Towards LTL Model Checking of Unmodified Thread-Based C & C++ Programs*

J. Barnat, L. Brim, and P. Ročkai**

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{barnat,brim,xrockai}@fi.muni.cz

Abstract. In this paper we present a new approach to verification of multi-threaded C/C++ programs. Our solution effectively chains the parallel and distributed-memory model checker DiVinE with CLang and the LLVM bitcode interpreter. This combination offers full LTL, distributed-memory model checking of virtually unmodified C/C++ source code and is supported by a newly introduced path-reduction technique. We demonstrate the efficiency of the reduction and also the capacity to produce human-readable counter-examples in two small case studies: a C implementation of the Peterson's mutual exclusion protocol and a C++ implementation of a shared-memory, lock-free FIFO data structure designed for fast inter-thread communication.

1 Introduction

Direct applicability of model checking to unmodified, or at most lightly annotated, software systems is extremely desirable, since it substantially reduces costs associated with this otherwise appealing technique. Tools that bypass the modelling step, i.e. those, that model-check software directly, remove the need for a significant part of the specialist work normally required for model checking. This in turn enables wider applicability of automated formal verification.

A number of advancements have been made in this area. One of the first forays into the territory is the support for combining C code with ProMeLa models in SPIN [11], which can be used, although with a number of caveats and substantial amount of extra work, to verify implementation-level properties. Another early approach to the problem is constituted by automated model extraction [10, 12, 6]. In a similar spirit, the ZING [1] model checker is shipped with automated model extraction tools. More direct approaches, which are in many cases also easier to apply, are embodied by model checkers based on a particular programming language (or runtime), like CMC [15], JCat [8], MCP [19], Java PathFinder [20] and MoonWalker [7].

* This work has been partially supported by the Czech Science Foundation grant No. GAP202/11/0312 and by ARTEMIS-IA iFEST project grant No. 100203.

** Petr Ročkai has been partially supported by Red Hat, Inc. and is a holder of Brno PhD Talent financial aid provided by Brno City Municipality

While the CEGAR-based software model checkers like SLAM [2], BLAST [9] or Magic [5] have a related set of goals, they in fact live on a somewhat remote branch in relation to previously mentioned explicit-state model checkers. With real programs, the abstraction refinement process allows for processing of larger systems, but also introduces a degree of infidelity into the process. Therefore, the overall result is more in the spirit of static analysis (with a model checker back end) than it is to exhaustive model checking. Even though the CEGAR approach (and predicate abstraction in general) has been quite successful, it does require a significant insight into the input language of the tool. As a direct consequence, there is no C++ support in CEGAR-based tools. Moreover, verification of multi-threaded programs is usually not supported by CEGAR-based tools, since a suitable symbolic representation for control-based, interleaving parallelism is not available.

With the lack of abstraction in direct software model checking the problem of state space explosion is even more poignant. The most successful techniques used in explicit-state model checking to fight state explosion are Partial Order Reduction [16] and distributed-memory processing, the latter of which has not been yet applied to direct model checking of programs. Hash compaction and bitstate hashing techniques can also be used to help overcome a state space explosion, although at a cost of small infidelity in the model checking process. Hash compaction combined with distributed-memory computation can achieve enormous capacity (on the order of 10^{10} states) in an explicit-state model checker [4]. While symbolic approaches can match or even exceed this capacity, this only applies to special classes of models: unfortunately, software systems with interleaving concurrency (in the imperative style) resist significant symbolic reduction.

Another trait of existing approaches to direct software model checking is their focus on verification of safety properties. However, liveness properties, such as the requirement of guaranteed progress or response, are crucial in specifications of parallel programs. Verification of general liveness properties poses a far greater technical challenge compared to state-space exploration (which is sufficient for an important class of safety properties). Nevertheless, even fully generic safety properties are often neglected. For both generic safety and for liveness, we need a mechanism to conveniently describe atomic propositions, which relate the properties of any given instantaneous state of a system (this problem will be described in more detail in Section 3). Additionally, a property automaton (which refers to these atomic propositions to describe undesired program behaviour) needs to be synchronised with the execution of the program.

