The MCP Model Checker

Sarah Thompson[1], Guillaume Brat[1], and Karl Schimpf[2]

[1] NASA Ames Research Center, MS-269/2, Moffett Field, CA 94035-1000, USA
{thompson, brat}@email.arc.nasa.gov
[2] Google Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043, USA
karlschimpf@gmail.com

Abstract. MCP is an explicit-state software model checker that supports the entire C++ programming language. In this paper, we describe its architecture and present some initial results.

1 Introduction

The MCP[3] model checker was constructed specifically to allow programs written in C or C++ to be model-checked directly without requiring prior translation or model extraction. It builds upon the LLVM[4] compiler infrastructure [1], thereby avoiding the requirement to directly recognise the (extremely complex) C++ language at source-level. This approach has allowed us to support the entire C++ language [2], including templates. The C language [3] is handled separately and fully, not just as an improper subset of C++.

1.1 Model Checking at NASA

NASA has been involved with research on software model checking for some time. Our group at NASA Ames developed Java Pathfinder (JPF) [4], a publicly available software model checker for the Java language, and also LTSA, a model checker for state machines that supports compositional verification [5, 6]. Our colleagues at the NASA Jet Propulsion Laboratory developed the well-known SPIN model checker [7].

Experience from the JPF project has demonstrated the utility of *software model checking* (i.e. model checking that acts directly on a program, rather than on a model that has been manually extracted from it). However, current flight software is mostly implemented in C, not Java, and in future it seems increasingly likely that C++ will become the platform of choice. The MCP model checker was developed to fill the requirement for an explicit-state software model checker, in the style of JPF, that fully supports the C++ language.

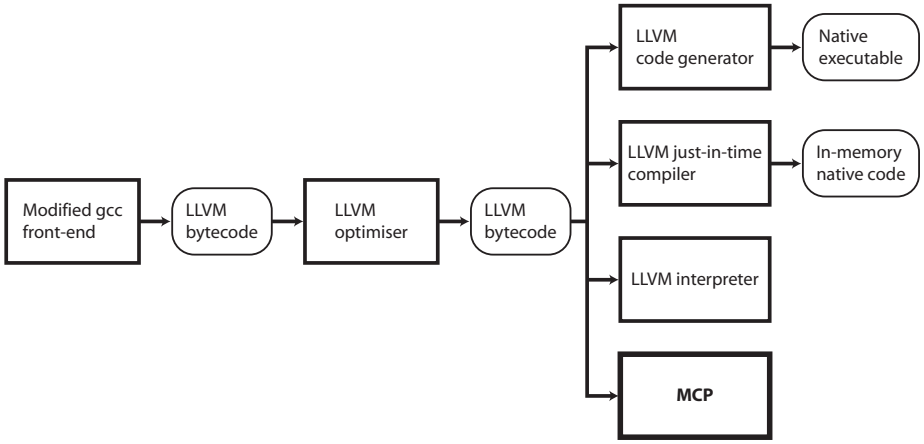

Fig. 1. Simplified LLVM architecture

1.2 LLVM

Fig. 1 shows a simplified version of the LLVM flow[5]. A modified version of the `gcc` front-end is used to parse the C++ source code and to lower most of the language's constructs to a level closer to that of a typical C program. The original `gcc` back-end is discarded in favour of emitting *LLVM bytecode,* which is then optimised and passed on to various alternative back-ends.

One of the main motivating ideas behind LLVM was enabling whole program optimisation, something that has been traditionally difficult to approach with existing compiler architectures. The LLVM bytecode has been specifically designed to support this and other kinds of optimisation – a Static Single Assignment (SSA) representation [8] is adopted, making many analyses (including ours) far more straightforward than they might otherwise be.

2 MCP Architecture

MCP is, in effect, a heavily modified, extended interpreter for LLVM bytecode that supports threads, state abstraction and a variety of search strategies. No attempt is made to extract a model from the code; rather, the code is executed directly, trying all possible interleavings, with the intention of triggering otherwise-deeply-hidden assertions.

Fig. 2 gives an outline of MCP's internal structure. A bytecode interpreter, closely based on LLVM's original `lli` tool, has the responsibility of executing

[3] Model checker for C Plus plus.
[4] Low Level Virtual Machine
[5] Many LLVM tools have been omitted here for clarity – LLVM is a large, rich toolset, so we concentrate on the subsystems that are specifically relevant to MCP.

