Intrepid: a Scriptable Model Checker for Control Engineering (a preliminary report)

Roberto Bruttomesso

roberto.bruttomesso@gmail.com

Abstract. Intrepid is an SMT-based infinite state model checker library with a rich API that allows standard verification tasks, interaction with the internals of the circuit representation, and incremental multi-objective verification calls. This flexibility, in conjunction with the rapid prototyping framework of the python ecosystem, allows to build applications on top of Intrepid that go beyond the classical "encode, run, decode" cycle provided by traditional model checkers. Intrepid comes with a parser for (the discrete subset of) MathWork's Simulink language, and thus can be used as a free alternative to existing proprietary verification tools.

1 Introduction

Intrepid is an infinite-state model checking library that has been conceived to facilitate the application of formal methods in control engineering and industrial automation applications, by providing a rich API for python that allows a higher degree of interaction between the engineer and model checking algorithms in a rapid prototyping environment. Intrepid relies on the powerful SMT-solver Z3 [16]. The interaction happens with the C API, and thus it is in principle possible to use other SMT-solvers that provide equivalent APIs. A translator from Simulink models (made up of discrete-time blocks) into python scripts, `simulink2py`, comes as part of the distribution, allowing Intrepid to be used as a free and flexible alternative to proprietary model checkers. The translator is stand alone, i.e., it does not require interaction with a Matlab session to be executed.

Supported data types. Intrepid supports the following data types: Booleans, signed and unsigned integers of size 8, 16, 32, and 64 bits, 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].

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).

Model Checking. Intrepid provides two model checking engines, a Bounded Model Checker (`BMC`) and a Backward Reachability (`BR`) engines. 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 [3].

`BR` is an adaption of the exploration algorithm behind the MCMT tool [11], extended to handle input signals[1]. 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[2].

1.1 Organization of the paper

In the remainder of the paper we shall showcase some applications of Intrepid by means of concrete examples. All the examples are part of the tool distribution and can be reproduced by running the companion scripts. The experiments have been perfomed on an Intel Centrino with 8 GBs of main memory, running Windows 10.

[1] However notice that Intrepid's code base has been written from scratch.

[2] Notice that Mathwork's proprietary model checker Simulink Design Verifier [15] cannot handle non-linear arithmetic at the time of writing this paper.

2 Simulink to python

The tool `simulink2py` comes as part of the Intrepid distribution to translate a Simulink design into a python script that can be used to load the circuit into the main memory. The translation from Simulink is almost a 1:1 between the two representations: inputs and outputs map to inputs and outputs creation APIs in Intrepid; blocks (e.g., an and gate) map to corresponding operator creation APIs in Intrepid; subsystems map to a python functions, whose bodies are corresponding APIs calls and other function calls (recursion is not allowed though), assumptions and proof objectives are mapped to Intrepid assumptions and targets. Moreover, the translation generates some dictionaries and comments that can be used to track the correspondence between the original names of the Simulink blocks and the generated python constructs. The translator is built on top of the java library provided by CQSE [9].

2.1 Sudoku Generator Example

To demonstrate the robustness of our translator, we have encoded the Logic behind a sudoku puzzle (standard 9x9 grid) in Simulink. The encoding is (purposely) very na ive, and it consists of 81 8-bits-integer inputs. Each input encodes the value of one cell of the Sudoku grid. Inputs are subjected to the well known rules of Sudoku: each input is between 1 and 9, inputs in each row are all different, inputs in each column are all different, inputs in each 3x3 square (out of the 9 squares) are all different. The toolchain proposes a solution for an empty board using `BMC` in a total time of 6.83 seconds: 2.02 seconds for the translation from Simulink to python, 0.08 seconds for parsing the python encoding, and 4.73 seconds to solve and display the solution.

3 Software Cost Reduction (SCR) Verification

Software Cost Reduction [13] defines a methodology to reprensent reactive systems by means of "event tables" that describe the evolution of the state of a system upon receiving input events. SCR has been successfully employed in safety-critical domains such as avionics and nuclear power plant specifications. We refer the reader to the Appendix A of [2] for a simple example of a SCR specification.

3.1 Implementation of SCR in Simulink and Intrepid

In order to encode SCR specifications in Intrepid, we employ a hybrid approach. The system architecture (inputs, outputs, and the control logic connecting the signals) is encoded in Simulink; the SCR transition logic is not encoded in Simulink (that would require an extremely tedious and error-prone effort, if to be done manually), but it is abstracted with an empty SubSystem block that follows a special naming convention. The SCR transition logic is saved into a

standard comma-separated-value format. `simulink2py` recognizes the naming convention when invoked, and the appropriate code for reading the SCR table is placed in the final python encoding.

