Local Nontermination Detection for Parallel C++ Programs*

Vladimír Štill and Jiří Barnat

Faculty of Informatics, Masaryk University, Brno, Czech Republic
divine@fi.muni.cz

Abstract. One of the key problems with parallel programs is ensuring that they do not hang or wait indefinitely – i.e. there are no deadlocks, livelocks and the program proceeds towards its goals. In this work, we present a practical approach to detection of nonterminating sections of programs written in C or C++, and its implementation into the DIVINE model checker. This complements the existing techniques for finding safety violations such as assertion failures and memory errors. Our approach makes it possible to detect partial deadlocks and livelocks, i.e. those situations in which some of the threads are progressing normally while the others are waiting indefinitely. The approach is also applicable to programs that do not terminate (such as daemons with infinite control loops) as it can be configured to check only for termination of selected sections of the program. The termination criteria can be user-provided; however, DIVINE comes with the set of built-in termination criteria suited for the analysis of programs with mutexes and other common synchronisation primitives.

1 Introduction

Assessing correctness of parallel programs is a hard task even for experienced programmers. Therefore, the standard program development includes a bunch of quality assurance activities such as testing. Unfortunately, the nondeterministic nature of thread scheduling and concurrency makes it quite hard for testing to achieve good guarantees of quality in the case of parallel programs. Formal methods, on the other hand, provide a more systematic approach and in some cases may even prove the absence of erroneous behaviours. However, they are not used very often in practice due to the extra effort required for their application or simply because they are not powerful enough to handle the overall complexity and size of real-world programs. Nevertheless, continuous improvement in formal methods is desirable to cover the corner cases of their use and to allow them to become more usable in software development.

Especially beneficial techniques are those that can be directly applied to programs written in mainstream programming languages. Such techniques significantly lower the barrier towards their usage by programmers. However, the

* This work has been partially supported by the Czech Science Foundation grant No. 18-02177S.

```
// can be used for synchronization
std::atomic< int > x = 0;

void worker() {
  while ( x != 0 ) { } // wait
  do_work();
}
```

```
int main() {
  // start thread running worker
  std::thread t( worker );
  x = 42; // let worker run
  // ...
  t.join();
}
```

Fig. 1. A simple C++ code snippet with two threads, it uses C++ standard threading support and atomic variables. A programmer's intention was that the `worker` function first waits until `x` becomes non-zero, and then proceeds with `do_work`. However, the waiting condition (at the first line of the `worker` function) is incorrectly just the opposite. Therefore, if `main` executes `x = 42` before waiting in `worker` starts, the wait will never end (assuming `x` is never set to 0 again). Note that none of safety checks is able to detect that the program might hang. For the rest of the paper, we will omit the `std::` namespace to simplify the notation.

development of these techniques is extremely demanding due to numerous specific features the real-world programming languages exhibit. As a result, many techniques introduced and implemented stay at the level of prototypes without being mature enough to be applicable outside the scientific community – e.g., they might be missing features such as pointer arithmetic, functionality of the standard libraries or the concept of exceptions. See Software-Verification Competition (SV-COMP) [6], to find some examples of tools aiming at verification of real-world programs written in C.

A significant limitation of many existing tools for analysis of parallel programs in programming languages such as C and C++ is that they are only concerned with safety checking – they check that a bad state of the program is unreachable. Most common examples of bad states include assertion failures and memory errors (such as invalid memory accesses and memory leaks). Unfortunately, this is far from being sufficient in practice. See, for example, the code given in Figure 1. That piece of code easily passes any safety checks; however, when executed in reality, it often hangs and does not terminate.

In this paper, we report about our new technique for checking nontermination for parallel programs written in C and C++ that may be applied to programs with arbitrary synchronisation primitives. In particular, we can check that *a specified part* of a program finishes whenever its execution has been started, which in turn enables us to check for problems such as partial deadlocks or local nontermination. Note that our technique does not require the program under analysis to terminate at all. Therefore, it is also applicable to programs that do not terminate but have some parts that are supposed to finish. It does; however, require that the program has a finite state space because our technique is built on top of a state space exploration. Note that even for a finite state space, a program may exhibit infinite behaviour.

The main observation is that a program often has sections which once entered should also be left: for example critical sections, certain function calls (such as a pop from a queue, which can wait for an element to become available; or a thread join, etc.), or parts of code which wait for a resource or an action (waiting for a mutex, waiting on a barrier, waiting until a variable is set to a given value). If the analysis of the program focuses on such sections, it is possible to detect when these sections are started, but do not terminate. This covers partial deadlock and partial livelock detection in which such sections participate. We also provide a global nontermination detection mode that decides if the program as a whole terminates, nevertheless this is not the primary goal of our approach.

Our technique is built on top of explicit-state model checking. We believe that while explicit-state model checking is prone to state space explosion, it is well suited for the detection of problems related to infinite runs of parallel programs which cannot be handled by techniques such as bounded model checking or stateless model checking. While our approach is closely related to checking for properties written in temporal logic such as LTL or CTL*, our *local nontermination* technique cannot be substituted equivalently with CTL* model checking. One of the reasons is that these logics are unable to relate to entities which are dynamically created during the execution of the program, and there is no bound to their number. For example, there is no way to express in CTL* that for all mutexes it holds that if they are locked, they are also eventually unlocked unless all the mutexes are enumerated beforehand. This is an essential concern for realistic programs where mutexes and other synchronisation primitives can be created dynamically at runtime, and their number can depend on the computation of the program itself. Furthermore, to avoid counterexamples which are unrealistic with practical thread schedulers, we need a form of *fairness* of process scheduling different from fairness constraints used typically with LTL model checking.