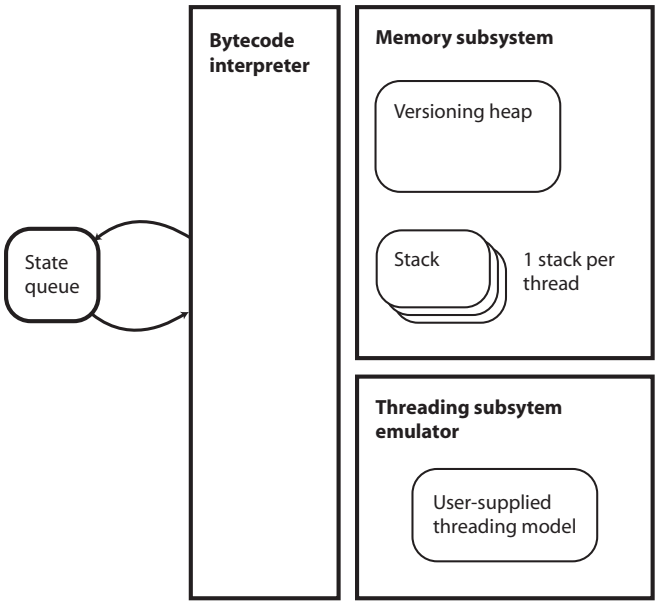

Fig. 2. MCP architecture

LLVM instructions. All memory access is mediated through a versioning memory management subsystem that allows the system to be checkpointed at any time, and for the system state to be rolled efficiently to any other saved checkpoint. Threading is implemented in such a way as to allow user-supplied threading subsystems to be loaded, thereby enabling the emulation of the majority of operating system threading models. A *state queue* maintains the list of states that have yet to be evaluated; manipulating the order in which states are added and removed from this queue allows a variety of search strategies to be implemented.

2.1 Threading Subsystem Emulation

One of the trickier issues surrounding the model checking of C++ source code is the fact that the language standard specifically does not mandate any particular threading model. Real-time program semantics are therefore dependent upon the execution environment, so any attempt to analyse multithreaded code must inevitably make some kind of assumption about the underlying threading model.

At an early stage of the development of MCP it was unclear what kinds of operating system platform might need to be supported, so it was not possible, for example, to assume that any existing threading model (*e.g.,* POSIX threads [9]) could act as a suitable infrastructure. Any implicit assumptions made about threading have profound implications for the real-time semantics of programs, so using one existing threading model to emulate another was thought to be too

dangerous. Furthermore, this approach may omit certain areas of functionality that emulating new operating systems might require.

The approach adopted within MCP is very simple, though entirely generic. At all times, a list of executing threads is maintained. MCP's bytecode interpreter inner loop executes code fragments until such time as a non-thread-specific memory access occurs – at this point, the state is checkpointed and the list of executing threads is reexamined in order to identify which, if any, threads may validly continue to execute. Entries are then placed in the state queue for each executable thread that has been identified – when they are later evaluated, the saved checkpoint is retrieved, and the thread is allowed to proceed for one step. The inner loop continues until either an assertion is detected, or the state queue is exhausted.

At first sight, this may not appear sufficient to allow the implementation of any threading model. However, all threading models and all synchronisation primitives, including real-time concepts such as interrupt service routines, ultimately can be viewed simply as ways of choosing when threads may be preempted and which thread may execute next.

MCP implements its own low-level API, on top of which arbitrary threading models may be constructed. Threads are initiated either by calling `mcp_begin_thread()`, which spawns a thread based upon executing a specific function, or by calling `mcp_fork()`, which makes a copy of the current thread modified in such a way that the child thread perceives the function's return value to be *true* and the parent thread perceives it to be *false.* Threads are terminated either by simply returning from the function or by explicitly calling `mcp_terminate_thread()` or `mcp_kill_thread()` which respectively terminate the current thread or forcibly terminate another thread given its handle. Threading packages must supply a *comparison function,* which given a pair of thread handles a and b returns true iff thread a can preempt thread b. The special thread handle `MCP_IDLE` represents a notional idle thread – any thread that can not preempt `MCP_IDLE` is (by definition) blocked. The comparison function has the responsibility of checking any internal thread-specific synchronisation mechanisms that might be required – MCP itself deliberately does not directly implement thread synchronisation. MCP uses this function to perform a sort on the list of currently existing threads, then it walks down from the highest priority thread until it finds the first thread that can no longer execute, thereby identifying all currently executable threads. Since this is the only scheduling information needed by MCP's inner loop, no other infrastructure is necessary within MCP itself.

Since thread comparison functions are called after every execution step, for performance reasons, the LLVM just-in-time compiler mechanism is used to generate a native-code version of the comparison function that is then called on-the-fly by the sort algorithm. This approach in early testing showed itself to be orders of magnitude faster than executing the comparison function in the same context as the program being model-checked – something that is particularly important given that the thread list must be re-sorted after every execution step.

2.2 Memory Management

The MCP memory management subsystem was specifically intended to allow arbitrary search strategies to be adopted independently from its implementation. As a consequence of this requirement, the following design criteria were adopted:

1. No order of evaluation should be assumed; it should be possible to load any saved memory state at any time.
2. All states must be abstracted, in the sense that a statistically-likely-to-be-unique hash is calculated.
3. For efficiency reasons, any algorithms related to the state tree should, where feasible, operate in time and space linear to the amount of difference between states.

