# Intrepid: A Scriptable and Cloud-Ready SMT-Based Model Checker

Roberto Bruttomesso(✉)

Via Castronno, 48, 21040 Morazzone, VA, Italy

**Abstract.** Intrepid is an SMT-based model checker that provides a rich set of APIs for creating, simulating, and verifying state machines expressed as circuits (just like Simulink or Lustre models). Intrepid may be further used in its Docker container version to be deployed on a local or in a cloud-based infrastructure. The container exposes an equivalently powerful REST API for operating with the model checker. Verification of safety properties in Intrepid is performed in a bit-precise manner, including operations involving integers and floating point arithmetic. Intrepid features standard verification engines as well as multi-property optimizing engines which are suitable for automated test generation tasks, such as MC/DC test generation for avionics.

## 1 Introduction

Model Checking has been successfully employed for decades in industrial environments such as Electronic Design Automation (e.g., equivalence checking for RTL power reduction [2]) and Control Engineering (e.g., automated test generation for avionics and automotive [6,13]), or modern re-implementations of established techniques for network security such as the derivation of attack graphs [1,16,27,29].

Oftentimes companies do not have the resources, the knowledge, or the interest in building an in-house model checker to use as a backend for a new application. Rather, they tend to rely on a free academic tool [7–9,15,17,23,25,30] or to buy licenses for a commercial product from a third-party company [22]. In either case the user of the chosen tool is immediately confronted with the task of translating an instance of her problem into the particular language accepted by the backend, as well as parsing a counterexample for mapping it back to the original problem. While in some contexts the translation effort is not an issue, in some scenarios it represents a major hurdle, especially from a performance and usability perspective. This issue is particularly evident in applications that require a high degree of interaction between the application and the model checker, such as Automated Test Generation [13] or attack graph generation [1,29].

Intrepid aims to tackle these problems by providing a rich Python-based API where input models and the verification steps are executable Python scripts[1],

A demonstration video is available at https://youtu.be/n-0Y_iJqkqY.

[1] E.g.: `[ctx.mk_input('i' + str(i)) for i in range(100)]` to create 100 inputs.



A. Lluch Lafuente and A. Mavridou (Eds.): FMICS 2021, LNCS 12863, pp. 202–211, 2021.
https://doi.org/10.1007/978-3-030-85248-1_13

which can be imported, reused, and extended. Counterexamples can be stored as Python dictionaries or as pandas dataframe, one of the most popular representation for tabular data in Python. Intrepid can be thus used as a rapid-prototyping tool, where the heavy solving tasks are silently delegated to an underlying efficient C++ library. Intrepid can also be started as a server running in a Docker container, on a local or remote machine. The server exposes a rich REST API that can be used to construct, simulate, and solve model checking problems. Under the hood, Intrepid relies on the powerful SMT-solver Z3 [24] for solving satisfiability queries and for performing quantifier elimination required by the model checking engines.

Intrepid is distributed as a library for Python-3.8. It can be installed by issuing the command `pip3 install intrepyd`[2]. The REST API relies on the Docker container `robertobruttomesso/intrepid`. The Python code is available at https://github.com/formalmethods/intrepid and can be used under the liberal BSD-3 license.

## 2 Constructing Models

In Intrepid models are standard, word-level sequential circuits defined as follows:

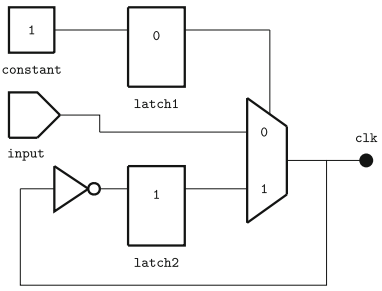
```
circuit : constant | input | latch | circuit op circuit
```

where constants, inputs, and latches can be of type Boolean, signed or unsigned integers of size in {8, 16, 32, 64}, floating point of size in {16, 32, 64}, or real (the only infinite-precision type). `op` is an arithmetic operator, a comparison relation, or a Boolean gate, applied to the proper circuit types, essentially following the typing rules of the SMT-LIB language [3]. Constants, inputs, latches, and operators can be created using an instance of the `Context` object. Intrepid's language is similar in semantic and expressiveness to the BTOR2 format for hardware model checking [25].