The approach described in this paper is implemented in a modified version of the DIVINE model checker [3, 13]. The implementation, as well as all the examples, can be found on the paper webpage[1].

The rest of the paper is structured as follows: Section 2 gives a short overview of related work and Section 3 gives definitions and preliminaries needed for the rest of the work. In Section 4 we define our *local nontermination* property, in Section 5 we discuss how it can be checked and the implementation in DIVINE, and it Section 6 we evaluate it. Finally, Section 7 concludes this work.

2 Related Work

For related work, we consider only results which go beyond safety checking. There are many approaches to find problems such as assertion violations or memory safety violations, but they are fundamentally limited to properties concerning finite runs of the program, and we are focusing here on infinite behaviour, namely

[1] `https://divine.fi.muni.cz/2019/lnterm/`

absence of termination. Similarly, we do not mention techniques which specialise on checking sequential programs and have no support for parallelism, as well as techniques which are tailored to specific modelling language and cannot be applied in general.

Several techniques for checking properties other than safety exist – indeed usage of various temporal logics, such as Linear Temporal Logic (LTL) [2, Chapter 5] and Computation Tree Logic (CTL) [2, Chapter 6] in the context of model checking dates way back to the beginning of research in formal methods. Unfortunately, these are not often applied to programs written in real-world programming languages such as C and C++.

As for techniques which detect nontermination, both static and dynamic techniques exist for detecting deadlocks caused by circular waiting for mutexes [7, 1, 5]. However, these techniques specialise on mutexes and do not allow general nontermination detection, and it is unlikely that they could be naturally extended to cover it. There are also techniques that detect deadlocks of the whole program (i.e., a program state from which the program cannot move) [8, 9], but these techniques cannot find cases in which only some threads of the program are making progress, while other threads are blocked forever. Also, these global deadlock detection techniques are inadequate in the presence of synchronisation which causes busy waiting instead of blocking (for example spin locks) or in the cases when normally blocking operations are implemented using busy waiting (which can be easier to handle for the verifier in some cases). A somewhat different approach based on communicating channels is proposed in [11], but this approach is aiming at the Go programming language which primarily uses shared channels for communication between threads. Overall, neither of these techniques is applicable in general for detection of nontermination in programs which use a combination of synchronisation primitives in shared memory.

3 Preliminaries

In this section, we shortly describe necessary details about representation of programs, their state space, and resource sections so that we can define local nontermination.

3.1 State Space of a Program

The *state space* of a program is a directed multigraph with labelled edges. The vertices of the state space multigraph are called *states* (of the program). Each state represents a snapshot of the program (its memory, program counters, ...). States v_1, v_2 are connected by an edge in the state space if v_2 can be reached from v_1 in an atomic step, which is a sequence of instructions that executes at most one action which can interfere with any action executed in parallel with it. In DIVINE, the state space generator attempts to make the longest possible atomic step while ensuring that the generation of the edge terminates. Edges are labelled, and the labels can be used to indicate accepting edges and error

edges. Error edges are edges on which safety violation occurs (e.g., an assertion violation or memory error). The notion of accepting edges was taken initially from transition-based Büchi automata and used for LTL model checking, but in general, it is a way to mark an edge as interesting for the verification algorithm, but not erroneous. These labels are set by the verified program, which can be instrumented to influence edge labelling or by DIVINE when it detects an error.

The state space of a program can be an infinite graph. However, in DIVINE, we are primarily concerned with programs which have finite state space. If the state space is infinite, DIVINE might find an error if it is present there, or it might compute until its resources are exhausted. Please note that programs with finite state space can have infinite behaviour as they can loop through the same set of states indefinitely.

3.2 Resource Sections

A *resource section* of a program is a block of code with an identifier of a resource and type of the resource section. Each resource section is delimited in the source code by section start and section end annotations. Examples of such sections are a mutex-waiting section that denotes a block of code in which a thread is waiting for the acquisition of a mutex. Mutex-waiting section is identified by a mutex and the thread which waits for it. Another example can be a critical section, which is identified by a mutex (there is no need to use thread for the identification, as a mutex can be owned by at most one thread at any point in time). Resource section can also be bound to a function – in this case, it is identified by the stack frame of the function and by the program counter of its beginning. Regardless of the identification, the idea for a resource section is that once it is entered, it should also be exited.

As a resource section can be entered repeatedly (for example when it is on a cycle or in a function which is called multiple times) we will define a *resource section instance* to be a particular execution of a resource section with the given identifier. The author of annotations which define resource sections should ensure that the same resource section is not entered again before it is left. Please note that this does not limit the usage of function-associated resource sections to non-recursive functions – each such section is also identified by the stack frame, and therefore resource sections corresponding to different recursion depths are different resource sections. Similarly, a program can be in multiple resource sections which wait for the same mutex at the same time, each of them corresponding to a different waiting thread.

