Intrepid: an SMT-based Model Checker for Control Engineering and Industrial Automation

Roberto Bruttomesso

Nozomi Networks SA, Mendrisio, Switzerland
`roberto.bruttomesso@nozominetworks.com`

Abstract

Intrepid is an SMT-based model checking library. It provides a rich API for constructing, simulating, and verify circuits. Verification of safety properties is performed in a bit-precise manner, including operations involving integers and doubles. The incremental multi-property engines are suitable for automated test generation tasks, such as MC/DC test generation. Intrepid can parse Lustre specifications, and a subset of IEC-61131 ST language: by using the PLC language kit provided by Mathworks, we show that also Simulink and Stateflow languages can be processed by Intrepid.

1 Introduction

Formal methods have been successfully employed in Control Engineering (avionics, automotive, train controllers) for decades. Property verification, requirement analysis, and test generation have been partially or fully automated with theorem provers and model checkers, where the latter approach is often referred to as the one with the higher degree of automation and precision in reporting counterexamples.

On the other hand Industrial Automation is a domain where formal methods never really took off: this is unfortunate since Programmable Logic Computers (PLCs) are at the heart of modern industrial plants such as dams, and nuclear power plants. There is no need to highlight the importance of the correctness of programs that control such installations, considering also the growing demand of automation of the industry 4.0.

Intrepid is a model checking library that has been conceived to facilitate the application of formal methods in control engineering and industrial automation applications by providing (i) a rich C and python API that allows for a higher degree of interaction between the engineer and model checking algorithms in a rapid prototyping environment, and (ii) the ability to parse widely used languages in industry such as Lustre, Simulink/Stateflow, and a subset of IEC-61131-3 ST language for PLCs.

Intrepid relies on the powerful SMT-solver Z3 [23] for solving satisfiability queries and for performing quantifier elimination.

1.1 Availability

The easiest way to use Intrepid is via it's Python API. It can be installed using Python's `pip` utility, by issuing the command `pip install intrepyd`. The source code is available at https://github.com/formalmethods for both the core library written in C++ (repository `intrepid` [15]), and the Python API (repository `intrepyd` [17]). The code is released under the liberal BSD-3 license.

1.2 Related Work

Kind2 [6], Luke [21], nuXmv [5], and MCMT [11] are four other SMT-based model checkers.

Kind2 and Luke read Lustre specifications, and they implement a temporal induction engine to solve proof objectives. Kind2 includes several enhancements over the basic algorithms, such as invariant generation. Differently from Intrepid and Luke, Kind2 interprets integers and reals in the classical mathematical domain, not as machine-precise entities. Also, Luke does not offer support for reals. Kind2 and Luke are the most similar tool to ours. Notice also that Lustre and Simulink models have very closely related semantics: Lustre is often used as an intermediate language for processing Simulink.

nuXmv can read SMV models extended with keywords that allow for the specification of infinite-state systems. It implements a rich portfolio of reachability as well as LTL algorithms, some of which are inherited from NuSMV and specialized for purely Boolean reasoning. Lustre and Simulink models can be translated into SMV models, but we are not aware of a publicly available tool that performs such translation.

MCMT was conceived to reason about array-based systems, an expressive class of infinite state systems that allows encoding of protocols. Intrepid's backward reachability engine is, conceptually, inspired to that of MCMT, and it is extended to handle free inputs (MCMT operates on closed systems instead).

Formal Specs Verifier (FSV) [4] is a proprietary tool developed at ALES/UTRC that can verify Simulink designs by means of a translation into nuXmv. In contrast to ours, their Simulink translator is based on the MatLab APIs, and thus it requires interaction with MatLab. FSV-ATG [9] is an extension of FSV for Automatic Test Generation. Our approach to ATG presented in Section 4 is inspired to the same algorithm. In contrast, our approach is not based on repeated and monolithic calls to the backend (nuXmv again), but leverages the richer API provided by Intrepid.