Figure 1 shows the creation of a circuit representing a clock signal (a signal that toggles at each time step). The `Context` is created at line 3 and stored in variable `ctx`. At line 6 and 7 the `input` and a `latch1` of type Boolean are created. At lines 8–10 `latch1` is given an initial and a next state: this step is performed after the latch creation to allow the specification of sequential loops (i.e., loops that involve at least a latch. Combinational loops are not allowed). Lines 11–15 perform the creation and initialization of `latch2`.

Because models are Python scripts, they can be placed into convenient functions or classes inside Python modules. The clock model above, for instance, can be imported from the `intrepyd.components.eda` submodule.

[2] Notice the "y" in the name of the Python package: the name "intrepid" was already taken.




(a)

```
1 from intrepyd import Context
2
3 ctx = Context()
4 bt = ctx.mk_boolean_type()
5 constant = ctx.mk_true()
6 input = ctx.mk_input('input', bt)
7 latch1 = ctx.mk_latch('latch1', bt)
8 ctx.set_latch_init_next(latch1,\
9                          ctx.mk_false(),\
10                         constant)
11 latch2 = ctx.mk_latch('latch2', bt)
12 clk = ctx.mk_ite(latch1, latch2, input)
13 ctx.set_latch_init_next(latch2,\
14                         ctx.mk_true(),\
15                         ctx.mk_not(clk))
```

(b)

**Fig. 1.** The encoding of a clock signal, starting with a random value: (a) the schematic of the circuit (b) the encoding in Intrepid, `clk` being the output.

### 2.1 Translating Industrially-Relevant Models

Intrepid is intended to be used via its API, however, in order to facilitate the processing of existing industrial models, Intrepid comes with two submodules `intrepyd.lustre2py` and `intrepyd.iec611312py`. The first one translates Lustre models [18], while the second one translates the IEC-61131-3 Structured Text language models in OpenPLC format, into Intrepid's scripts. Both translators, based on the ANTLR parser [26], do not fully support all the aforementioned languages' constructs (e.g.: arrays are not supported).

The Lustre frontend has been tested with a subset of the models available from [20] (see Sect. 4.1). The Structured Text encoder is currently in an earlier proof-of-concept stage: its main purpose is to demonstrate that the translation is possible, by providing an initial implementation. It has been tested on an OpenPLC model[3], in turn translated from a Simulink/Stateflow model of an infusion pump, using the Simulink PLC Coder tool[4]. The infusion pump model is part of the CocoSim tool [7] test suite[5].

## 3 Simulating Models

Since models are circuits, they can be simulated for a given number of time steps. The result of the simulation is a `Trace` object. For each simulation step the trace assigns a value to the sub-circuits that are being watched.

Figure 2 shows the simulation of the clock circuit of Fig. 1, conveniently imported from the `eda` module. The simulator is created at line 7. Both the clock output and input are "watched" at lines 8 and 9: watched signals are those that will show up in the trace. Line 10 creates a new empty trace, and line 11

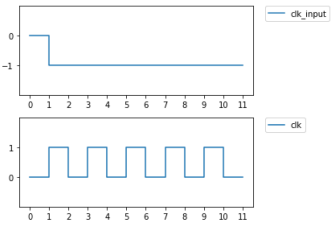
[3] Available at https://bit.ly/3hggxgV.
[4] Simulink PLC Coder is a Mathwork's proprietary tool.
[5] Available at https://bit.ly/3qw4osy.

```
1 from intrepyd import Context
2 from intrepyd.components.eda
      import mk_clock
3 from intrepyd.plots
      import plot_trace_dataframe
4
5 ctx = Context()
6 clk, clk_input = mk_clock(ctx, 'clk')
7 simulator = ctx.mk_simulator()
8 simulator.add_watch(clk)
9 simulator.add_watch(clk_input)
10 trace = ctx.mk_trace()
11 trace.set_value(clk_input, 0, 'F')
12 simulator.simulate(trace, 10)
13 df = trace.get_as_dataframe(ctx.net2name)
14 print(df)
15 plot_trace_dataframe(df)
```

(a)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| clk_in | F | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| clk | F | T | F | T | F | T | F | T | F | T | F |

(b)




(c)

**Fig. 2.** Simulating a clock signal over 10 time steps: (a) the encoding in Intrepid, (b) the pandas dictionary printed at line 14, (c) the signal values graphs generated at line 15.

initializes the first value of the input to `F`. If an input value is not specified for a time step, the simulator will assign to a "don't care" value `?`. During the simulation, at line 12, these values are propagated through the circuit, but, if irrelevant, will not show up at the circuit output: this is exactly the case in our example. At line 13 the trace is converted in a pandas dataframe, and then printed and plotted at lines 14 and 15. In the plots, `F` is mapped to 0, `T` to 1, and `?` to $-1$.

