Ultimate Taipan and Race Detection in Ultimate (Competition Contribution)

Daniel Dietsch[⋆], Matthias Heizmann, Dominik Klumpp(✉), Frank Schüssele, and Andreas Podelski

University of Freiburg, Freiburg im Breisgau, Germany
klumpp@informatik.uni.freiburg.de

Abstract. Ultimate Taipan integrates trace abstraction with algebraic program analysis on path programs. Taipan supports data race checking in concurrent programs through a reduction to reachability checking. Though the subsequent verification is not tuned for data race checking, the results are encouraging.

1 Verification Approach

Ultimate Taipan [6,7] verifies programs using an approach based on trace abstraction [8]. The program is represented as a control flow automaton: Letters correspond to program statements, accepting states correspond to error locations, and accepted words are *error traces*. The verification consists of proving that all error traces are *infeasible* (they cannot be executed). To this end, Taipan picks an error trace from the control flow automaton, and computes the corresponding *path program*, i.e., the projection of the program on the statements in the trace. Taipan then uses *symbolic interpretation with fluid abstractions* [6], a variant of algebraic program analysis, to prove correctness of this path program. If this fails, the algorithm falls back to an interpolation-based method to prove correctness of the trace itself. In either case, the resulting predicates are used to build a Floyd/Hoare-automaton [8] that accepts a regular language of infeasible traces. This automaton is subtracted from the program's control flow automaton, yielding a refined abstraction. Taipan repeats this procedure in a loop until it finds a feasible error trace (the program is incorrect) or the abstraction is empty (all error traces are infeasible, the program is correct).

For concurrent programs, Taipan performs a *naïve sequentialization*, and considers the interleaving product of all threads as a (nondeterministic) sequential program. Verification then proceeds on this program as it would for any other sequential program. Note that this also affects the notion of *path program*, i.e., path programs are also just sequential programs.

Taipan is part of the Ultimate framework, and uses the same front-end as other Ultimate tools. C programs are first translated to the intermediate verification language Boogie [10], the resulting Boogie program is converted into a control flow automaton, which is then verified. The translation from C to

⋆ Jury Member: Daniel Dietsch

Boogie models heap and stack memory through Boogie arrays (associative maps), where pointers correspond to indices. To simplify the subsequent verification, any variables, arrays and structures that are guaranteed to never be accessed through a pointer are instead translated to corresponding Boogie variables.

2 From Data Races to Reachability

Since SV-COMP'22, TAIPAN can check for data races in concurrent programs. A program written in C contains a data race if there are two different threads, *(i)* one thread writes to a memory location and the other thread writes to or reads from the same memory location, *(ii)* at least one of the accesses is not atomic, and *(iii)* neither access *happens-before* the other. The C standard [9], section 5.1.2.4, gives the precise definition. Data races constitute undefined behaviour.

ULTIMATE supports data race checking through a reduction to reachability. This reduction is implemented as part of our translation from C to our custom Boogie dialect. Contrary to C, data races do not constitute undefined behaviour in our Boogie dialect. The semantics prescribes that "simple" Boogie statements – (nondeterministic) assignments and `assume` statements – execute atomically. We consider all interleavings of these atomic statements, i.e., we assume sequential consistency. Hence the correctness of the generated Boogie programs is well-defined, even if the input C program has undefined behaviour. Any verification algorithm for concurrent programs can be applied to the resulting Boogie program, including the algorithm implemented by TAIPAN.

The reduction to reachability proceeds as follows. For every global variable `x`, we introduce a fresh Boolean global variable `race_x`, which tracks read and write accesses to `x`. By comparing the current value of `race_x` to some value it previously held, we can detect if `x` has been accessed since. We call an atomic Boogie statement that represents a C statement or an evaluation step for a C expression an *action*. Let `<read(x)>` denote an action that reads the value of `x`, and let `<write(x)>` denote an action that assigns a new value to `x`. Our translation wraps such actions in data race detection code as shown in the following listings, where `tmp` is a boolean, thread-local variable.