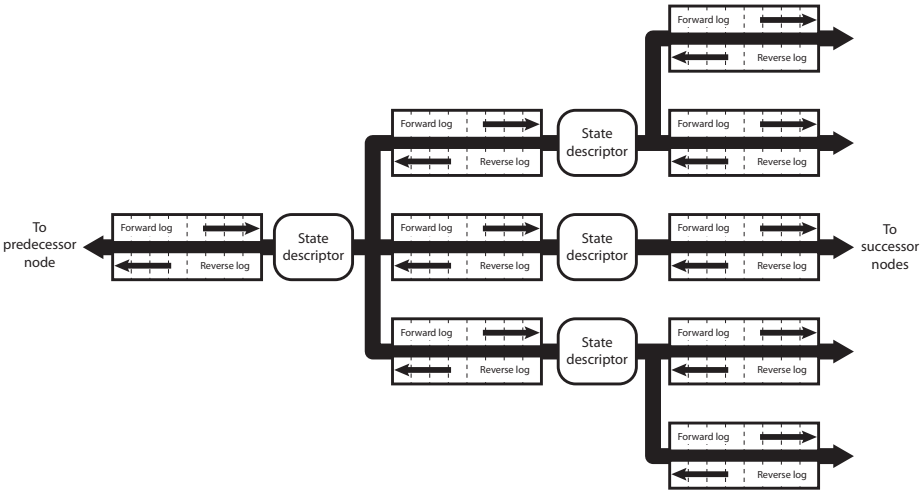

Fig. 3. State Tree

State Deltas. Fig. 3 illustrates the approach taken. States are represented by unique *state descriptors,* each of which having a statistically unique hash. Any successor states are referenced by the predecessor state and vice-versa, allowing the state tree to be navigated efficiently.

When a state is checkpointed, a new state descriptor is created and linked to its parent state. At all times, the memory management system maintains an incremental hash that is updated whenever memory is allocated, freed or modified, so at the time a checkpoint is generated, this simply needs to be copied into the state descriptor's data structure. As a consequence, checkpointing has very little overhead.

As execution proceeds, two *logs* are accumulated associated with the current state descriptor. The *forward log* stores changes to memory that occur as a consequence of program execution, whilst the *reverse log* saves previous values in such a way as to allow the changes recorded in the forward log to be undone. It is therefore possible to reload any saved state by traversing the state tree along well-defined edges and replaying the relevant logs. As an implementation convenience, the forward log may also contain information about visited instructions, visited source lines and user-supplied textual logging information – this extra information is used when reporting counterexample traces in order to make them more human-readable.

Saved states may be reloaded at any time by the following algorithm:

1. Initialise pointers c and d to the current and destination states in the state tree.
2. Working toward the root of the tree from c and d, find a corner node a representing the deepest common ancestor of c and d.
3. Starting at state c and traversing up the tree, replay all reverse logs until the state has been regressed to that represented by the corner node a.
4. Working down the tree from a toward the destination state d, replay the forward logs in sequence until the destination state is reached.

As well as the contents of the heap, saved states also include the current stacks of executing and paused threads, as well as the current values of any temporary variables used by the bytecode interpreter. Reloading a saved state therefore leaves the interpreter immediately ready to continue execution at the relevant point – states can therefore be evaluated in any order convenient to the chosen search strategy without affecting the semantics of the program that is being model-checked.

2.3 Search Strategies

As described in Section 3, states can be evaluated in any order, so there is no implementation-specific requirement to restrict order of evaluation in any way.

The interpreter's inner evaluation loop operates as follows:

1. Take the next state to be evaluated from the state queue.
2. Reload that state.
3. Evaluate instructions until a scheduling point is reached (normally a read or write to non-thread-local memory, though certain MCP API calls may also trigger scheduling).
4. Checkpoint the current state.
5. If the state has been encountered previously, ignore it and repeat immediately.
6. Add the state to the state tree.
7. Use the user-supplied threading subsystem to determine the (possibly empty) list of successor states.

8. Insert those states into the state queue according to the chosen search strategy.
9. Repeat until the state queue is empty.

With this approach, search strategies may be implemented very straightforwardly, simply by affecting how and in what order states are added to the state queue.

Breadth-first. Breadth-first search, MCP's default, is implemented by adding new states to the back of the search queue in arbitrary order. As the search proceeds, depth is therefore kept as shallow as possible.

Depth-first. The MCP state queue is double-ended; depth-first search is implemented by adding states to the front, rather than the back, of the queue, with the state representing the next action of the current thread added last, causing it to be executed next. Depth-first search is invoked by including the `-depth-first` switch in the MCP command line.

Heuristic. In this mode, invoked by the `-heuristic` command-line switch, MCP's normal state queue is replaced by a priority queue that returns states in descending order of a program-specified metric. The search is still complete, in the sense that unless terminated early manually or by the discovery of a counter-example, all states are still visited eventually, but the program under test can influence the order of evaluation.