## 4 Model Checking

Model Checking in Intrepid consists in defining reachability `targets`, i.e., Boolean signals for which the tool tries to find a `trace`. In standard terminology a target is a bad state corresponding to the negation of a safety property, and a trace is a counterexample that disproves its validity. Safety properties are the only ones supported by Intrepid. In order to support a wider range of properties a user could rely on the approach of [11] to create monitor circuits, by constructing them using Intrepid's basic APIs. It is important to notice that Intrepid's engines attempt to reach multiple targets at once.

**Bounded Model Checking and Temporal Induction.** Bounded Model Checking (BMC) is the process of reaching a target by unrolling the circuit for finite number of steps. The unrolling is performed in a backward manner, from the targets to the inputs. Latches are recursively replaced with their unrolled next state signal. Optionally, Temporal Induction (TI) can be enabled to prove target's unreachability. Intrepid essentially follows the "Zig-zag" approach of [12], as well as its strategy of dynamically adding difference constraints.

**Optimizing Bounded Model Checking.** Some applications such as Automated Test Generation require to find traces that satisfy the most number of targets at once. Since each trace is turned into a test, and each test might need to undergo manual revision, it is important to produce a small number of traces that cover all the targets. The Optimizing Bounded Model Checking (OBMC) engine aims at solving exactly this problem. By relying on the optimization procedure of Z3 [5], it is possible to simply use the MAX-SMT solver instead of the default one to find traces that satisfy the maximum number of active targets (a sample application is presented in Sect. 5.2).

**Backward Reachability.** Backward Reachability (BR) is inspired to the exploration algorithm behind the MCMT model checker [16], which is adapted for Intrepid's circuit-like models. The algorithm keeps a frontier of states to explore, and a set of blocked states. Blocked states are states that have been already explored, and therefore do not need to be enumerated again. The frontier is initialized with the targets to reach. Then the main loop starts. At each iteration a state $S$ is popped from the frontier: if it intersects the initial states, then some target is reachable, otherwise the pre-image states of $S$ that are not blocked already are added to the frontier, and $S$ is added to the blocked states. The loop exits when the frontier is empty. Backward Reachability relies on Z3's quantifier elimination to rule out non-Boolean inputs from the enumerated states.

### 4.1 A Comparison of the Engines

Table 1 reports an evaluation of the engines on a subset of the benchmarks from [20]. Overall we run 848 divided in 6 families. Each benchmark contains either a safe or an unsafe property. The experiments show that BMC solves

**Table 1.** A comparison on the lustre models from [20]. The tests have been run on an Intel i7-8565U 1.80 GHz machine, with 32 GB of RAM, running Ubuntu Linux and Intrepid version 0.10.3. The timeout was set at 60 s. The column "TO" reports the number of timed-out benchmarks. The column "Best" indicates on how many benchmarks the tool was the fastest to find the answer. Full raw data is available at https://bit.ly/3duFd4a. The benchmarks and the scripts to run the tests are available under the folder `benchmarks` of the Github repository). Bold-face fonts highlight the best performing solver per each benchmark family.

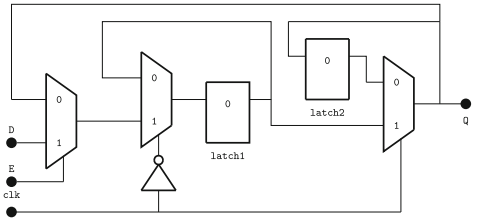
| Family | BMC | | | | BMC+TI | | | | BR | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Safe | Unsafe | TO | Best | Safe | Unsafe | TO | Best | Safe | Unsafe | TO | Best |
| Protocol | 0 | **14** | 22 | 14 | **16** | **14** | **6** | **16** | 15 | 7 | 14 | 0 |
| Simulation | 0 | **58** | 132 | 57 | **68** | 52 | **70** | **68** | 64 | 43 | 83 | 6 |
| Memory1 | 0 | **110** | 172 | **108** | 10 | 109 | **163** | 8 | **17** | 10 | 255 | 11 |
| Memory2 | 0 | **98** | 84 | **97** | 25 | 96 | **61** | 22 | **35** | 57 | 90 | 15 |
| Misc | 0 | **62** | 48 | **54** | 18 | **62** | 30 | 22 | **34** | 62 | **14** | 21 |
| Large | 0 | 0 | 48 | 0 | 12 | 0 | 36 | **7** | **13** | 0 | **35** | 6 |
| Total | 0 | **342** | 506 | **330** | 149 | 333 | **366** | 143 | **178** | 179 | 491 | 59 |

the most unsafe benchmarks, as expected, while BR solves the most safe ones. BMC+TI solves the most benchmarks overall (it reports the least number of timeouts). A comparison with other model-checkers is left as future work.

