Step-by-Step Guide for SATCheck Artifact

Patrick Lam and Brian Demsky

The Getting Started guide explains how to reproduce the results in the paper, up to the generation of the graphs. This document provides more information about the parts of the artifact and how they specifically connect to the paper.

1. Example

The motivating example corresponds to the `linuxlock` example with problem size 1. Recall that you use `bench.sh` to run a benchmark; to only run problem size 1, change `SIZES` in the `benchmark-config.sh` file in the benchmark directory.

We'll discuss each part of the example below.

The example discusses the SATCheck main loop. The code for the main loop is in `model.cc`. Execution points are implemented in the `ExecPoint` class, found in `execpoint.cc`.

2. Event Graph & Encoding

If you add `DUMP_EVENT_GRAPHS` to `config.h`, you will get the event graphs in `.dot` format when you run a program under SATCheck. The graphs in the paper have been trimmed and we've manually extracted subgraphs to better illustrate our points.

The event graph is stored in memory as a tree of `EPRecord` objects, manipulated by the `ConstGen` class. We implemented the value set analysis and encodings in the `StoreLoadSet` class.

Section 4 in the paper is a fairly literal description of the `ConstGen` class, and we don't think that much needs to be said here. Note that remaining goals are stored in the `goalset` field of `ConstGen`. These goals search for unobserved behaviours.

3. Exploring

The schedule builder is found in the `ScheduleBuilder` class. That class includes the construction of the wait pairs, which modifies the execution's schedule for the next concrete execution to be visited.

4. Extensions

Field support Most of the work to support fields happens in the front-end; see Section 6 in this document for information on the front-end. Our handlers for reads and writes (`LoadHandler` and `StoreHandler` respectively) account for field accesses and generate the appropriate instrumentation, which the backend handles.

Sharing between Instances of Uninterpreted Functions. Internally, our instrumentation identifies the uninterpreted function instance using the `MC2_function_id()` call. The frontend assigns a unique function identifier for each static point in the code that generates an uninterpreted function annotation. In the backend, uninterpreted functions are handled in `MCExecution::function`.

Incremental Solving. The SAT encoding reuse code can be found in `ConstGen::canReuseEncoding()`.

TSO Extension. As described in the Getting Started guide, enable TSO by defining it in `config.h`. The Makefile creates two copies of the runtime library: one for SC and one for TSO. The implementation of TSO is scattered around the codebase, but `constgen.cc` generates key TSO constraints.

5. Test Schedule Generation

Enabling the `SUPPORT_MOD_ORDER_DUMP` option in `config.h` will expose schedule graphs in `dot` format; see `MCExecution::dumpExecution` in `model.cc`. Generating test case specifications would be analogous to `dumpExecution`'s implementation.

6. Instrumentation

The Clang front-end lives in the `clang` subdirectory. It takes an unannotated C file and outputs annotated C code to standard output (as invoked by `bench.sh`). It assumes that the C file performs operations on shared memory using calls to `load_NN`, `store_NN`, and `rmw_NN`; the primitives are defined in `include/libinterface.h`.

When the AST traversal encounters a shared memory access, it records the call and inserts instrumentation; see, for example, the `LoadHandler` code in `clang/src/add_mc2_annotations.cpp`. When the frontend inserts instrumentation, it also records the variables being instrumented and, in a second pass, adds additional instrumentation to indicate uninterpreted functions to the model checker.

Caveats. (1) We assume that programs terminate under unfair schedules. If you have a spin loop for which this is not true, you need to manually insert `MC2_yield()` calls where control flows back to the top of a loop. (2) Our front-end also does not get correct information about source locations for macros. As a result, it will not annotate uses of variables of type `bool` (which is implemented as a macro).

7. Evaluation

The Getting Started guide explained how to reproduce our evaluation. One note: the CAS spinlock example in the paper corresponds to the `linuxlock` example in the codebase.

The benchmark runs work as follows. The `bench.sh` script is shared among scripts; it relies on `benchmark-config.sh` to specify behaviour. If front-end compilation is appropriate, it runs the Clang frontend on the unannotated file to generate an annotated file. Then the script iterates on the requested sizes (`$SIZES` for non-TSO, `$SIZES_TSO` for TSO), compiles the benchmark with the command-line flag `-DPROBLEMSIZE`, and records the time to the `log` file and the output to the `logall` file. We then created a quick and dirty Java parser to read the `log` files as generated by `bench.sh` and create gnuplot files.

The expectations for each benchmark are 1) that its main function is called `user_main()`, and 2) that it uses the shared memory access functions which we define in `libinterface.h`. Apart from that, benchmarks should use the standard C11 thread library. Our `bench.sh` script then compiles the benchmark against our `libmodel.so` shared library, which contains its own `main()` function to explore the benchmark's behaviours as described as "the SATCheck main loop" in the paper.

We've included scripts for CDSChecker, Nidhugg, and CheckFence, but we did not package them as part of our evalution.