To use this feature, a program must nominate a variable of type `volatile int` to hold the metric – whenever a state is added to the priority queue, this value is read automatically and used to prioritise any successor states. For example:

```
volatile int metric;

int main()
{
    mcp_register_heuristic(&metric);

    // ... Code under test can modify metric at any time ...

    return 0;
}
```

This approach has the advantage that no per-execution-step overhead is incurred in evaluating the metric if it does not require updating. In use, if the `-heuristic` switch is not given, MCP ignores calls to `mcp_register_heuristic()`, making it possible to easily switch between search strategies without needing to recompile the code under test.

Randomised. Randomised search, selected by the `-random` switch, is broadly similar to breadth-first search, with the exception that next states are inserted at randomly chosen positions within the state queue. As with the other search strategies, randomised search is complete, and states are never revisited – despite the superficial similarity, this search strategy should not be regarded as equivalent to conventional randomised testing.

In practice, randomised search has a 'look and feel' somewhere between that of breadth- and depth-first search, but in practice it appears to have far less tendency than either to get stuck in local minima within the search space.

Interpret-only. For convenience, the `-interpret-only` switch causes MCP to execute the program under test purely as an interpreter – no search *per se* is performed. Since programs annotated with calls to the MCP API can not easily be executed natively without modification, this mode provides a convenient means of executing programs conventionally within the MCP environment.

Programs executing in interpret-only mode are scheduled by a simple round-robin scheduler executing on top of any threading emulation package that may be in use. As usual, however, all instructions and visited source lines are logged, and a counterexample trace is still generated if the program under test triggers it.

2.4 C++ Language Issues

One of the main difficulties that exists in analysing the C or C++ programming languages is the absence of a threading model in the language standard. As a consequence of this, there can be no generically applicable approach to identifying opportunities for partial order reduction statically. Similarly, compiler optimisations that are entirely safe for single-threaded code are not guaranteed to be appropriate for multithreaded code.

LLVM Optimisations. LLVM's approach of carrying out most of its optimistions at the level of an internal representation based on SSA (Static Single Assignment) form means that in its generated bytecode, there is a clear delineation separating operations that would ideally be carried out (on a CPU with sufficient resources) on a register-to-register basis, from those that are required to act upon the heap.

Some of LLVM's optimisations are actually a little too aggressive for our purposes – small blocks of memory allocated by calls to `malloc()` can in certain circumstances be lowered to a form where the data is handled entirely at the register-to-register level, allowing the call to `malloc()` to be omitted, along with much expensive pointer dereferencing. For single-threaded code, such optimisations do not affect the program's observable functionality, but for multithreaded code there is a problem with this approach: all threads share a single heap, but since stacks are thread-local, all other data is not shared. Any program written with an assumption that a particular heap address is shared by multiple threads

```
#include <mcp.h>          // Include file containing definitions for the mcp API.

int main()
{
        volatile int *xxx = (volatile int *) malloc(sizeof(int));
        *xxx = 123;
                          // Declare a volatile integer with storage on the heap
                          // and initialise it to 123.

        bool f = mcp_fork(1);                 // Fork a thread. The child thread sees f = true,
                                              // whereas the parent thread sees f = false.
        if(f)
        {
                (*xxx) += 321;                // The child thread adds 321 then terminates.
                return 0;
        }

        mcp_printf("xxx = %d\n", *xxx);       // Print what we see in the variable.

        mcp_assert(*xxx == 444);              // Assert if we don't get the 'expected' value, 444.

        return 0;
}
```

Fig. 4. Trivial race condition

runs the risk that this kind of optimisation strategy can break the program's semantics.

The C and C++ language standards both allow for variables to be marked with the `volatile` storage class. Older compilers commonly ignored this attribute, though this was rarely a problem since their relatively primitive optimisers didn't perform dangerous modifcations of multithreaded semantics. As a consequence of this, many programmers never learned to use `volatile` correctly, so there is a large amount of potentially unsafe legacy code in existence. LLVM does take the `volatile` keyword seriously, generating quite different code when it is used in comparison with the default case. Without the keyword, shared memory locations are assumed to be thread-local, so they typically get optimised away aggressively.

MCP's Approach. Though performing static analysis on code in order to identify opportunities for partial order reduction is a tempting goal, when C or C++ is the source language, this is actually fraught with difficulties. As mentioned previously (see Section 2.4), any such analysis requires knowledge of what exactly constitutes a thread; something that can not be extracted from the code as defined in the language standard alone. If it had been our intention to base MCP upon a specific threading model, it may have been feasible to go beyond this limitation, but our requirement to support many possible kernel threading models has precluded this.

LLVM, by default, already tends to optimise code such that accesses to heap (and therefore, accesses to memory that might possibly be shared between

threads) is kept to a minimum. Improving on this would essentially require an alias analysis that is thread-sensitive, but since we can't guarantee to be able to identify threads, we can not safely take this approach. We therefore schedule on heap accesses and do not attempt further analysis – in practice, LLVM's aggressive optimisation means that we lose relatively little by this, whilst having the advantage of not risking unexpectedly incorrect semantics as a consequence of failing to identify a thread statically.