## 5 Sample Applications

### 5.1 Equivalence Checking for Clock-Gating

Sequential clock gating is an important technique used in EDA to reduce the power consumption of digital circuits [4]. The idea of clock-gating is to reduce flip-flop value toggles, which is known to be draining a substantial amount of power, by adding extra logic to the circuit, in such a way that the power used for the new logic is highly compensated by the reduced toggles.




```
 1 from intrepyd.components.eda import \
 2       mk_clock, Delay, FlipFlopD, FlipFlopDE
 3
 4 t = ctx.mk_uint16_type()
 5 bt = ctx.mk_boolean_type()
 6 init = ctx.mk_number('0', t)
 7 d = ctx.mk_input('d', t)
 8 e = ctx.mk_input('e', bt)
 9 clk, _ = mk_clock(ctx, 'clk')
10 back1 = FlipFlopDE(ctx, 'back1', t, clk)
11 back1.set_init_next(init, d, e)
12 forth1 = FlipFlopD(ctx, 'forth1', t, clk)
13 forth1.set_init_next(init, back1.Q)
14 back2 = FlipFlopDE(ctx, 'back2', t, clk)
15 back2.set_init_next(init, d, e)
16 e_prime = Delay(ctx, 'delay', bt)
17 e_prime.set_init_next(ctx.mk_true(), e)
18 forth2 = FlipFlopDE(ctx, 'forth2', t, clk)
19 forth2.set_init_next(init,\
20                       back2.Q,\
21                       e_prime.Q)
22 diff = ctx.mk_neq(forth1.Q, forth2.Q)
23 br = ctx.mk_backward_reach()
24 br.add_target(diff)
25 result = br.reach_targets()
```

(c)

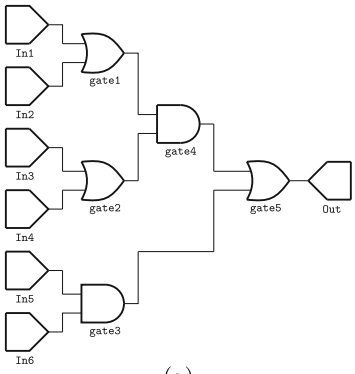
**Fig. 3.** An an equivalence checking problem for a power reduction technique called Stability Condition [14]: (a) the encoding of a register with enable, (b) the schematic of the problem, where the enable signal `en` is propagated to the register `forth`, delayed by one cycle, and (c) the encoding of the problem in Intrepid.

Figure 3b shows a transformation of a chain of a pair of registers `back1-forth1` into a more power-efficient one `back2-forth2`. Due to the changes in the design, clock-gating opportunities must be proven correct: in the schematic above the pin `diff` must never evaluate to `1`. `diff` is be passed as a target to the backward reachability an engine for proving its unreachability.

The SMT-like language of Intrepid allows the definition of clock-gating checks at the "word level", thus operating on the original registers as a whole rather than on their individual flip-flops (in the example we are running the check for a 16-bits register).

### 5.2 Automated Test Generation of MC/DC

The avionics standard DO-178C [28] dictates that every Level-A control software must be fully covered by a test suite using the Modified Condition/Decision (MC/DC) coverage metric [10]. Encoding of MC/DC conditions as Boolean or SMT formulas is a well-studied topic: the interested reader may refer to [6] for a simple logical formulation of the problem. Essentially the idea is to create reachable targets such that their traces correspond to tests that satisfy the coverage. Intrepid implements a simple ATG algorithm using only 300 LOC of Python that roughly follows the approach of [13], and is based on repeated calls to the OBMC engine, as shown in Fig. 4.