CocoSim [8] (and its successor CocoSim2) is a tool that integrates with Mathworks Matlab and provides utilities for verification and code generation for Simulink/Stateflow designs. Our approach to the Simulink verification differs in that Intrepid does not require an active execution of Matlab once a circuit has been dumped. At the same time Intrepid could be potentially used as a backend for CocoSim.

Simulink Design Verifier (`SLDV`) [22] is the proprietary model checker that comes as part of the MatLab-Simulink suite. `SLDV` has the capability of solving reachability queries and perform test case generation on Simulink and Stateflow models. Unfortunately only little information is publicy available about the internals of the tool.

2 Preliminaries

2.1 Basic definitions

In the following we shall use the term *circuit* to refer to a connected directed graph where nodes are elements called *nets*. A net can be either an *input*, an *output*, a *gate*, or a *latch*. Inputs are nets that have no incoming connections, outputs are nets that have no outgoing connections, gates are combinational elements such as logical gates, adders, etc., and latches are sequential elements with one incoming and one outgoing connection (simple memory units, also known as delays). Combinational loops are not allowed, i.e., every loop in a circuit must contain at least a latch.

The circuit is typed: every input must be associated with a type, and gates can be applied only between nets that are compatible with their signature. The type of a latch is the same of the net in its incoming connection. In this way all the nets of a circuit have a well defined type.

2.2 Intrepid's Anatomy

Figure 1 depicts a schematic view of Intrepid's software modules. The upper part constitutes the core of the system, the model-checking library (intrepid.dll) that exposes a C-API, easily wrappable by basically any other programming language. The library is written in C++ for performance reasons. At the heart of the library lies the integration with Z3 SMT-solver (z3.dll), which is used by Intrepid to solve combinational proof obligations incrementally, and to perform quantifier elimination when necessary.

Intrepyd is a Python wrapper for the C-API provided by Intrepid. It extends the API by introducing an Object-Oriented layer that hides most low-level details and that allows a seamless and interactive programming experience with the tool. The low-level C-to-Python wrapper is automatically generated by the SWIG framework.

Intrepid compiles and has been tested on Windows, macOS, and Linux.

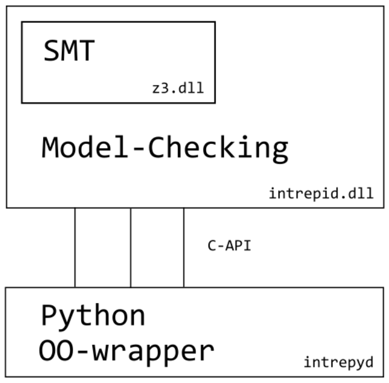


Figure 1: Intrepid's software modules architecture.

2.3 Problem specification and verification

Intrepyd is the main entry-point for problem specification and property verification. In other words, Intrepid's low-level input language is simply python: the circuit is constructed with the creational APIs and the interface of the circuit with the external world is defined by choosing which nets are inputs and ouputs. In this process native python functions can be used to declare sub-circuits that might need to be constructed multiple times with different parameters; programming constructs such as "for" loops can be used to automate the circuit creation process.

Intrepid supports the following data types: Booleans, signed and unsigned integers of size 8, 16, 32, and 64 bits, single and double precision floating point numbers, and reals (as an approximation for single and double precision floating-point numbers). These supported types are common in control engineering applications and are the basic types of dataflow languages such as Simulink, Lustre, and the industrial automation languages defined in the IEC-61131-3 standard for Programmable Logic Computers [14].

Intrepid provides a simulation engine and two verification engines for performing bit-precise safety model-checking of a circuit. It is possible to specify which nets are to be tracked during simulation, and which are to be used as reachability targets by means of the provided API (some examples can be found in the `intrepyd` repository under directory `examples`).