4 Local Nontermination

With our local nontermination property, we aim at detection of resource section instances which are entered but are never left – *nonterminating resource section instances*. We will first use examples of terminating and nonterminating resource section instances, and then we will define them precisely.

```
mutex m;

void thread0() {
  unique_lock lock(m); // Error
  while (true) {
    do_work();
  }
} // unlock

void thread1() {
  while (true) {
    unique_lock lock(m);
    do_other_work();
  } // unlock
}
```

Fig. 2. A program with nonterminating critical section (in `thread0`) and a deadlock (if `thread0` enters its critical section, `thread1` will wait infinitely). Please note that in C++ it is possible to use scope-based locks: the critical section belonging to mutex `m` is entered when `unique_lock lock(m)` is executed and left at the end of the scope in which the `lock` variable was defined (at the matching curly brace; also marked with comment `// unlock`).

```
mutex m;

void thread0() {
  while (true) {
    unique_lock lock(m); // Fixed
    do_work();
  } // unlock
}

void thread1() {
  while (true) {
    unique_lock lock(m);
    do_other_work();
  } // unlock
}
```

Fig. 3. A fixed version of the program from Figure 2 (the start of the critical section was moved from the position `// Error` in the left code to `// Fixed` and therefore the critical section can end now). Intuitively, each critical section in this program terminates. However, as we can see in Figure 4, it is possible to find an infinite path in the state space of this program that infinitely waits for one of the critical sections. To make matters worse, this path can respect weak fairness.

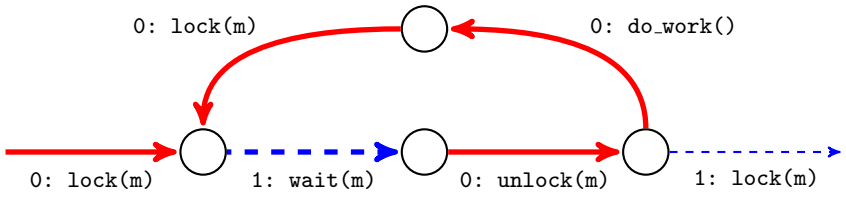

Fig. 4. A fragment of state space of program in Figure 3 with starving lasso marked with bold edges. Each edge is marked by the thread it belongs to and the action of this thread. Furthermore, to ease the orientation, actions belonging to `thread0` are marked with continuous red edges while actions belonging to `thread1` are marked with dashed blue edges. We can see that both threads participate in the repeated part of the counterexample and `thread0` is denied the possibility (starves) to execute after `0: unlock(m)` (the thin blue dashed edge).

A simple example can be seen in Figure 2. There we have a mutex which is locked, but never unlocked as the corresponding critical section contains an infinite loop. We have four different resource sections in this example. Two of them corresponds to the critical sections guarded by the mutex, and two of them are hidden inside `unique_lock`, where they implement waiting until the mutex is unlocked. Nonterminating resource section instances are the instances corresponding to the critical section in `thread0` and any instances corresponding to waiting for mutex in `thread1` which is executed after the critical section in `thread0` is entered. We can fix this example by putting the critical section in `thread0` inside the infinite loop, as shown in Figure 3.

Suppose that we have defined nonterminating section as one in which it is possible to stay indefinitely (i.e., for the specific case of waiting for `m` in `thread1`, termination could be expressed by LTL formula $\mathbf{G}(\textit{wait-m-t1-start} \implies \mathbf{F}\ \textit{wait-m-t1-end})$). We can witness the existence of such nonterminating section in a program with a finite state space by a lasso-shaped path. Such the nontermination witness can also be found for the program in Figure 3, even though the code might intuitively seem to terminate. First `thread0` executes its `lock` action, then `thread1` starts waiting. If `thread0` always executes `unlock` and `lock` before `thread1` is allowed to run, `thread1` will never be able to finish waiting. The counterexample is illustrated in Figure 4 and is valid also under weak fairness assumptions.

In general, if a thread waits for some condition which is both infinitely often true and infinitely often false, there can be a run in which the waiting thread is only allowed to run at those moments when the condition is false. This type of run is present in any program that uses busy waiting, which is very common in practice. For this practical reason, we cannot rely on the definition of nontermination as expressed with the LTL formula above, and we need a different way to describe nontermination sections.

Definition 1 (Nonterminating resource section instance). *A resource section instance is nonterminating if and only if it can reach a point from which it is not possible to reach its end.*

For a particular resource section (e.g., again waiting for `m` in `thread1`), checking for absence of nonterminating resource section instances can be expressed using a CTL* property

$$\mathbf{AG}\,(\textit{wait-m-t1-start} \implies \mathbf{A}[(\mathbf{EF}\ \textit{wait-m-t1-end})\ \mathbf{W}\ \textit{wait-m-t1-end}])$$

(where **W** is the weak until operator).

In general, the CTL* approach cannot be used, as it requires the set of resource sections to be known before the analysis starts, so that the formula can be created as a conjunction of formulas for each resource section. This is hard to do if resource sections can be created at runtime, which is often the case when dealing with programs in languages such as C and C++.