In this paper we describe an extension of the model checker DiVinE [3] that enables analysis of models written in LLVM assembly language (in the form of LLVM bitcode). Since compilers that produce LLVM bitcode are available for a variety of programming languages, including C, C++, and Java, this extension allows DiVinE to effectively verify unmodified software. As DiVinE is a parallel and distributed-memory model checker, this is, to our best knowledge, the first approach that allows for verification of unmodified C/C++ programs using distributed memory. Additionally, we introduce a path reduction technique (τ-

reduction) that works at the level of LLVM bitcode. This technique significantly reduces the space explosion effect of the very fine detail available in the model structure arising from the LLVM bitcode. We also describe how general atomic propositions can be defined for C/C++ programs so that verification tasks performed by model checking can go beyond the typically delivered state space exploration that allows for deadlock detection and assertion violation checking.

The rest of paper is organised as follows. In Section 2 we describe the LLVM extension to DiVinE, in Section 3 we elaborate on full LTL model checking of C/C++ programs, in Section 4 we describe the principle of the τ-reduction. In Section 5 we give some related work, in Section 6 we demonstrate the technique on two small use cases, and finally, in Section 7 we give some conclusions and future work.

2 Model Checking LLVM Bitcode

The LLVM system [13] is based on an assembly-level, single-static-assignment language. The main goal of the LLVM framework is to provide support for code transformations using the LLVM assembly language as both the input and output languages (most notably optimisation passes). Moreover, thanks to code generation and just-in-time compilation tools available for the LLVM assembly, LLVM can be transformed into a native program on a number of platforms, either in an ahead-of-time or a just-in-time manner. The combination of these traits makes LLVM a very attractive choice for compiler back ends: a number of optimiser passes are available on the LLVM level, and many code generation choices come basically for free. This fact is illustrated by the widespread support of LLVM-based code generation in various programming language compilers.[1]

The general work-flow for model checking of programs directly from their source code is following: the program source code is compiled into LLVM bitcode, the bitcode is verified with a model checker, and finally the target binary is generated from the verified bitcode. A major advantage of this approach is the ability to apply compiler optimisations before running the model checker. This enables discovery of subtle effects of unspecified behaviour in the presence of various levels of compiler optimisation. Of course, the code generation step still constitutes a possibly unfaithful translation which can introduce property violations unseen by the model checker at the LLVM bitcode level. Nevertheless, this fidelity gap is much narrower here than it is in high-level model checkers and static analysis tools operating on the "statement level" of the input language.

To explore the state space of the program (as described by a LLVM bitcode file), we employ a modified version of the upstream LLVM interpreter. We serialise and store states of the interpreter as byte vectors (one vector corresponding to one state) and each such vector fully describes the configuration of the virtual machine executing the program under consideration. Within a single state, each executing thread is represented by an execution stack, consisting of execution

[1] Including GCC (C, C++, Objective C, Java, Ada and Fortran), Clang (C and C++), GHC (Haskell) and others.

contexts. Each context in turn contains a set of LLVM registers[2], including a program counter. Additionally, the state contains a shared heap for both non-register local variables and for dynamically allocated memory. A thread to be executed is picked non-deterministically in each step, producing all the possible thread interleavings.

Thanks to symbolic debugging interfaces and metadata stored at the bitcode level, it is possible to reliably map assembly locations to original source locations and provide meaningful counterexample traces.

2.1 Preparing programs

While the approach allows for model checking of unmodified programs, there is nevertheless a certain minimal amount of preparation that needs to be done to apply the DiVinE model checker. On the level of C/C++ source code, we provide a header file, `divine-llvm.h` which transparently maps native (pthreads and other supported API) calls to their equivalents required for model checking. In the current version, the header file defines or alters the definition of at least the following functions:

- `malloc(int)` – For the purpose of verification a non-deterministic choice is made whenever `malloc` is called by the program, with a `NULL` return in one of the branches and successful allocation in the other one. This behaviour can be suppressed by a CPP macro `NO_MALLOC_FAILURE`.
- `malloc_guaranteed(int)` – A variant of `malloc` that never fails, even when `NO_MALLOC_FAILURE` is not defined.
- `free(void *)` – Release heap-allocated memory. Deterministic.
- `assert(int)` – Semantically equivalent to `assert` provided in system's `assert.h`; however, the program is not terminated. Any transition that calls a failing assert will raise an "assertion failed" flag in the resulting state. Reachability analysis can be used to search for system states with the flag.
- `trace(...)` – A debugging API, printing a message to the console whenever the corresponding transition is executed by the verification core; implemented as standard error output in non-verification runs.
- `pthread_create(...)` – Create a new thread. Part of the pthread library.
- `arbitrary(n)` – An interface to data-based non-determinism, produces a value between 0 and n. In non-verification builds, a single random value is produced.