2.4 An example

Figure 2 and Figure 3 show the definition of a counter using Intrepid's python API. The counter is instantiated with a limit of 10, and it is then simulated over twelve time frames. The output of the simulation is reported in Figure 4.

```
import intrepyd as ip
import intrepyd.trace

ctx = ip.Context()
int8type = ctx.mk_int8_type()
ten = ctx.mk_number("10", int8type)
counter, Q = mk_counter(ctx, "counter", type=int8type, limit=ten)
simulator = ctx.mk_simulator()
tr.mk_trace()
simulator.add_watch(counter)
simulator.add_watch(Q)
simulator.simulate(tr, 12)
df = tr.get_as_dataframe(ctx.net2name)
print df
```

Figure 2: Simulation of the output values of a counter for twelve time frames.

```
def mk_counter(context, name, type, limit, init=None, increment=None, enable=None, reset=None):
    """
    Counts from init to limit by increment. When limit
    is reached, (limit, true) is outputted. Enable,
    reset, and custom increment might be specified.
    """
    if init == None:
        init = context.mk_number("0", type)
    if increment == None:
        increment = context.mk_number("1", type)
    if enable == None:
        enable = context.mk_true()
    if reset == None:
        reset = context.mk_false()

    counter = context.mk_latch(name, type)
    notQ = context.mk_lt(counter, limit)
    next = context.mk_ite(reset,\
                          init,\
                          context.mk_ite(context.mk_and(enable, notQ),\
                                         context.mk_add(counter, increment),\
                                         counter))
    context.set_latch_init_next(counter, init, next)
    Q = context.mk_not(notQ, name + '.Q')
    return counter, Q
```

Figure 3: Definition of function `mk_counter`, that implements a sub-circuit counting from zero to a specified limit.

```
           0  1  2  3  4  5  6  7  8  9  10  11  12
counter    0  1  2  3  4  5  6  7  8  9  10  10  10
counter.Q  F  F  F  F  F  F  F  F  F  F   T   T   T
```

Figure 4: The output of the simulation.

3 Engines

3.1 Simulation

Intrepid provides a simulation engine that allows for a quick inspection of the behavior of the circuit by computing the values for the outputs at each time step. This feature is particularly useful for closed systems (circuits with no inputs, or where inputs have been connected with some stub logic); otherwise primary inputs are given default values (0 for numeric types, false for Booleans). The simulator may also be used to re-simulate a counterexample obtained with the model checking engines (in any case, internally the simulator is automatically triggered for validating any newly produced counterexamples).

3.2 Model Checking

Intrepid provides two model checking engines, a Bounded Model Checker (`BMC`) engine and a Backward Reachability (`BR`) engine. Both engines are reachability-based, i.e., they can produce a counterexample showing that a Boolean signal (target) can evaluate to “true”. `BR` can also show that no counterexample can be produced (target is unreachable). Both `BMC` and `BR` are multi-target engines, i.e., they can perform reachability for multiple targets at once, thus reducing the overall computational effort compared to a batch of single-target reachability calls. In particular, the engines pause when the first counterexample is found for at least one target, and it may be resumed to reach the remaining ones, without having to reset the engine. This feature is of great use for Automatic Test Generation.

`BMC` is implemented as a classical unroll-and-solve procedure. A noteworthy feature of Intrepid’s `BMC` engine is the ability to perform an “optimizing” reachability, i.e., it can be used to produce the counterexample that satisfies the highest numbers of targets at once (for a given depth). The optimization feature relies on the powerful and flexible algorithms provided by Z3 [1].

`BR` is an adaption of the exploration algorithm behind the MCMT tool [11], extended to handle input signals. The procedure is complete (can prove a target to be unreachable), and it takes care of eliminating non-Boolean inputs in proof obligations by means of quantifier elimination.

Both `BMC` and `BR` may be used to solve circuits containing non-linear real arithmetic polynomials, since Z3 supports solving and quantifier elimination for that fragment, a rare feature that only a handful of model checkers can display at the time of writing this paper[1].

[1]Notice that Mathwork’s proprietary model checker Simulink Design Verifier [22] cannot handle non-linear arithmetic at the time of writing this paper.