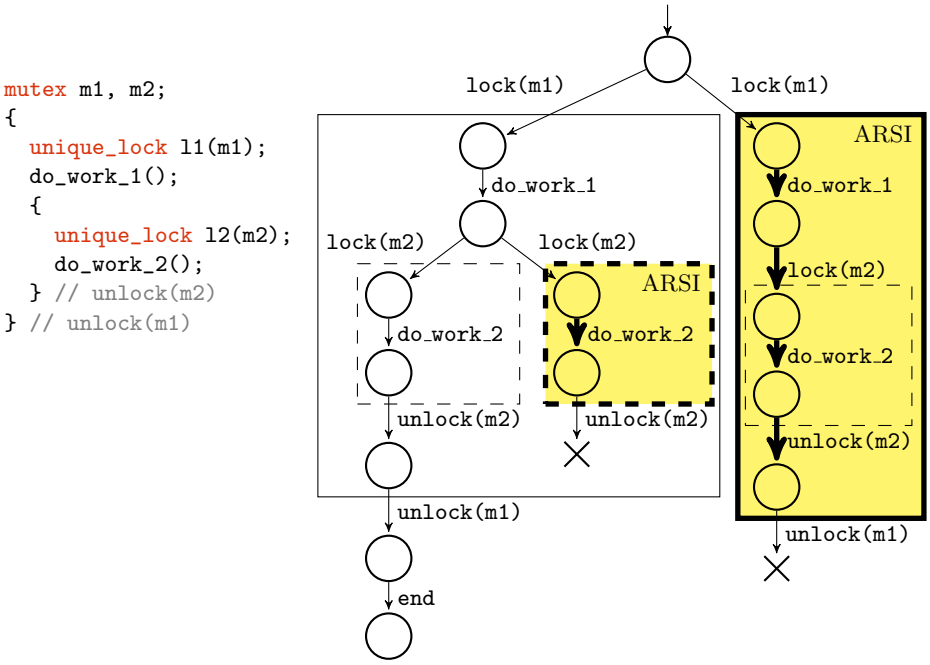

Fig. 5. A small example of a program with two resource section instances (on the left-hand side) and its state space, which shows active resource section instances (ARSIs; on the right-hand side). The resource section instances belonging to the critical section of mutex `m1` are wrapped in a solid rectangle in the image, while resource section instances belonging to `m2` are wrapped in a dashed rectangle. Active resource section instances are denoted by thick frame and yellow background and accepting edges in the state space are marked by thick arcs. Please recall that active resource section instances cannot be nested. Crosses at the end of edges denote points where exploration of the state space was terminated due to reaching the end of an active resource section instance.

5 Detection of Nontermination

The detection of nonterminating resource section instances in the context of explicit-state model checking proceeds as follows. The basic idea behind the detection of nonterminating resource section instances is that the model checker focuses on them one at a time. Every time a resource section instance is about to be entered during the state space exploration, the algorithm introduces a nondeterministic branching to the state space graph. In one branch the resource section instance remains inactive, in which case the state space exploration proceeds as usual to discover other resource sections. However, in the other branch, the instance becomes active. Under this branch the resource section instance is checked for being nonterminating. Note that the nondeterministic branching happens only outside of active resource sections, which means the *active resource section instances* (ARSIs) cannot be nested. Once the state space graph in the

active branch reaches a state that is out of the scope of an ARSI, the state space exploration within this branch is stopped (a state with no successors is generated outside the ARSI). Active resource section instances cannot be nested, but for any instance of a resource section nested in an active section instance, there is also an instance which is nested in an inactive section instance, and therefore can become active elsewhere in the state space. As a result of this construction, for every nonterminating resource section in the original program, there is a corresponding ARSI in the augmented state space graph. To let exploration algorithm know that it is exploring a part of the state space that is within an ARSI, we mark all edges within ARSIs as accepting. An illustration of a state space graph augmented with the nondeterministic choices is given in Figure 5. Now to discover ARSIs which are nonterminating according to Definition 1, it is enough to detect terminal strongly connected components made of accepting edges only.

5.1 Detection Algorithm

Henceforward, we assume the state space graph is finite, and if the program to be verified terminates then this fact is reflected with a state with no successors in the underlying state space graph. Note that the program may terminate even within a resource section instance. An ARSI terminates either by reaching the end of the section instance, or by the termination of the whole underlying program. In both cases, this means a state with no successors is generated and reachable from the ARSI entrance point. As a result, the detection of nonterminating ARSIs can be performed as a search for an accepting terminal strongly connected components in the state space graph.

Definition 2 (Terminal Strongly Connected Component). *A strongly connected component S is* terminal[2] *if for each state v in S all successors of v are in S (there are no edges out of S).*

Definition 3 (Fully Accepting Terminal SCC). *A terminal strongly connected component of the state space is* fully accepting *(*fully accepting terminal SCC, or FATSCC*) if and only if it is nontrivial and all its edges are accepting.*

Theorem 1. *A program contains a nonterminating resource section instance if and only if its state space graph contains a fully accepting terminal strongly connected component.*

Proof. Assume the program contains a nonterminating ARSI $\mathcal{A}$. Then there must exist a set of states in $\mathcal{A}$ from which neither program end nor the corresponding resource section end can be reached. Among these states, there must be a subset which can be repeated indefinitely and cannot be left – a nontrivial terminal SCC which is part of an ARSI and therefore it is fully accepting – a FATSCC in the state space.

[2] Also sometimes called bottom strongly connected components, or closed communicating classes, especially in the area of probabilistic system analysis [12].