The pthread mutex API will be provided in a future revision, most likely in the form of spinlocks. Condition variables can be implemented in terms of a busy wait as well, although this needs to be done carefully to preserve the specified semantics.

The use of `divine-llvm.h` header file is, however, fully transparent. The header does not modify behaviour of the program in any way, unless a CPP

[2] The registers in LLVM are dynamic, and only created upon assignment. Their existence and values are lexically scoped. Allocation of processor registers is part of the native code generation step.

macro `DIVINE` is defined. Therefore, the only difference between a "normal" compilation (producing a natively executable binary) and a "verification" build (for model checking with DiVinE) is that in the latter case, `-DDIVINE` needs to be passed on the command line to the compiler, and native code generation needs to be suppressed. For C and C++ programs, there are two options to generate the requisite LLVM bitcode:

- with **CLang** (which is the preferred option), passing `-emit-llvm` at the command line,
- using **GCC** with **dragonegg**, which will use the GCC compiler front- and middle- ends; the options `-flto -fplugin=/path/to/dragonegg.so` will cause GCC to emit LLVM bitcode.

The library of APIs available to programs under verification is of course subject to future extensions. As of now, the `arbitrary` calls can be used to implement an exhaustive search based on "arbitrary" inputs from the environment, and in turn to provide verification-friendly implementations of system APIs that interact with the environment. In its present form, though, this option is only viable for the smallest of programs: a form of hybrid data-symbolic model checking will be needed to at least partially lift this restriction. Therefore, support for data-intensive forms of system interactions, like reading files or sockets, is not planned until the model checking back end can usefully deal with the consequences.

3 Atomic Propositions and LTL in C/C++ programs

Model checking of general LTL properties is often neglected in favour of safety checking in the form of detection of deadlocks and assertion violations detection.

Of course, LTL model checking of programs per se is not devoid of difficulties: a language of atomic propositions needs to be devised. While it is tempting to simply provide an expression language on top of global variables, such an arrangement is likely to interfere with optimisers (and with store buffer simulation). In particular, the ordering of memory stores at the LLVM bitcode level is not guaranteed to be the same as at the source-code level. Hence, augmenting the source code with new global variables to represent properties of individual states is not satisfactory. Moreover, there is further interference with "instantaneous" occurrences of atomic propositions: often, it is desirable to mark a single state with a certain proposition – in context of programs, this is a very useful ability – allowing the developer to mark, for an LTL formula, that program has passed through a certain location in the source code. Setting and immediate resetting a global variable is not quite a faithful simulation of the instantaneous moment as the two succeeding modifications of the variable may interleave non-deterministically with execution of a parallel thread.

Therefore, we propose (and implement) a system where assert-like statements can be added to the program to encode the atomic propositions: there are two statements, first of which is called simply `ap`, and its only parameter is the identifier of the atomic proposition drawn from an enumerated type in the source language. The other is called `ap_set`, and apart from the atomic proposition

identifier, it takes a value indicating the value to set the atomic proposition to (0/1 in the usual false/true interpretation). The `divine-llvm.h` header provides a no-op implementation for the `ap` procedure, so it can be used in real programs without the need to keep separate versions of the source.

Whenever a call to either of the `ap*` built-in procedures is found in the instruction stream, the atomic proposition values visible to LTL formulae may change. If we were to admit the `ap` mechanism as the only one, the valuation of atomic propositions could only ever change at these points. Of course, the `ap` calls can be, in the source language, guarded by arbitrary conditionals, making (virtually) the complete source language *the* language of atomic propositions. The `ap_set` calls comprise a very "imperative" approach to valuation of atomic propositions, while the `ap` calls are more declarative.