4 A Sample Application: Automated Test Generation of MC/DC

The avionics standard DO-178C [24] dictates that every Level-A control software must be fully covered by a test suite using the Modified Condition/Decision (MC/DC) coverage metric. Being able to compute such test suites with automatic means is of paramount importance for the avionics industry.

A test suite satisfies the MC/DC coverage criterion for a decision in a circuit iff each condition can be shown to independently affect the outcome of the output (see [7] for the precise definitions of decision, condition, MC/DC). Encoding of MC/DC conditions as Boolean/SMT formulas is a well-studied topic: the interested reader may refer to [2] for a recent elegant logical formulation of the problem.

Intrepid implements a simple ATG algorithm on top of the basic API using only 200 LOC of python, including comments. The algorithm roughly follows the approach of [9] for deriving test cases from model checking counterexamples, but, in addition, it also detects unreachable test objectives (by calling the `BR` for individual test objectives as a preprocessing step). The algorithm is based on repeated calls to the `BMC` engine; the optimization feature is used to generate a minimal number of counterexamples (and therefore tests) by maximizing the number of solved targets at each engine invocation.

5 Frontends

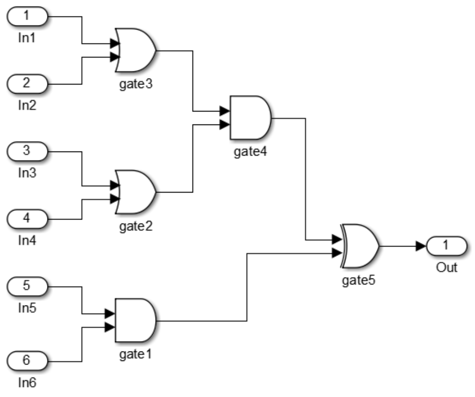
In order to broaden the appliacability of Intrepid we have written two translators from commonly used languages in control engineering and industrial automation into the Intrepid's API: a translator for Lustre [12], and one for IEC-61131-3 Structured Text language [14] (ST from now on).

In both cases the encoding is bit-precise, i.e., there is no abstraction of finite integers or floating-point values into mathematical integers and reals as other model-checkers do: we argue that this choice is the most suitable for a prover that needs to be employed in safety-critical verification tasks, as there must be no room for false positive or false negative results.

Our translator for ST is only partial, and it focuses on a subset of the language ST^0 that is rich enough to express state-machines. Iteration statements for example are not part of the subset we can deal with.

Most importantly, the ST^0 subset is sufficient to capture Mathworks' Simulink and Stateflow dataflow languages: Mathworks provides a so called ”PLC Toolkit” which can be used to translate Simulink/Stateflow circuits into Structured Text (as well as other PLC languages, such as Ladder Logic). The translation comes with a rich set of comments that allow a complete and precise backtrace from the translated model to the original. Therefore, indirectly, Intrepid is able to handle also Simulink/Stateflow.

Originally we started to translate Simulink directly, by building on top of existing Java parsers. However soon we realized that there are many advantages of using a translation from ST rather than a native one from Simulink/Stateflow. First of all ST is a textual language, and it is easy to check if being parsed and translated correctly compared to a graphical language, whose textual representation is harder to read. Second, as far as we know there is no official and complete document provided by Mathworks that explains the details all the Stateflow constructs. The work of [3] gives a deep insight on the semantic of Stateflow, and we have no reason to believe it is incorrect. However it has been validated with simulation rather than



```
examples/atg_2$ python atg_generation.py
There are 6 test objectives:
- 0 unreachable test objectives
- 6 reachable test objectives

Generated tests:
      In1    In2    In3    In4    In5    In6 circuit/Out
0   false  false   true  false   true   true        true
1    true  false   true  false   true   true       false
3   false   true   true  false   true   true       false
4   false   true  false  false   true   true        true
7    true  false  false   true   true   true       false
9    true  false   true  false  false   true        true
11   true  false   true  false   true  false        true
```

Figure 5: An example of an execution of Automated Test Generation on a simple combinational circuit taken from [13]. Each row of the table is an assigment 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 ids that shows MC/DC for "In1" (a so-called independence pair). The test generation takes about 1 second.

being build on top of a formal specification. ST specification instead is public, well understood, and easy to implement.