(a)

| | In1 | In2 | In3 | In4 | In5 | In6 | Out |
|---|---|---|---|---|---|---|---|
| 0 | F | F | T | F | F | F | F |
| 1 | T | F | T | F | F | F | T |
| 2 | F | T | T | F | F | F | T |
| 3 | T | F | F | F | F | F | F |
| 4 | T | F | F | T | F | F | T |
| 5 | F | F | F | F | F | T | F |
| 6 | F | F | F | F | T | T | T |
| 7 | F | F | F | F | T | F | F |

(b)

**Fig. 4.** An example of an execution of ATG on a simple combinational circuit (a) taken from [19]. Each row of the table (b) is an assignment to the inputs representing a test. Pair of tests show the satisfaction of the MC/DC coverage criterion for a specific input. For example $(0, 1)$ is a pair of tests that shows MC/DC for `In1` (a so-called independence pair): `In1` toggles between the two tests, all the other inputs keep the same value, and `Out` toggles. This is a proof that `In1` can affect the behavior of the circuit.

## 6 A REST API for Model Checking

Intrepid is also available as a Docker container, that can be (downloaded and) run with `docker run -p 8000:8000 -d robertobruttomesso/intrepid`[6].

[6] The `docker` framework can be obtained from https://www.docker.com/.

The command starts a local server at port 8000 that exposes a rich REST API that roughly wraps the Python API so far described. The API is still experimental (it lacks for instance a mechanism for authentication), but it is fully operational for constructing, simulating, solving, and retrieving traces.

Figure 5 shows a sample interaction that creates an `and` gate using a command-line tool; similar interactions can be programmed and automated with popular languages such as Python, Ruby, or Javascript. Beside increasing the tool's interoperability with other frameworks and languages, having a containerized application with a REST API is a first step towards the embedding of Model Checking in a cloud environment such as AWS, Azure, or Digital Ocean: these providers can easily host and orchestrate multiple containers with frameworks like Docker-compose or Kubernetes.

One application that we envision for this setting is that of solving large problems that can be partitioned and dispatched to several different engines: the application described in Sect. 5.1, for instance, can be often tackled by partitioning the global equivalence checking problem into thousands of independent smaller ones, one per each clock gating opportunity discovered, using the notion of cut-points [21]. However, several challenges needs to be considered, such as the dispatching of problems and the reconstruction of the results. We leave the investigation of the feasibility of this research direction for a future work.

```
 1 c> curl --location --request POST 'localhost:8000/api/v1/contexts/create' \
 2         --header 'Content-Type: application/json' \
 3         --data-raw '{"name": "default"}'
 4 s> {"result":"default"}
 5 c> curl --location --request POST 'localhost:8000/api/v1/inputs/create' \
 6         --header 'Content-Type: application/json' \
 7         --data-raw '{"context": "default", "type": "bool"}'
 8 s> {"result":"__i0"}
 9 c> curl --location --request POST 'localhost:8000/api/v1/inputs/create' \
10         --header 'Content-Type: application/json' \
11         --data-raw '{"context": "default", "type": "bool"}'
12 s> {"result":"__i1"}
13 c> curl --location --request POST 'localhost:8000/api/v1/nets/ands/create' \
14         --header 'Content-Type: application/json' \
15         --data-raw '{"context": "default", "x": "__i0", "y": "__i1"}'
16 s> {"result":"__n10"}
```

**Fig. 5.** A client-server interaction using the popular command-line tool `curl` (`c>` is the client's query, `s>` is the server's response): lines 1–3 create a new context named `default`, lines 5–7 and lines 9–11 create two inputs, and lines 13–15 create an `and` gate using them. Further documentation for the REST APIs is available at https://bit.ly/3bn2h42.

## 7 Conclusion

We have introduced Intrepid, a scriptable SMT-based Model Checker. We have presented a sketch of its Python API by applying it to a concrete industrially relevant sample application. Intrepid is additionally shipped as a Docker container,

and it exposes a REST API that enables the deployment of the model checker on a remote server, a first step towards the employment of Model Checking in a cloud-based environment.