Nevertheless, even though as explained above, values of variables alone are not sufficient to satisfactorily formulate some of the desirable LTL properties, the `ap*` statements alone are not sufficient either: in some cases, the atomic propositions should be derived from variable content, and it is often inconvenient to encode such a relation in explicit `ap` calls. Unfortunately, a solution to this problem is only straightforward when we restrict our interest to global variables – in case of those, we can ask the user to supply a Boolean function which essentially becomes the "implementation" of an atomic proposition. However, not all variables of interest live in the global scope, and we need to deal with atomic propositions that depend on variables that may be currently out of scope. To allow flexible specification of atomic propositions by the users, we propose the following mechanism.

An atomic proposition relies on a function of the source language with Boolean return value and arbitrary parameters. To bind such a function to an actual atomic proposition, we need to match the input parameters of this Boolean function to specific variables in a given scope. This binding will work differently for global and differently for intermittent (lexically scoped) variables. On the global level, a macro `ap_global(proposition, function, N, p1, ..., pN)` is provided. This informs the model checker that whenever the value of the atomic proposition `proposition` is required, it can call the function `function` with values of variables `p1` through `pN` as parameters. Since all these variables must be in the global scope, such a call can be made at any time.

On the other hand, scope-dependent propositions are somewhat more complicated. Whenever local variables may be declared in the program, we allow a macro `ap_local` to be used, similar to `ap_global`. The `ap_local` call may refer to the values of arbitrary in-scope variables, and their **current scope** values will be passed to the evaluation function. This means that when a different scope is entered, even if it shadows some of the parameters to `ap_local`, *the original, shadowed variable values* are used for evaluating the atomic proposition. For example, see Figure 1.

The main problem with these scope-dependent propositions arises in presence of recursive and parallel scopes (lexical scope combined with recursion or with

```
#include "divine-llvm.h"

enum AP { Progress };
LTL(GF(Progress));

bool progress( int x ) { return x == 2; }

void thread( int *x ) {
    int y = 0;
    ap_local( Progress, progress, y );
    while ( true ) {
        y = arbitrary( 3 ); // 0 - 2
        while ( *x == y ); // wait
        y = *x;
    }
}

int main() {
    int x = 0;
    pthread_t tid;

    pthread_create( &tid, NULL, thread, &x );
    while ( true )
        x = (x + 1) % 3;
    return 0;
}
```

Fig. 1. A simple example demonstrating the use of LTL and `ap_local`. Fair scheduling is assumed. The forked thread is expected to continue making progress (i.e. Progress is true infinitely often). This progress may fail to happen if the compiler moves the dereference of `x` out of the inner loop (which it may legally do).

multi-threading gives rise to this case): the same atomic proposition could be assigned different values in different contexts at the same time.

A similar problem arises with the "imperative" `ap_set` construct – in this case, however, it is the responsibility of the developer to manipulate atomic propositions carefully. The `ap_set` call is provided for completeness, and because there are corner cases that are difficult to address otherwise. Whenever possible, it should be avoided in favour of `ap` or of the "declarative" style.

Nevertheless, even the "declarative" approach has certain limits, related to the already mentioned "recursive" scopes. We define the value of a declarative atomic proposition as false whenever it has been bound to none of the currently active scopes. However, it can happen that multiple such scopes are active simultaneously (either "below" each other, in a recursion stack, or "beside" each other in multiple execution threads). In this case, we can no longer relegate this problem to the developer, since they have no explicit control over precedence in such cases. There are multiple candidates for a solution, but the simplest and most intuitive seems to be to set the atomic proposition to 1 (true) when *any*

```
#include "divine-llvm.h"

enum AP { Cap };
LTL(G(Cap));

bool cap( int x ) { return x < 3; }

int rec( int x ) {
    ap_local( Cap, cap, x );
    return x >= 3 ? x : rec( x + 1 );
}

int main() {
    int x = rec( 0 );
    trace( "%d", x );
    return 0;
}
```

rec(0)	rec(1)	rec(2)	rec(3)		result
–	–	–	–		¬Cap
Cap	–	–	–	→	Cap
Cap	Cap	Cap	–	→	Cap
Cap	Cap	Cap	–	→	Cap
Cap	Cap	Cap	¬Cap	→	Cap

Fig. 2. An example of recursive program where atomic proposition `Cap` is defined in multiple (shadowed) scopes with different value.

of the active scopes proscribes this (in other words, we take the logical disjunction of the values given by all active scopes). For example, see Figure 2. This approach is also consistent with the behaviour of the `ap` built-in function.