For the other direction let us assume that there is an FATSCC in the state space graph. Since any edge which enters or leaves an ARSI is not accepting (which follows directly from the construction of the state space graph), all states that are part of the FATSCC must be states within a single ARSI. Since the component is terminal and non-trivial, it is impossible to reach either program termination point or a state that would be outside of the resource section instance, therefore, the FATSCC witnesses a resource section instance that does not terminate. □

To detect the presence of a FATSCC in the state space graph we employ the standard Tarjan's algorithm for finding strongly connected components. To detect if a terminal component is nontrivial and fully accepting it is enough to check that the component contains at least one state with some successors (it is nontrivial) and that all states of the component have only accepting outgoing edges (it is fully accepting), which is just a minor modification of the algorithm.

Note that it is also possible to define *global nontermination* using Definition 1. In this case we only need to treat the whole program as a single active resource section instance.

5.2 Scheduling and Fairness

To provide further context, we also want to discuss the relation of our nontermination property to LTL model checking with fairness. Fairness constraints [2, Chapter 3.5] are needed in analysis of temporal properties of parallel systems to avoid reporting of unrealistic counterexamples, such as those in which an enabled thread never gets the chance to make an action. Basically, even if we use LTL formula to describe nontermination and allow for LTL model checking under weak fairness, we still may obtain counterexamples that are totally unrealistic. This is because a weakly-fair scheduler[3] admits runs in which the context switches that happen among participating threads are very regular, hence unrealistic.

The nontermination as defined in Definition 1 can be seen as a manifestation of an additional assumption about the thread scheduler. It claims that the scheduler is in the essence somehow irregular, i.e., it will not allow for a context switch always after a fixed number of instructions or at a specific location in the code. Another way of looking at this is to assume that the scheduler is probabilistic and assigns some non-zero probability to interruption between any two instructions. With a probabilistic scheduler, we can equivalently define nonterminating resource section instance as a section instance which can get to the point when there is zero probability of reaching its end. Under the probabilistic view we

[3] For our purposes, a weakly-fair scheduler is a scheduler which ensures that on every accepting cycle in the state space all threads which existed during the execution of this cycle were also executed at least once on the cycle. To make sure this definition of fair scheduler provides reasonable semantics, we further require that threads do not block (they can, however, exit). This is not a problem in practice as any blocking synchronisation (such as waiting for a mutex) can be simulated by a busy waiting loop.

can also say that programs we denote as correct, i.e., without nonterminating sections, have zero probability of looping forever.

5.3 Implementation and Usage

We have implemented our nontermination detection approach in a branch of the DIVINE model checker. Resource sections can be specified by annotating the source code of the program to be analysed by the user of the tool. Furthermore, DIVINE provides predefined resource sections for various POSIX thread (`pthread`) synchronisation primitives, namely for mutexes (including recursive and reader-writer mutexes), condition variables, barriers, and joining of threads. Since C++ threading support in DIVINE uses the libc++ library which uses POSIX threads, these resource sections are also used for native C++ threading.

User-defined annotations can be given in one of the following categories: exclusive section, waiting for an event, and waiting for function end. For user-defined resource sections, DIVINE provides C and C++ interface which can be found on the web page accompanying this publication.[4] To make it possible to specify which resource section types should be considered for analysis, we use program instrumentation, which enables resource sections based on commandline arguments (for more details see the accompanying web page).

The detection of nonterminating resource sections in DIVINE uses Tarjan's algorithm for finding strongly connected components. The algorithm runs on-the-fly, which means that it generates the state space graph as needed, and therefore, it can terminate before the entire state space graph is explored. The algorithm finishes if it finds a fully accepting terminal strongly connected component, if it discovers a safety error (to avoid the need for a separate safety verification), or once the entire state space is explored.

5.4 Interaction with Other Features of DIVINE

Since DIVINE is a research tool not all the features implemented within the tool are expected to run together. In this case there are even some other features of DIVINE which interfere with local nontermination detection in a not so obvious way.

Counterexamples When an error is found DIVINE has support to show a counterexample and walk through it using an interactive simulator [3]. For safety properties, this counterexample is a sequence of states which ends with an error. For verification of properties described by LTL or Büchi automata (which are partially supported by DIVINE), the counterexample is a lasso-shaped trace. For nontermination, the part of the state space to be reported consists of a fully accepting terminal strongly connected component and a path that leads to it. However, it is not practical to output the information about the whole SCC, as it can be large. For this reason, DIVINE gives only a trace to the first state of the

[4] `https://divine.fi.muni.cz/2019/lnterm`

fully accepting terminal SCC (i.e., the first state from which end of the given resource section instance is not reachable).

Spurious Wakeups Condition variables are often used in parallel programs to block threads until some event occurs (e.g., a shared queue becomes non-empty). They provide a function which blocks the current thread (`wait`) and a function which signals the condition variable and causes waiting threads to proceed (`signal`). In most implementations, including C++ standard APIs and platform-specific APIs on Windows and Linux, `wait` is allowed to return before it is signalled: this behaviour is called *spurious wakeup* and programmers must take it into account when using condition variables.

To help with the discovery of bugs caused by spurious wakeup, DIVINE simulates spurious wakeup using nondeterministic choice. For nontermination detection, it is necessary to ensure that any spurious wakeup does not hide nontermination – we want to report resource section instances which can be only left by spurious wakeup as nonterminating. This can be done by careful implementation of the `wait` function in DIVINE – it first nondeterministically decides if a spurious wakeup will happen, and then, if it is not happening, it enters resource section which waits for `signal` and cannot be woken up spuriously. If the spurious wakeup is simulated, it behaves as if the thread was blocked and allows other threads to run. Once the waiting thread is used again for generation of successor states, it is unblocked and `wait` returns spuriously. The exhaustive enumeration of possible thread interleavings ensures that other threads can run arbitrarily long.