## References


1. Al Ghazo, A.T., et al.: A2G2V: automatic attack graph generation and visualization and its applications to computer and SCADA networks. IEEE Trans. Syst. Man Cybern. Syst. **50**(10), 3488–3498 (2020). https://doi.org/10.1109/TSMC.2019.2915940
2. Babighian, P., Benini, L., Macii, E.: A scalable ODC-based algorithm for RTL insertion of gated clocks. In: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, vol. 1, pp. 500–505 (2004). https://doi.org/10.1109/DATE.2004.1268895
3. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). http://www.smt-lib.org/
4. Benini, L., et al.: Symbolic synthesis of clock-gating logic for power optimization of control-oriented synchronous networks. In: Proceedings of the European Design and Test Conference, ED TC 1997, pp. 514–520 (1997). https://doi.org/10.1109/EDTC.1997.582409
5. Bjørner, N., Phan, A.-D., Fleckenstein, L.: *vZ* - an optimizing SMT solver. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 194–199. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_14
6. Bloem, R.P., et al.: Model-based MCDC testing of complex decisions for the Java card applet firewall. In: VALID Proceedings. Ed. by IARIA, pp. 1–6 (2013)
7. Bourbouh, H., Brat, G., Garoche, P.-L.: CoCoSim: an automated analysis framework for Simulink/Stateflow. In: Model Based Space Systems and Software Engineering - European Space Agency Workshop (MBSE 2020) (2020)
8. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
9. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 510–517. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_29
10. Chilenski, J.J.: An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical report. DOT/FAA/AR-01/18, Boeing Commercial Airplane Group, April 2001
11. Claessen, K., Eén, N., Sterin, B.: A circuit approach to LTL model checking. In: FMCAD, pp. 53–60 (2013). http://ieeexplore.ieee.org/xpl/freeabsall.jsp?arnumber=6679391
12. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electron. Notes Theor. Comput. Sci. **89**(4), 543–560 (2003). BMC 2003. ISSN: 1571-0661
13. Ferrante, O., Ferrari, A., Marazza, M.: Model based generation of high coverage test suites for embedded systems. In: 19th IEEE European Test Symposium, ETS, pp. 1–2 (2014). https://doi.org/10.1109/ETS.2014.6847843
14. Fraer, R., Kamhi, G., Mhameed, M.K.: A new paradigm for synthesis and propagation of clock gating conditions. In: 2008 45th ACM/IEEE Design Automation Conference, pp. 658–663 (2008)

15. Gacek, A., Backes, J., Whalen, M., Wagner, L., Ghassabani, E.: The JKind model checker. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 20–27. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_3 ISBN: 978-3-319-96142-2
16. Ghilardi, S., Ranise, S.: MCMT: a model checker modulo theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 22–29. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_3
17. Goel, A., Sakallah, K.: AVR: abstractly verifying reachability. In: TACAS 2020. LNCS, vol. 12078, pp. 413–422. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_23 ISBN: 978-3-030-45190-5
18. Halbwachs, N., et al.: The synchronous data flow programming language LUSTRE. In: Proceedings of the IEEE 1991, pp. 1305–1320 (1991)
19. Hayhurst, K.J., et al.: A Practical Tutorial on Modified Condition/Decision Coverage. TM 2001-210876. Langley Research Center. NASA, Hampton, May 2001
20. Kind2 benchmarks. https://github.com/kind2-mc/kind2-benchmarks
21. Kuehlmann, A., Eijk, C.A.J.: Combinational and sequential equivalence checking. In: Hassoun, S., Sasao, T. (eds.) Logic Synthesis and Verification. SECS, vol. 654, pp. 343–372. Springer, Boston (2002). https://doi.org/10.1007/978-1-4615-0817-5_13
22. Mathworks: Simulink Design Verifier. https://www.mathworks.com/products/sldesignverifier.html
23. Mattarei, C., et al.: CoSA: integrated verification for agile hardware design. In: FMCAD. IEEE (2018)
24. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
25. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC and Boolector 3.0. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 587–595. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_32
26. Parr, T.: The Definitive ANTLR 4 Reference, 2nd edn. Pragmatic Bookshelf, Raleigh (2013). ISBN: 978-1-93435-699-9. https://www.safaribooksonline.com/library/view/the-definitive-antlr/9781941222621/
27. Ritchey, R.W., Ammann, P.: Using model checking to analyze network vulnerabilities. In: Proceeding of the 2000 IEEE Symposium on Security and Privacy, SP 2000, pp. 156–165 (2000)
28. RTCA: DO-178C: Software Considerations in Airborne Systems and Equipment Certification
29. Sheyner, O., et al.: Automated generation and analysis of attack graphs. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy, pp. 273–284 (2002). https://doi.org/10.1109/SECPRI.2002.1004377
30. Vizel, Y., Gurfinkel, A.: Interpolating property directed reachability. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 260–276. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_17 ISBN: 978-3-319-08867-9