As a future enhancement, we propose a scheme with (optional) parametric formulae that can use quantifiers over scopes, and a matching (and likewise optional) indexing scheme for atomic propositions. This will allow substantially more flexibility in specification of scope-dependent behaviours in multi-threaded and recursive programs.

4 τ-reduction

The fine-grained nature of LLVM bitcode further aggravates the state-space explosion problem. To counter this, we have devised a very simple yet efficient state-space reduction technique that mimics the path reduction as presented in [21]. The reduction is based on the observation that some transitions in the state space are invisible for other active threads in the system. These, the so-called τ actions, of a thread or process, can be delayed over other τ actions of other processes. In fact, multiple subsequent τ actions of a single process/thread can be safely collapsed into a single transition without any effect on other threads or the observed property.

In traditional model-based model checking of asynchronous systems, this reduction would be quite ineffective, because τ chains are fairly rare in purpose-built models. On the other hand, they are extremely ubiquitous in the SSA bitcode produced by LLVM. While identifying all τ actions is very complex, there is a simple yet very efficient heuristic that will identify and collapse a majority of safe (i.e. not forming a loop) τ transitions. All transitions that:

– do not access memory (only registers), and
– are within a single basic block

can be safely treated as τ transitions. From experience, we know that assembly-level programs, especially in the RISC style with explicit loads and stores (as is the case of LLVM), contain a significant share of instructions (actions) that meet both these criteria.

To illustrate the extreme payoff for the otherwise very simple reduction, let's take our `peterson.c` running example (listing in Figure 3): the state space size (when compiled with `-O2`) before τ-reduction was 37482 states and 107533 transitions and safety verification took 135 seconds. The reduced state space has 5301 states, 14585 transitions and verifies in 18 seconds, which is about 7-fold decrease in state count and 7.5-fold decrease in verification runtime.

It should be noted that this reduction is closely related to "superstep POR" proposed in [22], although simpler. A good candidate for improving both these reductions may be a heuristic that would identify (some of the) memory writes that are (provably) invisible to any other threads. On the other hand, since the LLVM virtual machine has a possibly infinite register file, no register spilling happens (register spilling is normally a major source of invisible memory writes; in LLVM-based compilers, register allocation takes place in a later phase of code generation). This naturally limits the number of invisible writes, and easily explains why the reduction is so successfull despite its simplicity.

5 Related Work

According to [17], the MCP model checker uses a similar approach to model checking of C/C++ program. MCP uses the LLVM interpreter, while it is more closely following the Java PathFinder model, where safety checking is the primary goal. Later papers on MCP [18, 19], however, present a different approach to model checking – through automatic code transformation (annotation). While MCP appears to provide extra flexibility by allowing user-level implementation of the thread scheduling algorithm, we focus on an out-of-the box experience. It is already possible to apply the DiVinE model checker to a subset of unmodified pthread-based programs, with semantics closely resembling those of actual (hardware and operating system) implementations. The major outstanding feature in this regard are store buffers [14], which will bring the model checking process to almost perfect fidelity with real executing code. On the other hand, we believe that the focus of MCP is more on dedicated, mission-critical systems, whereas our primary focus is on commodity hardware and software. However, while we do not rule out the possibility of allowing user override of the scheduler in future releases, we will very likely always provide a default behaviour that mimics real hardware and operating systems.

A different approach to LLVM-based explicit-state model checking is presented in [22] – instead of making it possible to directly model check programs, a hybrid approach is chosen. A program is supplemented with a driver written in the modelling language ProMeLa, which can then call into LLVM-compiled

model variant	assertion	state count	transition count
`-O0`	safe	1992772	5323045
`-O0`, BUG	**unsafe**	1609112	4237118
`-O1`	safe	18631	49624
`-O1`, BUG	**unsafe**	23849	63804
`-O2`	safe	5842	16121
`-O2`, BUG	**unsafe**	12718	35439

Table 1. Verification results for `peterson.c`, as listed in Figure 3. The compiler used was *clang*, version 2.9. All figures are of the entire state space; no early termination was done in this experiment.

functions, with proper interleaving. The approach employs SPIN as the model checking back end, with its respective advantages (maturity, speed) and disadvantages (lack of distributed memory support and limited parallelism). In many cases, this is more laborious, and requires knowledge of another programming language and of the special interface between SPIN and LLVM.