Data Nondeterminism and Symbolic Data To make it possible to verify programs that depend on input data, DIVINE has support for symbolic values [10]. In an analysis of programs with symbolic values, the computation can be split when a branch depends on a symbolic value. This splitting can cause problems for nontermination detection if leaving some resource section instance requires a particular value of an input variable. Therefore, in the presence of symbolic data, nontermination checking might miss some instances of nontermination. We defer this problem to future work.

Relaxed Memory Models DIVINE has support for analysis of parallel programs under the x86-TSO memory model of Intel and AMD CPUs [14], which allow the program to exhibit behaviour not present under the interleaving semantics of threads. One of the main problems in interaction between nontermination and relaxed memory is that relaxed memory models over-approximate the possible behaviours of the system to cover all possibilities of contemporary processors of a given architecture. As nontermination is checking for *absence of termination*, it can spuriously hide nontermination if the state space of the program is over-approximated. Again, we defer this problem to future work.

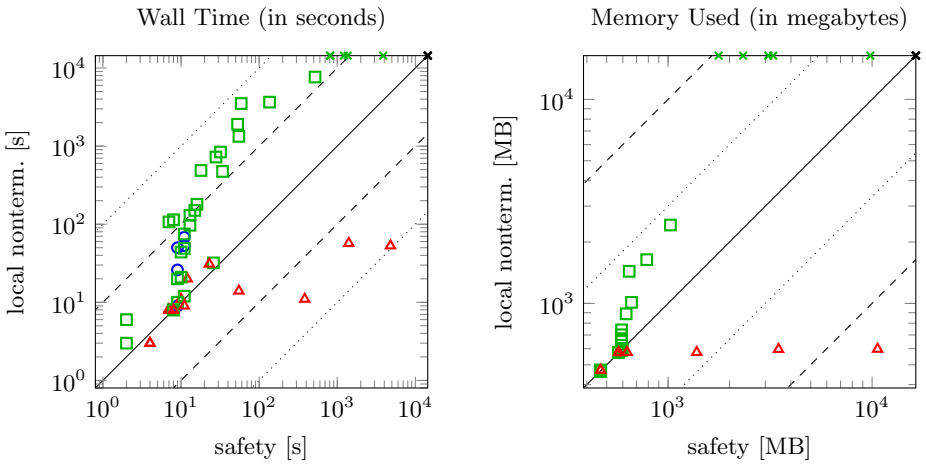

Fig. 6. Scatter plots which compare local nontermination detection with safety checking as implemented in DIVINE. Please note that both axes use a logarithmic scale. The dashed and dotted lines in wall time graphs signify 10× and 100× difference respectively. For graphs of memory usage, the dotted lines signify 3× difference and the dashed 10× difference. Green squares correspond to benchmarks which were error-less in both modes and blue circles correspond to benchmarks which contained errors in both cases. Red triangles correspond to benchmarks which contained a nonterminating section. The crosses on the outer edge of the plot correspond to timeouts and out-of-memory errors. All the failures for local/global nontermination were due to timeouts, benchmarks which failed with out-of-memory did so in all cases.

6 Evaluation

To our best knowledge there is no suitable benchmark set that would cover termination in parallel programs, therefore, we had to develop a suitable benchmark on our own. We naturally wanted to analyse performance of our verification method on real-world data structures. Unfortunately, it is hard to reuse any existing real-world test cases of parallel data structures for verification, as these tests are usually developed as stress tests. Stress tests use large amount of data and are supposed to be run for a long time in order to maximise a chance that a parallelism-related bug is found during the testing period. For the purpose of application of formal verification tool such as DIVINE, the mentioned approach to testing of parallel programs is inappropriate. Since a model checker explores systematically all interleavings of the program within a single execution, further repeated executions, such as the ones within a stress test, are useless and only add to the complexity of verification task. For these reasons, the tests we included in our benchmark are tests we created or adapted and modified specifically for the purpose of nontermination detection we wanted to evaluate.

To preserve some diversity at least, we opted for the following tests to be included in our benchmark. First, to cover some real-world scenarios, we created

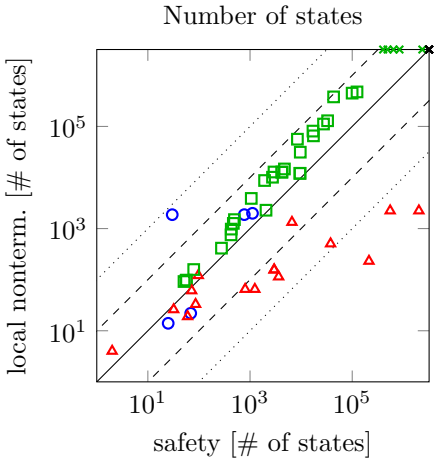

Fig. 7. A comparison of state space sizes for local nontermination and safety. The dashed and dotted lines signify 10× and 100× difference respectively. The meaning of the marks in the graph is the same as in Figure 6.

some tests for the Thread library from widely used C++ Boost[5] (35 test cases). Second, we used some tests from DIVINE project itself (8 test cases), and finally we developed a couple of specific tests for small programs demonstrating behaviour of local nontermination with various synchronisation primitives (16 test cases). Overall, the benchmark covered usage of lockfree and mutex guarded parallel data structures (e.g. parallel queues), synchronised variables, less-used synchronisation primitives such as reader-writer locks, or a single-producer-single-consumer queue and the parallel hashset from [4].