6 Preliminary Experiments

6.1 Lustre

In this section we report on some experiments conducted on the benchmark suite of the Kind2 tool, available at [18]. The suite contains around 900 designs with only one output, representing a property, or equivalently, the negation of a target. Some targets are reachable (design is Invalid) some other are not (design is Valid).

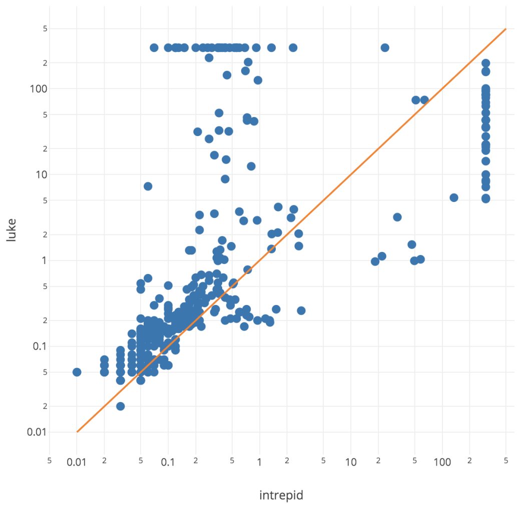
Instead of reporting raw execution times for Intrepid on the benchmarks, we report a comparison with existing tools.

Comparison agains Kind2

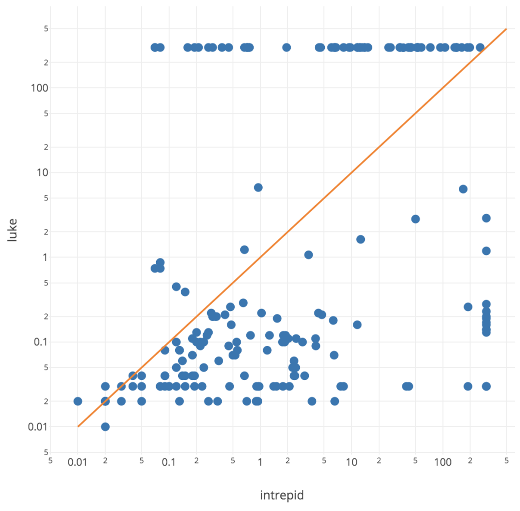
We do not compare against Kind2, because of the different semantics it implements (abstraction of fixed-width integers into mathematical integers in $\mathbb{Z}$). Would it make sense to compare anyways ? We believe it would not. Take for instance a circuit representing a counter: in integer arithmetic in $\mathbb{Z}$ one can always deduce that the value of the counter increases at each time step. This is an extremely powerful invariant that, added as assumption to the verification algorithm can be used to certain proofs immediately. However, in the machine-precise semantics, the invariant is not true, because the counter will eventually overflow. Many benchmarks in the Kind2 suite contain counters that could be optimized away with the invariant when assuming the integers in $\mathbb{Z}$.

Comparison against Luke

Luke is a model checker based on SAT and Temporal Induction that was created, to the best of our knowledge, with the main intent of being an educational tool. Yet Luke is extremely robust, lightweight, and it is the only tool we could find that interprets Lustre with the correct machine-precise semantics. The comparison against Luke is on the designs containing Boolean and Integer types, because these are the only ones that are supported by Luke (i.e., reals are not supported). We have run Intrepid and Luke with a timeout of 300 seconds per each benchmark. Intrepid is run with `BMC` and `BR` algorithms in parallel, while Luke is run with BMC and TI (experiments can be reproduced by means of the scripts available at [16]).