At the time of this writing, though, there was no support for LTL model checking in either of these tools, nor were the tools available publicly. The support for LTL model checking in the non-LLVM software model checkers (Java PathFinder, MoonWalker, and the like) is very limited as well.

6 Use Cases

A very simple, illustrative use case for the software model checking capabilities added to DiVinE is represented by the program listed in Figure 3. We can compile the listed program using the *clang* compiler into LLVM bitcode, either optimised or unoptimised. By running the model checker on the program (τ-reduction enabled), both with the bug indicated in the listing present and corrected, we obtain the results shown in Table 1. We can observe a significant decrease in state space size when compiler optimisations are enabled.

A second, much more elaborate use-case is the verification of a lock-free data structure for inter-thread communication. We do not include the listing of the source code in this paper due to space constraints, but it is available among the examples shipped as part of the development versions of DiVinE.[3] Since the implementation of the data structure as such is not a complete program, we have combined a *unit testing* approach with model checking, providing a small, standalone *unit* test-case (listing shown in Figure 5). Furthermore, since we know that the correctness of the implementation is independent of the actual data payload, we use the common approach of fixing the data values as part of the test-case, making the whole unit test into a closed system. This approach is ubiquitous whenever automated testing of software is performed. However, since

[3] In fact, this same data structure is employed by the parallel model-checking back end of DiVinE for fast shared-memory communication.

```
 1: #include "divine-llvm.h"
 2:
 3: struct state {
 4:     volatile int flag[2];
 5:     int turn;
 6:     volatile int in_critical[2];
 7: };
 8:
 9: struct p {
10:     int id;
11:     pthread_t ptid;
12:     struct state *s;
13: };
14:
15: void thread( struct p *p ) __attribute__((noinline));
16: void thread( struct p *p ) {
17:     p->s->flag[p->id] = 0; // BUG. Should assign 1 here.
18:     p->s->turn = 1 - p->id;
19:     while ( p->s->flag[1 - p->id] == 1 && p->s->turn == 1 - p->id ) ;
20:     p->s->in_critical[p->id] = 1;
21:     trace("Thread %d in critical.", p->id);
22:     assert( !p->s->in_critical[1 - p->id] );
23:     p->s->in_critical[p->id] = 0;
24:     p->s->flag[p->id] = 0;
25: }
26:
27: int main() {
28:     struct state *s = malloc( sizeof( struct state ) );
29:     struct p *one = malloc( sizeof( struct p ) ),
20:              *two = malloc( sizeof( struct p ) );
31:     if (!s || !one || !two)
32:         return 1;
33:
34:     one->s = two->s = s;
35:     one->id = 0;
36:     two->id = 1;
37:
38:     s->flag[0]   = 0;
39:     s->flag[1]   = 0;
40:     s->in_critical[0] = 0;
41:     s->in_critical[1] = 0;
42:     pthread_create( &one->ptid, NULL, thread, one );
43:     pthread_create( &two->ptid, NULL, thread, two );
44:     pthread_join( one->ptid, NULL );
45:     pthread_join( two->ptid, NULL );
46:     return 0;
47: }
```