To evaluate our verification approach we let each test run with a 4 hours timeout and 16 GB memory limit. We measured runtime and memory requirements for the three following configurations of our tool:

safety A baseline configuration, in which the tool merely generates the state space of the program and checks for the standard safety issues, such as assertion violation, invalid memory access, etc. In this mode no nontermination can be detected.

local nontermination The configuration in which the local nontermination resource section detection is used. Under this configuration, the state space of the original program is expanded with every entrance to the resource section as described in Section 4.

global nontermination The configuration that treats the whole program as a single resource section and detects if it terminates according to Definition 1. Since this configuration does not introduce additional nondeterminism, the state space of the program is roughly the same size as for *safety*.

The difference between local and global nontermination configurations is basically in the shape of the state space; both use the same algorithm (Tarjan's algorithm for SCC decomposition).

[5] `https://www.boost.org/doc/libs/1_69_0/doc/html/thread.html`

Comparison of safety and local nontermination can be seen in Figure 6. We evaluate wall time and memory consumption – in practice heavy duty tools like DIVINE are likely to be used in long-running overnight tests (preferably only if anything relevant for the test changed since the last run), therefore longer runtimes might not be a big problem up to some point, but it is important to test that the verification tasks fit in some reasonable amount of memory. As we can see, the time overhead of local nontermination configuration is quite significant (up to 59×) especially for larger programs which are correct. As for memory consumption, we can see that total overhead is less then threefold, which is mostly due to the state space compression employed by DIVINE.

The wall time blow-up is due to extra nondeterminism introduced by active resource sections – the state space can grow by a factor that is related to the number of resource section instances encountered in the original state space. Note that many resource sections are likely to be very short. For programs that were invalid, i.e., contained some nonterminating resource sections, the verification usually exited faster under local nontermination configuration than under safety configuration, which means that once a nonterminating section is encountered, it is checked relatively quickly. Further insight into comparison of safety and local nontermination can be seen in Figure 7, which compares sizes of state space for these two configurations. Here, we can see that the overhead in the size of state space is lower than the time overhead (less than 10×). The extra time overhead is likely caused by inefficiencies in DIVINE. For example, when DIVINE nondeterministically chooses from N values, it will re-execute instructions between the last remembered state and the point of the nondeterministic choice N times.

Figure 8 shows a comparison of local nontermination with global nontermination and safety with global nontermination. Here, we can see that the global nontermination behaves similarly to safety, with some time overhead caused by the somewhat more involved algorithm. This is well in line with our expectations, as global nontermination does not introduce any extra nondeterminism compared to safety and Tarjan's algorithm runs in linear time with respect to the size of the state space, and so does reachability. This further highlights that the overwhelming part of the time overhead of local nontermination is in the increase of state space size. It is important to note that local nontermination can be applied to programs which run infinitely (but have finite state space) – it can detect if there is a nonterminating resource section in such a program. As state space size and memory consumption is almost the same for safety and for global nontermination, we omit memory and state space size comparisons for the later two pairs of configurations.

Errors Found No errors were found in the C++ Boost tests, on the other hand, all the errors we artificially implanted in the test cases were found. As for the errors which were not deliberately introduced in the tests, we have found one error in a test of a lock-free queue from an older version of DIVINE. The test was part of DIVINE's test suite for a long time and was used to test that the queue works when it is continuously fed with elements while keeping its size

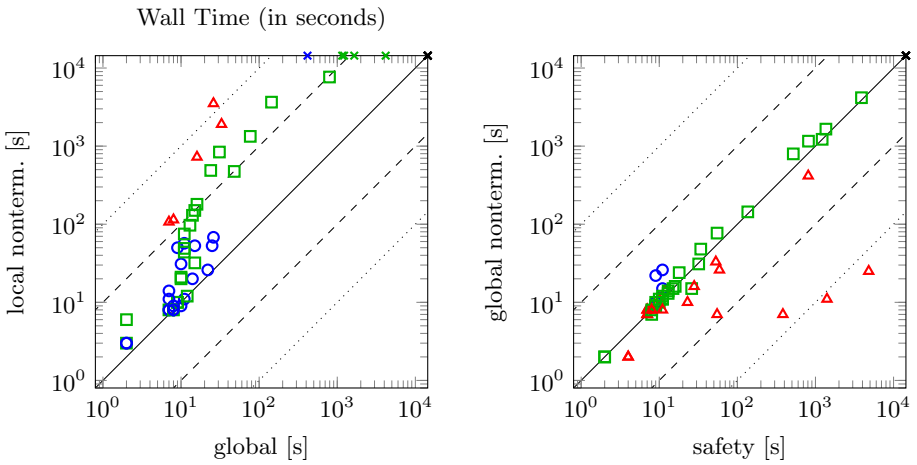

Fig. 8. The first graph compares local nontermination checking with checking if the whole program terminates (global nontermination). In this comparison, the red triangles correspond to benchmarks which did not end, but for which all resource sections terminated. Finally, in the second graph, we compare global nontermination checking with safety. Here, the red triangles correspond to benchmarks which did not terminate but were safe. See Figure 6 for the general description of the plot layouts.

bounded. This means that the test was deliberately non-terminating and the intention was that all the operations executed by main loops of the test's two threads terminate, which was not the case – a variable which was supposed to keep track of the size of the queue was not maintained properly, and therefore it could have happened that the reader thread would wait indefinitely, attempting to read from an empty queue which would never fill up. So far the test case was run under DIVINE with safety algorithm only, therefore the error did not manifest and remained undetected.