3 Experimental Results

At the time of writing, MCP is at a relatively early stage of development, so no large case studies have yet been attempted. For completeness, we include here two small examples from MCP's regression test suite.

3.1 Simple Race Condition

This is perhaps the simplest possible test case that still exhibits concurrent behaviour. The annotated code is shown in Fig. 4.

Storage for a variable of type `int` is created on the heap, then initialised. A child thread is spawned which increments the variable and then terminates immediately. The MCP-built-in `mcp_printf` function behaves similarly to the C/C++ standard library function `printf`, with the exception that its output is logged along with source line and instruction information, making it considerably easier to decode counterexample traces.

Compiling and running the code in interpreted mode is straightforward:

```
% llvm-gcc simplerace.cpp -o simplerace
% mcp -interpret-only -stats simplerace.bc
MCP V0.1 (Sarah Thompson, USRA-RIACS/NASA Ames)
|0>|0>
*** Forked thread [1]
|1>
*** Thread 1 terminated.
|0>xxx = 444
===-------------------------------------------------------------------------===
                          ... Statistics Collected ...
===-------------------------------------------------------------------------===

  3 mcp - Number of context switches
 31 mcp - Number of dynamic instructions executed
  1 mcp - Number of created threads
 10 mcp - Number of global vars initialized
167 mcp - Number of bytes of global vars initialized
```

Since MCP operates on LLVM bytecode, it is necessary to specify the `.bc` extension on the file name. By default, the output is moderately verbose – specifying the `-quiet` command-line switch reduces this to a minimum. In this case, simply interpreting the code with the default round-robin scheduler does not encounter the race condition, so `xxx` always takes the value 444.

Running MCP in its default breadth-first search mode is more fruitful (the output shown here is abridged slightly):

```
% mcp -stats simplerace.bc
MCP V0.1 (Sarah Thompson, USRA-RIACS/NASA Ames)
```

```
|0>|0:0>|0.0:0>*** Forked thread [1]
|0.0.0:1>|0.0.0:0>|0.0.0.1:0>|0.0.0.1:1>|0.0.0.0:1>|0.0.0.0:0>
xxx = 123
|0.0.0.1.0:0>xxx = 123
|0.0.0.1.0:1>|0.0.0.1.1:1>
*** Thread 1 terminated.
|0.0.0.1.1:0>|0.0.0.0.1:0>xxx = 123
|0.0.0.0.1:1>|0.0.0.0.0:1>|0.0.0.0.0:0>
===-------------------------------------------------------------------------===
                          ... Statistics Collected ...
===-------------------------------------------------------------------------===

 14 mcp - Maximum Queue Depth
 16 mcp - Number of states visited
 57 mcp - Number of dynamic instructions executed
  1 mcp - Number of created threads
 10 mcp - Number of global vars initialized
167 mcp - Number of bytes of global vars initialized

### Assertion failure detected: simplerace.cpp, 21: Expression '*xxx == 444' returned false.
=== BEGIN STACK DUMP ===
--- Thread 0 (Running) ---
[mcp_break: BB=0x00000000, CS=0x00000000, IP=0x00000000, StkFrm=0x08996120]
[main: BB=0x0898d510, CS=0x0898d570, IP=0x0898ddd8, StkFrm=0x089960b8]
   Caller:  tail call void %mcp_break( sbyte* getelementptr ([41 x sbyte]* %.str_6, int 0, int 0),
       sbyte* getelementptr ([15 x sbyte]* %.str_3, int 0, int 0), int 21 )
--- Thread 1 (Running) ---
[main: BB=0x0898cd08, CS=0x00000000, IP=0x0898d270, StkFrm=0x0899e998]
=== END STACK DUMP ===
### Decision trace:
Root: |0> (1 successor states)
thread: |0.0> (1
successor states)
   simplerace.cpp(7): {
   simplerace.cpp(8):     volatile int *xxx = (volatile int *) malloc(sizeof(int));
      %tmp.4 = malloc int     ; <int*> [#uses=5]
   simplerace.cpp(9):     *xxx = 123;
      volatile store int 123, int* %tmp.4
thread: |0.0.0> (2 successor states)
   simplerace.cpp(11):    bool f = mcp_fork(1);
      %tmp.7 = tail call bool %mcp_fork( ulong 1 )        ; <bool> [#uses=1]
thread: |0.0.0.0> (2 successor states)
   simplerace.cpp(13):    if(f)
      br bool %tmp.7, label %then.0, label %endif.0
   simplerace.cpp(19):    mcp_printf("xxx = %d\n", *xxx);
      %tmp.16 = volatile load int* %tmp.4     ; <int> [#uses=1]
thread: |0.0.0.0.0> (2 successor states)
      %tmp.13 = tail call int (sbyte*, ...)* %mcp_printf( sbyte*
             getelementptr ([10 x sbyte]* %.str_5, int 0, int 0), int %tmp.16 )
      -- xxx = 123
   simplerace.cpp(21):    mcp_assert(*xxx == 444);
      %tmp.18 = volatile load int* %tmp.4     ; <int> [#uses=1]
thread: |0.0.0.0.0.0> (0 successor states)
      %tmp.19 = seteq int %tmp.18, 444        ; <bool> [#uses=1]
   simplerace.cpp(21):    mcp_assert(*xxx == 444);
      br bool %tmp.19, label %endif.1, label %then.1
      tail call void %mcp_break( sbyte* getelementptr ([41 x sbyte]* %.str_6, int 0, int 0),
         sbyte* getelementptr ([15 x sbyte]* %.str_3, int 0, int 0), int 21 )
```

