	228	23.9%	38	38	101.4%	555.
fischer10	4517	410	9.1%	201	212	105.7%	2.5
fischer11	–	1174	–	–	1262	–	11.1
fischer12	–	4163	–	–	7810	–	48.8
fischer13	–	–	–	–	–	–	–
fixer	164	176	107.3%	1	2	135.4%	205.
train-gate8	445	256	57.6%	24	27	111.6%	726.
train-gate9	3420	1249	36.5%	218	289	132.7%	6.5
train-gate10	–	–	–	–	–	–	–

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

	memory [MB]			time [s]			# c
	normal	ntree	ratio	normal	ntree	ratio	stat
bridge	129	164	126.9%	0	0	130.2%	1.6
fischer9	2575	344	13.4%	1001	1693	169.1%	1.7
fischer10	3599	345	9.6%	206	349	169.8%	31K–6
fischer11	–	1044	–	–	2298	–	117.4
fischer12	–	4005	–	–	12921	–	195.9
fischer13	–	–	–	–	–	–	–
fixer	3604	3150	87.4%	446	644	144.6%	214K–

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

	memory [MB]			time [s]			
	normal	ntree	ratio	normal	ntree	ratio	
airlines	–	–	–	–	–	–	
elevator	5191	772	14.9%	4238	4417	104.2%	8
elevator2	75963	1792	2.4%	91575	91430	99.8%	1
pt_barrier	9247	1126	12.2%	4583	5052	110.2%	
pt_rwlock	25191	1845	7.3%	19604	22811	116.4%	
anderson	218	269	123.7%	0	1	151.0%	
at	414	285	68.8%	115	174	151.0%	4
bakery	254	305	120.3%	2	3	133.7%	
bridge	160	205	127.8%	0	0	247.8%	
collision	600	596	99.2%	97	196	203.2%	1
cyclic_sched	607	427	70.2%	202	255	126.2%	4
elevator_plan	170	207	121.9%	1	1	109.4%	
fifo	255	306	120.1%	2	2	111.6%	
fischer	334	289	86.6%	74	121	163.0%	2
global	193	243	125.6%	0	0	136.7%	
lamport	824	315	38.2%	334	362	108.2%	1
lamport_n1	247	299	120.8%	4	5	116.5%	
lamport_n2	215	267	123.9%	0	0	109.7%	
lead-uni_b	481	433	90.0%	217	250	115.4%	1
lead-uni_dkr	294	345	117.5%	2	2	116.5%	
lead-uni_pt	458	452	98.6%	84	91	108.9%	
peterson	191	241	125.7%	1	1	119.7%	
pt-showcase	465	517	111.0%	2	3	120.0%	
pt_cond_vars	361	412	114.2%	10	12	118.8%	
pt_mutex	203	255	125.3%	0	0	129.7%	
ring	187	239	127.5%	3	3	115.9%	
szymanski	280	331	118.4%	3	4	116.2%	

Figure 5.4: LLVM compression.

	memory [MB]			time [s]			
	normal	ntree	ratio	normal	ntree	ratio	
anderson.5	5590	6847	122.5%	156	433	277.1%	5
anderson.6	1484	1510	101.7%	66	197	299.5%	1
anderson.7	–	–	–	–	–	–	
at.5	2841	4285	150.8%	153	312	204.4%	3
at.6	–	–	–	–	–	–	
bakery.8	1706	1377	80.7%	86	199	232.2%	1
exit.4	1506	1493	99.2%	67	139	207.5%	1
firewire_tree.5	3151	834	26.5%	195	452	231.7%	
hanoi.3	1963	2920	148.7%	52	156	299.7%	1
hanoi.4	–	–	–	–	–	–	
iprotocol.5	3403	2489	73.1%	89	266	297.9%	3
iprotocol.6	4008	2483	61.9%	139	339	244.1%	4
iprotocol.7	6680	4810	72.0%	244	499	204.7%	5
lamport.7	2939	3450	117.4%	122	367	299.4%	3
leader_elect.5	1444	655	45.4%	44	173	391.7%	
leader_elect.6	–	5863	–	–	1661	–	3
leader_filt.6	1588	1359	85.5%	44	133	304.8%	1
mcs.5	5535	5262	95.1%	198	614	310.3%	6
phils.6	1054	935	88.7%	34	145	429.0%	
phils.7	–	–	–	–	–	–	
phils.8	2457	1702	69.3%	127	430	339.3%	1
plc.4	1092	825	75.6%	23503	23647	100.6%	
prod_cell.6	1970	1428	72.5%	50	163	328.0%	1
sokoban.3	–	4831	–	–	1215	–	7
sorter.4	1674	1381	82.5%	182	227	125.0%	1
szymanski.5	6449	4853	75.3%	334	1110	332.4%	7
telephony.6	–	–	–	–	–	–	
telephony.7	1658	1858	112.1%	137	301	220.3%	2
telephony.8	–	–	–	–	–	–	

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

Chapter 6

Conclusion

We proposed improved version of tree compression, a method of mitigating state-space explosion in explicit model checking. It works by replacing standard hash-table by compressed hash-table which is reusing parts of states that are already saved.

The aim was to provide memory efficient storage for model checking of programs with vast 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 particular state, which may contain for example dynamically allocated data. Also, such a pro-

grams have variable size of state and we support this in our tree compression.

The proposed tree compression can work in general way, not depending on particular modeling language and state space generator. In fact it could be used in different environments than model checking as a general hash-map as well. At the same time this tree compression can be easily integrated with specific state-space generator to further improve memory efficiency by specifying optimal shape of tree representation.

Tree compression was implemented in DiVinE model-checker and integrated with most of its functionality (currently with only exception of distributed MPI verification). Tree compression is being used for compression of BFS queues as well, and for partial compression of 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% of memory requirements of uncompressed run) with minimal impact to verification time. As a result of this efficiency, feasibility of LLVM examples shifted (mem-

ory wise) from multi-socket servers or network of workstations to single desktop or laptop computer. This is a starting point to verification of instances for which standard approach would require several terabytes of memory. Also memory overhead of LTL verification can be efficiently mitigated by tree compression.

Integration with parallel verification is supported however it looses some efficiency with partitioned setup used traditionally by DiVinE. This downside will be mitigated by use of new shared memory verification approach being developed in [16] at this time.

6.1 Future work

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

Memory efficiency of tree compression could be further improved by allowing to save certain parts of state explicitly in root of tree. Such parts would include short and heavily changing fragments, such as positions in property automaton when verifiing LTL or

program counters in LLVM.

As 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 interesting field which may further push limits of state-of-the art explicit 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).