Invalid benchmarks [19]



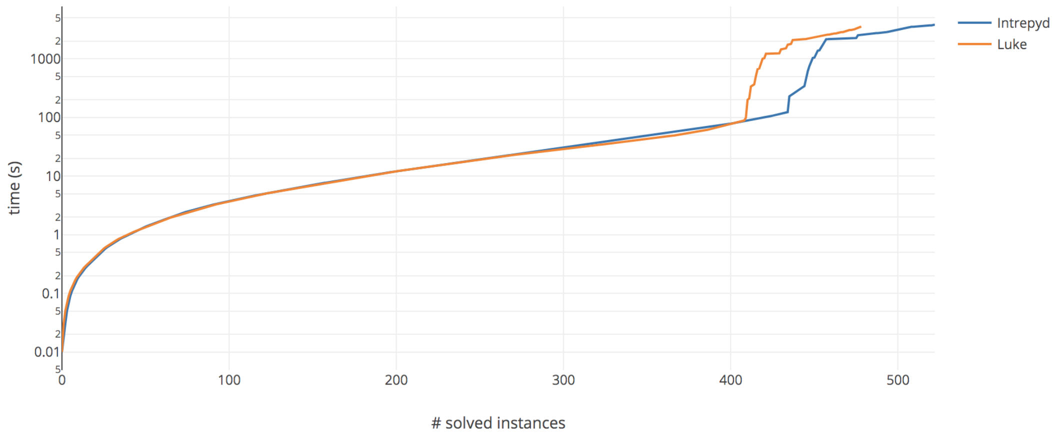
Valid benchmarks [20]

Figure 6: Comparison of Intrepid and Luke on invalid and valid benchmarks

The comparison over the invalid benchmarks is very similar between Intrepid and Luke, with Intrepid being slightly faster on solved instances. This is basically a comparison of two different implementation of a `BMC` algorithm.

The comparison over the valid benchmarks is more interesting because it is comparing an engine based on Temporal Induction (TI) with our own based on `BR`. We notice that `BR` is generally slower on small benchmarks, but overall it can solve more designs than TI. We believe that `BR` can solve those designs that are not provably inductive by TI. This suggests that TI and BR are complementary teqniques.

Overall Intrepid can solve more instances than Luke, as reported by the cumulative plot in Figure 6.1.



Intrepid vs Luke on all benchmarks [2]

Figure 7: Comparison of Intrepid and Luke summarized.

The benchmark suite contains also 48 designs that use the real data-type (`large/ccp*.lus` and `cruise_controller*.lus`). According to the companion description, these benchmarks come from Rockwell-Collins designs. Because Luke does not support the real type, we could not run it. Intrepid instead can process the designs by interpreting reals with single precision floating-point arithmetic. The designs are either very easy or very hard to solve for Intrepid: out of 48 designs only 10 can be proven valid in about 15 seconds overall.

6.2 Simulink/Stateflow via ST

In order to demonstrate the applicability of our translation approach for Simulink/Stateflow we have translated a benchmark from the CocoSim benchmark suite, namely `GPCA_Alarm`, which implements a part of an medical infusion pump mechanism. We have first translated the design in ST using Matlab's PLC toolkit, and then we have used Intrepid's ST frontend to translate the design into an equivalent in python. The design contains 8 properties, and Intrepyd can prove 4 of them to be invalid in about 50 seconds. 14 of which are taken for parsing (on an Intel Core with 4 GB of RAM).

7 Conclusion

Intrepid is a model-checker for Control Engineering and Industrial Automation with a bit-precise interpretation of the design, making it suitable for safety-critical verification tasks. Intrepid can be downloaded and intalled via the python utility `pip` by issuing `pip install intrepyd`. Our preliminary experiments show that Intrepid can be applied for the verification of Lustre, a subset of Structured Text and Simulink/Stateflow designs.

8 Notes

Some of the content of this paper first appeared online in form of blog posts at [10].

References

[1] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. "νZ - An Optimizing SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 2015, pp. 194–199. DOI: 10.1007/978-3-662-46681-0_14. URL: http://dx.doi.org/10.1007/978-3-662-46681-0_14.