This time, MCP's search encounters the assertion failure and generates a counterexample trace showing the exact sequence of encountered source lines and executed instructions that led up to the assertion. In this case, though the model checker did attempt interleavings where the child thread executed, as expected, a counterexample was given for a case where the child thread failed to execute before the assertion was evaluated.

Depth-first search, as expected, finds a similar (though not identical) trace after visiting only 7 states and after executing 26 instructions. Randomised search also finds a valid counterexample trace, with performance somewhere between the depth- and best-first results: 11 visited states and 41 executed instructions. Run time in all cases was too rapid to accurately measure.

3.2 Dining Philosophers Algorithm

The classic *dining philosophers* algorithm [10] is depicted in Fig. 5. In this thought experiment, n philosophers are seated at a round dining table with

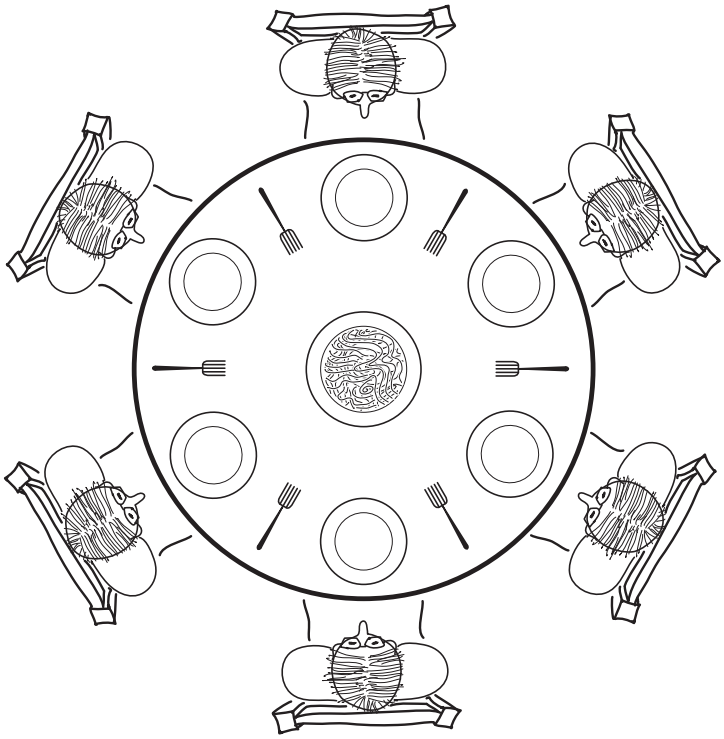

Fig. 5. Dining Philosophers

a large bowl of spaghetti in the centre. Since spaghetti is rather slippery, it is essential to use two forks in order to transfer it to one's plate. The philosophers must therefore pick up the forks on both sides of their plates, help themselves to some spaghetti, eat it, and then return the forks to the table. A typical algorithm, from the point of view of one of the philosophers, is as follows:

1. Pick up left fork.
2. Pick up right fork.
3. Serve spaghetti.
4. Eat spaghetti.
5. Put down right fork.
6. Put down left fork.
7. Repeat until full of pasta.

Of course, sensible philosophers would collaboratively arbitrate access to the forks. However, our particular philosophers have a tendency to always blindly follow rules, so there is a potential problem – if all of the philosophers pick up (say) their left-hand-side fork at the same time, none of them are then able

to pick up their right hand fork. Sadly, this deadlock condition causes them to collectively starve to death.