7 Conclusion

We have presented a novel approach to detection of parts of real-world programs written in C and C++ which do not terminate. Our method allows for detection of partial deadlocks (and livelocks) caused by misuse of synchronisation, but it is not limited to any particular mode of parallel programming (such as lock-based synchronisation, or programs with communication channels) and indeed allows any combination of synchronisation allowed by C++ itself. To achieve this, it is necessary to provide simple annotations for parts of the code which are to be checked for termination. Our implementation in the DIVINE model checker ships with these annotations already prepared for verification of programs which use C++ blocking synchronisation primitives (mutexes, condition variables), or similar synchronisation primitives from the POSIX threads library (`pthreads`).

Due to the universality of these synchronisation primitives, our annotations allow for checking of most programs which use blocking synchronisation out of the box. For lock-free programs, users have to annotate functions or blocks of code which are required to be exited once they were entered.

We have implemented our technique in an open-source model checker DIVINE, and evaluated it on a set of benchmarks including our tests of the Thread library from widely used C++ Boost. The evaluation shows that while time overhead of local nontermination checking can be quite significant (up to 59× compared to safety checking on our benchmarks), the memory overhead is quite modest (under 3×). During the evaluation, we have discovered a hidden bug that remained in the code for a couple of years, even though the code was subject to intensive safety checking.

Our technique enables checking nontermination in parallel programs, including detection of partial deadlocks and livelocks, and it supports cases when infinitely-running programs contain sections which are supposed to terminate but do not terminate – we believe that even the overhead shown in our evaluation is worth paying for the additional guarantees over safety checking. While related to verification of properties written in temporal logics such as CTL*, our technique cannot be subsumed into CTL* verification, as CTL* cannot quantify over objects which can be created while the program runs.

For future work, it is crucial to further investigate interactions between nontermination checking and relaxed memory, and nontermination and symbolic data representation, as the presence of either of these features can lead to programs being reported as terminating even if they are not in the current situation. Nevertheless, even in the presence of relaxed memory or symbolic data, any reported nonterminating section of the program is indeed a case when the program cannot proceed past the given point. We would also like to investigate better algorithms for detecting local nontermination that might avoid adding nondeterminism to the program under analysis.

References

1. Rahul Agarwal, Saddek Bensalem, Eitan Farchi, Klaus Havelund, Yarden Nir-Buchbinder, Scott D Stoller, Shmuel Ur, and Liqiang Wang. Detection of deadlock potentials in multithreaded programs. *IBM Journal of Research and Development*, 54(5):3–1, 2010.
2. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
3. Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. Model checking of C and C++ with DIVINE 4. In *Automated Technology for Verification and Analysis*, volume 10482 of *LNCS*, pages 201–207. Springer, 2017.
4. Jiří Barnat, Petr Ročkai, Vladimír Štill, and Jiří Weiser. Fast, dynamically-sized concurrent hash table. In *Model Checking Software (SPIN 2015)*, volume 9232 of *Lecture Notes in Computer Science*, pages 49–65. Springer International Publishing, 2015.

5. Saddek Bensalem and Klaus Havelund. Scalable dynamic deadlock analysis of multi-threaded programs. *Parallel and Distributed Systems: Testing and Debugging*, 2005, 2005.
6. Dirk Beyer. Automatic verification of C and java programs: SV-COMP 2019. In *Proc. TACAS, part 3*, LNCS 11429, pages 133–155. Springer, 2019.
7. Yan Cai and W. K. Chan. Magiclock: Scalable detection of potential deadlocks in large-scale multithreaded programs. *IEEE Transactions on Software Engineering*, 40(3):266–281, March 2014.
8. Sagar Chaki, Edmund Clarke, Joël Ouaknine, Natasha Sharygina, and Nishant Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, 17(4):461–483, Dec 2005.
9. Claudio Demartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent java programs. *Software: Practice and Experience*, 29(7):577–603, 1999.
10. Henrich Lauko, Petr Ročkai, and Jiří Barnat. Symbolic computation via program transformation. In *Theoretical Aspects of Computing – ICTAC 2018*, pages 313–332, Cham, 2018. Springer International Publishing.
11. Nicholas Ng and Nobuko Yoshida. Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 174–184, New York, NY, USA, 2016. ACM.
12. James R. Norris. *Markov Chains*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1997.
13. Petr Ročkai, Vladimír Štill, Ivana Černá, and Jiří Barnat. DiVM: Model checking with LLVM and graph memory. *Journal of Systems and Software*, 143:1–13, 2018. https://divine.fi.muni.cz/2017/divm/.
14. Vladimír Štill and Jiří Barnat. Model Checking of C++ Programs Under the x86-TSO Memory Model. In *Formal Methods and Software Engineering*, pages 124–140, Cham, 2018. Springer International Publishing.