Fig. 3. An example program implemented in C, using pthreads and shared memory for communication. The program implements Peterson's mutual exclusion.

```
===== Trace from initial =====

[ peterson.c:29 ]

[ peterson.c:42 ]

[ divine-llvm.h:54 ]
[ divine-llvm.h:57 ]
[ divine-llvm.h:53 ]

[ divine-llvm.h:54 ]
[ peterson.c:19, p = *0x800 { 0, 0, *0x400 { [ 0, 0 ], 1, [ 0, 0 ] } } ]
[ divine-llvm.h:53 ]

[ divine-llvm.h:54 ]
[ peterson.c:20, p = *0x800 { 0, 0, *0x400 { [ 0, 0 ], 1, [ 1, 0 ] } } ]
[ divine-llvm.h:53 ]

[ divine-llvm.h:54 ]
[ peterson.c:22, p = *0x800 { 0, 0, *0x400 { [ 0, 0 ], 1, [ 1, 0 ] } } ]
[ peterson.c:17, p = *0x80c { 1, 1, *0x400 <...> } ]

[ divine-llvm.h:54 ]
[ peterson.c:22, p = *0x800 { 0, 0, *0x400 { [ 0, 0 ], 1, [ 1, 0 ] } } ]
[ peterson.c:18, p = *0x80c { 1, 1, *0x400 <...> } ]

[ divine-llvm.h:54 ]
[ peterson.c:22, p = *0x800 { 0, 0, *0x400 { [ 0, 0 ], 0, [ 1, 0 ] } } ]
[ peterson.c:19, p = *0x80c { 1, 1, *0x400 <...> } ]

[ divine-llvm.h:54 ]
[ peterson.c:22, p = *0x800 { 0, 0, *0x400 { [ 0, 0 ], 0, [ 1, 0 ] } } ]
[ peterson.c:20, p = *0x80c { 1, 1, *0x400 <...> } ]

===== The goal =====

[ divine-llvm.h:54 ]
[ peterson.c:22, p = *0x800 { 0, 0, *0x400 { [ 0, 0 ], 0, [ 1, 1 ] } } ]
[ peterson.c:20, p = *0x80c { 1, 1, *0x400 <...> } ]
! ASSERTION FAILED
```

Fig. 4. Counterexample trace produced by DiVinE. The trace has been manually shortened to fit a single page, but is otherwise verbatim. Each block separated by a blank line represents a single configuration of the system, and in each block, each row represents a single thread of execution. Within a single row, the program counter is represented first (whenever possible, through a source file location) and a listing of in-scope variables follows. Type information is used to format values: pointers are automatically dereferenced, structures are shown using braces and arrays using square brackets. Aliased values are elided (shown as `<...>`), unless their occurrences are of different types.

```
LTL( G(Sent -> F(Recv)) )
void threads( Fifo< int > *f ) {
    int id = thread_create();
    if ( id ) {
        for ( int i = 0; i < 3; ++i ) {
            ap( Sent );
            f->push( i );
        }
        while( true );
    } else {
        for ( int i = 0; i < 3; ++i ) {
            int j = f->front( true ); // wait for value
            ap( Recv );
            f->pop();
        }
        assert( f->empty() );
        while ( true );
    }
}
```

Fig. 5. A parallel unit test for a lock-free queue.

our unit test employs multiple threads, the outcome of the test when simply executed is not deterministic. A possible bug in the implementation may go uncovered for many testing iterations, for certain bug classes even thousands or millions. However, by applying our model checker to this multi-threaded unit test, we can guarantee that if *any* thread interleaving violates the property, the problem is found reliably, even though the particular interleaving may be extremely improbable and never found through testing alone.

This hybrid approach is best applied in situations where the control aspect of a parallel program is most important and the input data can be fixed or taken from a small set of specific examples. Moreover, writing test-cases for automated software testing is already part of many software development methodologies and as such familiar to development staff. For these reasons, we believe that this particular combination of unit testing and model checking can become a new and rather useful part of the developer's toolkit when dealing with multi-threaded applications.

7 Future Work and Conclusions

We have presented how LLVM bitcode interpreter can be used in concert with a distributed-memory, explicit-state LTL model checker DiVinE, enabling direct verification of program source code.[4] By using the DiVinE model checker, we

[4] The final implementation will be released as part of DiVinE 3.0. Nevertheless, the work currently in progress is already available in development versions of DiVinE – see `http://divine.fi.muni.cz/development.html`.

are tapping a powerful tool designed for high-performance model checking on platforms ranging from commodity multi-core workstations to high-end compute clusters. The powerful model checking back end is further assisted by a path reduction in the LLVM-based front end.

Moreover, to facilitate LTL model checking of software, we have proposed a novel mechanism to specify atomic propositions, one that is both practical and unobtrusive, while at the same time sufficiently expressive. This makes it possible to realistically specify properties in terms of LTL formulae, as part of real-world programs.