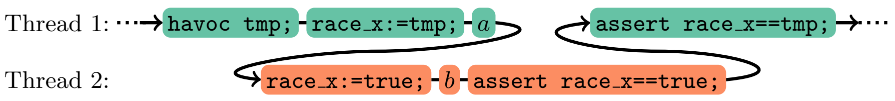
```
race_x := true;
<read(x)>
assert race_x == true;
```

```
havoc tmp; // nondeterministic assignment
race_x := tmp;
<write(x)>
assert race_x == tmp;
```

For an action a, we call the sequence of Boogie statements that results from this wrapping *block*(a). Note that a is always contained in *block*(a). Our translation ensures that if an action a is part of an atomic block (delimited by `__VERIFIER_atomic_*`), then the entire *block*(a) falls inside that atomic block.

For two actions a and b, we say that *block*(b) *can interrupt block*(a) if there exists a program execution that executes *block*(a) up to and including the action a, then fully executes *block*(b), and then continues to execute the remaining assert statement of *block*(a). Hence, a *block*(a) can interrupt *block*(b) or vice versa if and only if at least one of the actions a or b is not atomic, and neither *happens-before* [9] the other.

For an action a, the assert statement in $block(a)$ cannot fail, unless there is an action b such that *(i)* $block(b)$ can interrupt $block(a)$, and *(ii)* a and b both access the same variable `x`. For instance, let a be an action that writes to `x`, and let b be an action that reads from `x`. In the following example, $block(b)$ can interrupt $block(a)$ and the last `assert` statement can fail because `false` can be chosen as value of `tmp`.



Based on the definition of data races we distinguish three cases for the actions a and b:

two reads: The `assert` statements cannot fail for any interleaving because both blocks set `race_x` to the same value. The fact that this value is `true` has no significance; it only matters that the value is fixed.

a read r and a write w: If $block(w)$ can interrupt $block(r)$, the `assert` statement for r can fail if $block(w)$ assigns `tmp` (and consequently, `race_x`) to `false`. Similarly, if $block(r)$ can interrupt $block(w)$, the assert statement for w can fail (again, if `tmp` has value `false`).

two writes w_1, w_2: If some $block(w_i)$ can interrupt $block(w_{3-i})$, the assert statement for w_{3-i} can fail (the blocks may assign different values to `race_x`).

From this case distinction we conclude that in the translated Boogie program, an assert statement added for data race detection can fail if and only if the original C program contains a data race.

Our encoding is independent of the synchronization mechanisms used to rule out data races. Whether the program uses `__VERIFIER_atomic_*`, pthread mutexes, or directly implements locking mechanisms, no special handling is needed. Our implementation supports not only (primitive) global variables, but also data on the heap (accessed through pointers) as well as off-heap structures and arrays. In such cases, instead of a Boolean variable `race_x`, more complicated data structures are needed. We mirror the data layout with Boolean fields: For every data array, there exists a corresponding Boolean array, for every structure, there is a corresponding structure with Boolean-valued fields, etc.

This handling of complex data types also allows us to deal with aliasing issues: Ultimate models memory as an associative array `mem : [Pointer]Int`, with pointers as indices. Our race detection encoding creates a corresponding boolean-valued associative array `race_mem : [Pointer]Boolean`. The instrumentation for an access to a memory location through a pointer `p` then manipulates the entry `race_mem[p]`. If pointers `p` and `q` point to the same memory location ℓ at runtime, then `race_mem[p]` and `race_mem[q]` refer to the same array entry. Hence, if there is a data race on ℓ, one of the generated `assert` statements can fail.

3 Strengths and Weaknesses

Our encoding of data races is independent of the subsequent verification algorithm. We have employed this encoding since SV-COMP 2022 [2], for Taipan as well as in the Ultimate tools Automizer and GemCutter (Ultimate Kojak currently does not support concurrency).

We inherit limitations of the respective verification algorithms. Taipan is unable to prove correctness of programs with an unbounded (or very high) number of threads. The NoDataRace category contains many such programs. Overall, the Ultimate tools perform competitively in the NoDataRace-Main category, with Automizer, GemCutter and Taipan reaching 4th, 5th and 6th place, respectively. In comparison with last year's performance in the demo category (4th, 1st and 2nd place), a major factor seems to be the large number of new correct benchmarks, where we do not perform as well yet. Perhaps some tuning of the subsequent verification algorithms to the detection of data races can lead to improvements in the future.

The presented encoding of data races as reachability is compositional, and independent of the number of threads that are running concurrently: We always add a single assertion per access, in contrast to some other methods [4].