An extremely simplistic implementation coded directly for the MCP API is as follows:

```
#include <mcp.h>

const int n = 6;

volatile int total_picked_up = 0;
volatile int total_waiting = 0;

class Fork
{
public:
    volatile bool picked_up;
    int number;

    void pick_up(int by)
    {
        mcp_begin();
        if(picked_up == true)
        {
            total_waiting++;

            mcp_printf("%d philosophers are currently waiting.\n", total_waiting);

            // If all threads are waiting, we have a deadlock.
            mcp_assert(total_waiting < n);

            while(picked_up == true)
            {
                mcp_end();
                mcp_printf("Philosopher #%d is waiting for fork #%d\n", by, number);
                mcp_begin();
            }

            total_waiting--;
            mcp_printf("%d philosophers are currently waiting.\n", total_waiting);
        }

        mcp_printf("Fork #%d picked up by philosopher %d\n", number, by);

        picked_up = true;
        total_picked_up++;
        mcp_end();
    }

    void put_down(int by)
    {
        mcp_begin();
        mcp_printf("Fork #%d put down by philosopher %d\n", number, by);
        picked_up = false;
        total_picked_up--;
        mcp_end();
    }
};


class Philosopher
{
public:
    int id;
    Fork *left_fork, *right_fork;

    void thread()
    {
        while(true)
        {
            left_fork->pick_up(id);
            right_fork->pick_up(id);
            eat();
            right_fork->put_down(id);
            left_fork->put_down(id);
        }
    }

    void eat()
    {
        mcp_printf("Philosopher #%d eats.\n", id);
    }
};

int main()
{
    Fork forks[n];
    Philosopher philosophers[n];
```

```
    int i;

    for(i=0; i<n; i++)
    {
        forks[i].number = i;
        forks[i].picked_up = false;

        philosophers[i].id = i;
        philosophers[i].left_fork = forks + i;
        philosophers[i].right_fork = forks + ((i+1) % n);
    }

    for(i=0; i<n; i++)
    {
        if(mcp_fork(i))
        {
            philosophers[i].thread();
            return 0;
        }
    }

    return 0;
}
```

In the interests of simplicity, rather than providing a threading model that supports thread synchronisation, this example uses the `mcp_begin()...mcp_end()` mechanism, which has the effect of making the enclosed code behave atomically. Philosophers wait for forks to become available by spinning on the fork's `picked_up` member variable. This implementation approach (deliberately) introduces the potential problem that an unfair scheduler could spin forever without ever giving other threads time to execute.

On this algorithm, breadth-first search typically has an unfair advantage, in that the the case where all threads take the same first step is generally arrived at relatively early in the search. In the case of our example, since the threads are started one at a time, proceeding asynchronously, MCP's breadth-first search doesn't do as well as we might have hoped because the path that might have led to finding the deadlock is already well buried in the search queue before the final thread begins. Depth-first search tends to get stuck in the loop that spins on the value of `picked_up`, as expected, and fails to find the deadlock. Randomised search does much better, however, solving the problem for $n = 6$ philosophers in less than a second[6], visiting just 237 states. The counterexample trace generated by MCP in this case may be summarised as follows:

1. Fork #0 picked up by philosopher 0.
2. Fork #1 picked up by philosopher 1.
3. Fork #2 picked up by philosopher 2.
4. Fork #3 picked up by philosopher 3.
5. 1 philosophers are currently waiting.
6. Philosopher #2 is waiting for fork #3.
7. Fork #5 picked up by philosopher 5.
8. Fork #4 picked up by philosopher 4.
9. 2 philosophers are currently waiting.
10. 3 philosophers are currently waiting.
11. Philosopher #0 is waiting for fork #1.

[6] Timing based on executing MCP on Red Hat Enterprise Linux running under VMWare on a Windows-based laptop

12. 4 philosophers are currently waiting.
13. Philosopher #2 is waiting for fork #3.
14. Philosopher #3 is waiting for fork #4.
15. 5 philosophers are currently waiting.
16. 6 philosophers are currently waiting. *(Deadlock)*

At the point the fault was detected, the search tree was 42 layers deep, not including data structure creation or thread-spawning.

4 Related Work

Structurally, MCP probably bears closest resemblance to JPF (Java Pathfinder) [4], though at the time of writing it does not approach JPF's maturity. The most significant differences between JPF and MCP stem from the differences between Java and C++; for example, JPF takes advantage of reflection and the standard threading package in Java, which MCP can not since those features are not present in C or C++. The only other model checker directly addressing C++ is Verisoft [11], which takes a completely different approach. Verisoft follows a stateless approach to model checking while MCP follows a conventional explicit-state model similar to SPIN [7]. Though MCP's memory model bears some resemblance to that of SPIN, our approach of executing bytecode in an extended interpreted environment differs substantially from SPIN's approach of compiling high-level models into C that are then compiled and executed. Currently, we use LLVM's just-in-time compiler functionality to compile user-supplied threading models to machine code on-the-fly, though to-date this is not applied to the program under test since the interpreter is relatively inexpensive in comparison with memory management overheads.

There are, however, several model checkers that address C. SLAM [12] is really more of a static analyser than a model checker. It relies heavily on abstractions, starting from a highly abstracted form and building up to a form that allows a complete analysis. CMC [13] uses an explicit-state approach, but it requires some manual adaptation when dealing with complex types (*pickle* and *unpickle* functions). Finally, there have been some attempts within NASA to use the Valgrind tool [14, 15] as a model checker. Unfortunately, it implies using very crude steps between transitions.