[2] Roderick Paul Bloem et al. "Model-Based MCDC Testing of Complex Decisions for the Java Card Applet Firewall". In: *VALID proceedings*. Ed. by IARIA. 2013, pp. 1 –6.

[3] Hamza Bourbouh et al. "Automated analysis of Stateflow models". In: *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*. 2017, pp. 144–161. URL: http://www.easychair.org/publications/paper/340361.

[4] Marco Carloni et al. "Contract Modeling and Verification with FormalSpecs Verifier Tool-Suite - Application to Ansaldo STS Rapid Transit Metro System Use Case". In: *Lecture Notes in Computer Science*. Springer Nature, 2015, pp. 178–189. DOI: 10.1007/978-3-319-24249-1_16. URL: https://doi.org/10.1007%2F978-3-319-24249-1_16.

[5] Roberto Cavada et al. "The nuXmv Symbolic Model Checker". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 2014, pp. 334–342. DOI: 10.1007/978-3-319-08867-9_22. URL: http://dx.doi.org/10.1007/978-3-319-08867-9_22.

[6] Adrien Champion et al. "The Kind 2 Model Checker". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. 2016, pp. 510–517. DOI: 10.1007/978-3-319-41540-6_29. URL: http://dx.doi.org/10.1007/978-3-319-41540-6_29.

[7] John Joseph Chilenski. *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*. Tech. rep. DOT/FAA/AR-01/18. 2-122 Building, Door South 4 10. Work Unit No. (TRAIS) 7701 14th Ave. South Seattle, WA 98108: Boeing Commercial Airplane Group, Apr. 2001.

[8] *CocoSim*. https://coco-team.github.io/cocosim/.

[9] Orlando Ferrante, Alberto Ferrari, and Marco Marazza. "Model based generation of high coverage test suites for embedded systems". In: *19th IEEE European Test Symposium, ETS 2014, Paderborn, Germany, May 26-30, 2014*. 2014, pp. 1–2. DOI: 10.1109/ETS.2014.6847843. URL: http://dx.doi.org/10.1109/ETS.2014.6847843.

[10] *Formal Methods Little Corner*. https://formalmethods.github.io.

[11] Silvio Ghilardi and Silvio Ranise. "MCMT: A Model Checker Modulo Theories". In: *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*. 2010, pp. 22–29. DOI: 10.1007/978-3-642-14203-1_3. URL: http://dx.doi.org/10.1007/978-3-642-14203-1_3.

[12] N. Halbwachs et al. “The synchronous dataflow programming language LUSTRE”. In: *Proceedings of the IEEE*. 1991, pp. 1305–1320.

[13] Kelly J. Hayhurst et al. *A Practical Tutorial on Modified Condition / Decision Coverage*. TM 2001-210876. Langley Research Center, Hampton, Virginia 23681-2199: NASA, May 2001.

[14] IEC. *Programmable controllers - Part 3: Programming languages*. https://webstore.iec.ch/publication/4552.

[15] *Intrepid backend*. https://github.com/formalmethods/intrepid.

[16] *Intrepid vs Luke comparison*. https://github.com/formalmethods/lustreexperiments/tree/master/intrepydvsluke.

[17] *Intrepyd Python API*. https://github.com/formalmethods/intrepyd.

[18] *Kind2 benchmarks*. https://github.com/kind2-mc/kind2-benchmarks.

[19] *Link to interactive data*. https://plot.ly/create/?fid=robertobruttomesso:30#/.

[20] *Link to interactive data*. https://plot.ly/create/?fid=robertobruttomesso:32#/.

[21] *Luke*. https://www.it.uu.se/edu/course/homepage/pins/vt11/lustre.

[22] Mathworks. *Simulink Design Verifier*. https://www.mathworks.com/products/sldesignverifier.html.

[23] Leonardo Mendonça de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. URL: http://dx.doi.org/10.1007/978-3-540-78800-3_24.

[24] RTCA. *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*.