One limitation of our implementation is that, from a feasible trace that ends in an assertion violation, it is not always immediately clear which accesses have a data race. In order to support violation witnesses for data races in future editions of SV-COMP, a more detailed analysis of the trace will be needed.

Our performance suffers in some cases due to a large amount of instrumentation, e.g. in benchmarks where large structs are copied: Currently, we handle each byte in the struct separately. In the future, we hope to improve the implementation to (i) handle reads and writes of large memory chunks more efficiently, (ii) detect more situations in which a concurrent access can be easily ruled out, and no instrumentation is needed, and (iii) making parts of the generated data race detection code atomic, thus reducing the number of interleavings.

4 Architecture, Setup, Configuration, and Project

Ultimate Taipan is part of Ultimate[1], a program analysis framework written in Java and licensed under LGPLv3[2]. Taipan version 0.2.2-2329fc70 requires Java 11 and Python 3.6. The submitted `.zip` archive contains the Linux version of Taipan, binaries of the required SMT solvers[3], and a Python wrapper script. Taipan is invoked with

```
./Ultimate.py --spec <p> --file <f> --architecture <a> --full-output
```

where `<p>` is an SV-COMP property file, `<f>` is an input C file, `<a>` is the data model (`32bit` or `64bit`), and `--full-output` enables verbose output to `stdout`. A violation or correctness witness may be written to the file `witness.graphml`. The benchmarking tool BenchExec [3] supports Taipan through the tool-info module `ultimatetaipan.py`[4]. Taipan participates in all categories, as declared in its SV-COMP benchmark definition file `utaipan.xml`[5].

1 `ultimate.informatik.uni-freiburg.de` and `github.com/ultimate-pa/ultimate`

2 `www.gnu.org/licenses/lgpl-3.0.en.html`

3 Z3 (`github.com/Z3Prover/z3`), CVC4 (`cvc4.github.io/`) and Mathsat (`mathsat.fbk.eu`)

4 `github.com/sosy-lab/benchexec/blob/main/benchexec/tools/ultimatetaipan.py`

5 `gitlab.com/sosy-lab/sv-comp/bench-defs/-/blob/main/benchmark-defs/utaipan.xml`

Data Availability Ultimate Taipan can be found in the archive of all verifiers and validators participating in SV-COMP'23 [1]. Additionally, the `.zip` archive containing only Taipan is available online[6] and on Zenodo [5].

References

1. Beyer, D.: Verifiers and validators of the 12th Intl. Competition on Software Verification (SV-COMP 2023). Zenodo (2023). https://doi.org/10.5281/zenodo.7627829
2. Beyer, D.: Progress on software verification: SV-COMP 2022. In: TACAS (2). Lecture Notes in Computer Science, vol. 13244, pp. 375–402. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20
3. Beyer, D., Löwe, S., Wendler, P.: Reliable Benchmarking: Requirements and Solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
4. Coto, A., Inverso, O., Sales, E., Tuosto, E.: A prototype for data race detection in cseq 3 - (competition contribution). In: TACAS (2). Lecture Notes in Computer Science, vol. 13244, pp. 413–417. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_23
5. Dietsch, D., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate Taipan SV-COMP 2023 Competition Contribution. Zenodo (Dec 2022). https://doi.org/10.5281/zenodo.7480186
6. Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schüssele, F.: Ultimate Taipan with Symbolic Interpretation and Fluid Abstractions - (Competition Contribution). In: TACAS (2). Lecture Notes in Computer Science, vol. 12079, pp. 418–422. Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_32
7. Greitschus, M., Dietsch, D., Podelski, A.: Loop invariants from counterexamples. In: SAS. Lecture Notes in Computer Science, vol. 10422, pp. 128–147. Springer (2017). https://doi.org/10.1007/978-3-319-66706-5_7
8. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of Trace Abstraction. In: SAS. Lecture Notes in Computer Science, vol. 5673, pp. 69–85. Springer (2009). https://doi.org/10.1007/978-3-642-03237-0_7
9. ISO: ISO/IEC 9899:2011 Information technology — Programming languages — C. International Organization for Standardization, Geneva, Switzerland (2011)
10. Leino, K.R.M.: This is Boogie 2 (June 2008), `https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/`

[6] `gitlab.com/sosy-lab/sv-comp/archives-2023/-/blob/main/2023/utaipan.zip`