Other approaches to model checking code involve a translation step, be it automatic or manual. For example, Bandera [16] provides model checking of Java programs by translating automatically the program into a PVS [17], Promela [18] or SMV [19] model.

5 Conclusions

This paper provides a 'first-look' at the new MCP model checker. At the time of writing, we are about to start applying the tool to verification problems within NASA, supporting the NGATS (Next-Generation Air Transportation System) and Ares/Orion (Shuttle-replacement) programmes.

6 Future Work

This paper describes MCP at a relatively early stage of development. Though, largely thanks to inheriting LLVM's interpreter infrastructure, it does already support the entire C++ language, more work needs to be done in order to make it fully comparable with existing, more mature model checkers aimed at other languages.

6.1 State Delta Compression

In the current version, state deltas reflect all changes made to memory between checkpoints. Whilst this information is sometimes invaluable (it can be dumped alongside instruction traces as a command-line option), there are significant potential memory savings that could be achieved by compressing this information.

At a simplistic level, if a memory location is rewritten several times in sequence, or if two altered blocks overlap, these updates will normally occupy separate log entries[7]. Coalescing overlapping log entries into single entries is an obvious optimisation that could save significant amounts of memory.

If further memory savings are necessary, it should be possible to introduce code that compresses logs using one of the well-known algorithms such as LZW [20].

Ideally log compression would be implemented as a background thread, thereby removing compression overhead from the critical path on multicore and shared-memory multiprocessor systems.

6.2 User-Specified State Abstraction

Currently state abstraction operates purely at the level of state hashing, which happens behind-the-scenes automatically. It is our intention to extend the model checker's API to allow blocks of memory to be excluded from hashing, and to allow user-supplied hook functions to take over the responsibility of hashing particular areas of memory. This should provide a generic means of supporting arbitrary state abstraction strategies, whilst retaining the existing tool's ease of use for the default case.

6.3 Standard Threading Models

We intend to support a number of standard threading models, based on a variety of real-time operating systems that are of interest to NASA. First to appear is likely to be support for the Green Hills Integrity-178B real-time operating system [21], possibly followed by VxWorks [22]. Support for POSIX threads [9] is already in development.

[7] Overlapping updates that arise as part of the evaluation of a single instruction *are* always coalesced, however.

References

1. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California (Mar 2004)
2. Stroustrup, B.: The C++ Programming Language. Third edn. Addison-Wesley Professional (June 1997)
3. Kernighan, B.W., Ritchie, D., Ritchie, D.M.: C Programming Language (2nd Edition). Prentice Hall PTR (March 1988)
4. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer (STTT) **V2**(4) (March 2000) 366–381
5. Magee, J., Kramer, J.: Concurrency: state models & Java programs. John Wiley & Sons, Inc., New York, NY, USA (1999)
6. Jeff Magee, Jeff Kramer, R.C., Uchitel, S.: LTSA: Labeled Transition System Analyser http://www.doc.ic.ac.uk/ltsa/.
7. Holzmann, G.: The SPIN model checker: primer and reference manual. Addison-Wesley (September 2003)
8. Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, F.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems **13**(4) (1991) 451–490
9. IEEE: IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language). (1995)
10. Dijkstra, E.W.: Hierarchical ordering of sequential processes. In: Operating Systems Techniques. (1971) 72–93
11. Godefroid, P.: Model checking for programming languages using Verisoft. In: Symposium on Principles of Programming Languages. (1997) 174–186
12. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S., Ustuner, A.: Thorough static analysis of device drivers (2006)
13. Musuvathi, M., Chou, A., Dill, D.L., Engler, D.: Model checking system software with CMC. In: EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop: beyond the PC, New York, NY, USA, ACM Press (2002) 219–222
14. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proceedings of PLDI 2007. (June 2007)
15. Seward, J., Nethercote, N.: Using Valgrind to detect undefined value errors with bit-precision. In: Proceedings of the USENIX'05 Annual Technical Conference. (April 2005)
16. Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, Laubach, S., Zheng, H.: Bandera: Extracting finite-state models from Java source code. In: Proceedings of the 22nd International Conference on Software Engineering. (June 2000)
17. Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M.K.: PVS: Combining specification, proof checking, and model checking. In: CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification, London, UK, Springer-Verlag (1996) 411–414
18. SPIN web site: Promela language reference http://spinroot.com/spin/Man/promela.html.
19. CMU: The SMV system http://www.cs.cmu.edu/ modelcheck/smv.html.

20. Welch, T.A.: A technique for high-performance data compression. IEEE Computer **17**(6) (1984) 8–19
21. Green Hills Software: Integrity-178B real-time operating system http://www.ghs.com/products/safety_critical/integrity-do-178b.html.
22. Wind River: VxWorks real-time operating system http://www.windriver.com/vxworks/.