One of the priorities in our future work is to improve support for dynamic memory. Distinguishing states that are identical up to a symmetry with respect to their heap layout is a significant waste of the tool's capacity. Unfortunately, not all languages that the tool supports can provide reliable pointer tagging, which means that the more traditional heap canonisation approaches are not directly applicable. A conservative approach will be required, at least in cases where exact pointer tagging is not available in the given input language.

Additionally, while sequential DFS-based software model checkers can use delta compression to improve their memory efficiency, this is not straightforward with a distributed-memory system. Nevertheless, especially for software, where a single system state is often large, delta compression is usually very efficient and an adequate alternative is needed for use in distributed-memory environment.

Finally, we would like to investigate heuristics for obtaining efficient *ample* sets for LLVM bitcode sources, which would then enable the use of Partial Order Reduction as implemented in DiVinE.

References

1. T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie. Zing: A Model Checker for Concurrent Software. In *Computer Aided Verification*, volume 3114 of *LNCS*, pages 28–32. Springer-Verlag, 2004.
2. T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *IFM*, volume 2999 of *LNCS*, pages 1–20. Springer, 2004.
3. J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC)*, pages 4–7. IEEE, 2010.
4. B. Bingham, J. Bingham, F.M. de Paula, J. Erickson, and M. Singh, G.and Reitblatt. Industrial Strength Distributed Explicit State Model Checking. In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC)*, pages 28–36. IEEE, 2010.
5. S. Chaki, E. Clarke, and A. Groce. Modular verification of software components in C. In *IEEE Transactions on Software Engineering*, pages 385–395, 2003.
6. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. *International Conference on Software Engineering*, 0:439, 2000.

7. N. A. de Brugh, V. Nguyen, and T. Ruys. MoonWalker: Verification of .NET Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *LNCS*, pages 170–173. Springer-Verlag, 2009.
8. C. DeMartini, R. Iosif, and R. Sisto. A Deadlock Detection Tool for Concurrent Java Programs. *Software Practice and Experience*, 29(7):577–603, 1999.
9. T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *Model Checking Software*, volume 2648 of *LNCS*, pages 624–624. Springer-Verlag, 2003.
10. G. J. Holzmann. Logic Verification of ANSI-C Code with SPIN. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 131–147. Springer-Verlag, 2000.
11. G. J. Holzmann. *The SPIN model checker: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.
12. G. J. Holzmann and M. H. Smith. Software Model Checking: Extracting Verification Models from Source Code. *Software Testing, Verification and Reliability*, 11(2):65–79, 2001.
13. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*, Palo Alto, California, Mar 2004.
14. A. Linden and P. Wolper. An Automata-Based Symbolic Approach for Verifying Programs on Relaxed Memory Models. In *SPIN*, pages 212–226, 2010.
15. M. S. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *The Fifth Symposium on Operating Systems Design and Implementation*, 2002.
16. D. Peled. Ten Years of Partial Order Reduction. In *Computer Aided Verification*, pages 17–28. Springer-Verlag, 1998.
17. G. Brat S. Thompson and K. Schimpf. The MCP Model Checker, 2008. Submitted to PEPM 2008.
18. S. Thompson and G. Brat. Verification of C++ Flight Software with the MCP Model Checker. In *Aerospace Conference, 2008 IEEE*, pages 1 –9, March 2008.
19. S. Thompson, G. Brat, and A. Venet. Software model checking of ARINC-653 flight code with MCP. In César Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 171–181, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.
20. W. Visser, K. Havelund, G. P. Brat, and S. Park. Model Checking Programs. In *ASE*, pages 3–12, 2000.
21. K. Yorav and O. Grumberg. Static Analysis for State-Space Reductions Preserving Temporal Logics. *Formal Methods in System Design*, 25(1):67–96, 2004.
22. A. Zaks and R. Joshi. Verifying Multi-threaded C Programs with SPIN. In K. Havelund, R. Majumdar, and J. Palsberg, editors, *Model Checking Software*, volume 5156 of *Lecture Notes in Computer Science*, chapter 22, pages 325–342–342. Springer-Verlag, 2008.