3.2 A7E Requirements Verification

The A7E Software Requirements document [1] is an effort of formalizing software requirements specifications for the A7E US Navy Aircraft software. It is the result of many years of adjustments and refinements, and it has been made public by the US Navy as an example of a successful story in documenting requirements formally. The transitions of the A7E requirements come in three SCR tables, for a total of 667 lines. We have encoded the requirements in Simulink and SCR tables, and we have used Intrepid to model check the following safety property reported at page 177 of [1]:

> "The system may only be in *AflyUpd* when it is in either *BOC* or *SBOC*."

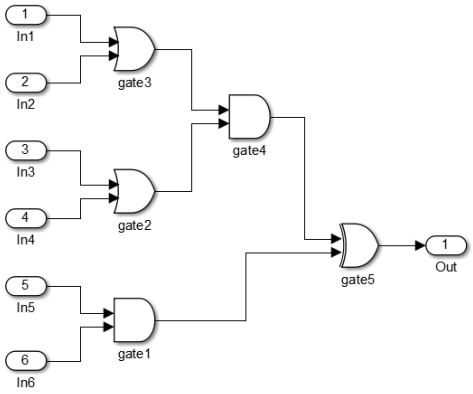
In order to encode the model and the property we have followed the guidelines from [19], however in our encoding we have left abstracted away the logic that is not directly interesting for the aforementioned property. Intrepid can prove the property in 0.35 seconds, plus another 1.67 seconds for the Simlulink translation into python, and 0.35 seconds for parsing the python and the SCR requirements.

4 Automated Test Generation

The avionics standard DO-178C [18] 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 [8] 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 [4] 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 [10] 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.




```
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
```

Fig. 1. An example of an execution of Automated Test Generation on a simple combinational circuit taken from [12]. 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.

5 Related Work

Kind2 [7], nuXmv [6], and MCMT [11] are three other SMT-based infinite state model checkers. Kind2 reads Lustre specifications, and it implements a temporal induction engine with numerous optimizations and enhancements over the basic algorithm. It is probably the most similar tool to our, as both Lustre and Simulink models have closely related semantics.

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.

MCMT was conceived to reason about array-based systems, an expressive class of infinite state systems that allows encoding of protocols. Intrepid's back-

ward 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) [5] is a proprietary tool developed at ALES/UTRC that can verify Simulink designs by means of a translation into nuXmv. In contrast to our, their Simulink translator is based on the MatLab APIs, and thus it requires interaction with MatLab. FSV-ATG [10] is an extension of FSV for Automatic Test Generation. Our approach to ATG presented in earlier sections is inspired to the same algorithm. In contrast, our approach is not based on repeated and monolithic calls to the backend (nuXmv again), but it leverage the richer API provided by Intrepid.

Simulink Design Verifier (`SLDV`) [15] 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.

6 Conclusion

We have presented Intrepid, a model checker for infinite state systems that enables flexible prototyping for formal methods applications by providing a rich python API. The frontend for Simulink allows a smooth application of Intrepid on existing control engineering models, and can be used as a free alternative to other proprietary solutions.

Intrepid is licensed under a BSD schema, thus allowing unrestricted use for individuals and companies. Intrepid is available for cloning as a github project at [17].

References

[1] Thomas A. Alspaugh et al. *Software Requirements for the A-7E Aircraft*. NRL Memorandum Report 3876. Washington, DC: Naval Research Laboratory, Aug. 1992. DOI: 100.2/ADA255746. URL: http://dx.doi.org/100.2/ADA255746.

[2] Ramesh Bharadwaj and Constance L. Heitmeyer. "Model Checking Complete Requirements Specifications Using Abstraction". In: *Automated Software Engineering* 6.1 (1999), pp. 37–68. DOI: 10.1023/a:1008697817793. URL: https://doi.org/10.1023%2Fa%3A1008697817793.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] CQSE. *Simulink library for Java*. https://www.cqse.eu/en/products/simulink-library-for-java/overview/.

[10] 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.

[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] 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.

[13] Constance L. Heitmeyer, Bruce G. Labaw, and Daniel L. Kiskis. “Consistency checking of SCR-style requirements specifications”. In: *Second IEEE International Symposium on Requirements Engineering, March 27 - 29, 1995, York, England*. 1995, pp. 56–65. DOI: 10.1109/ISRE.1995.512546. URL: http://dx.doi.org/10.1109/ISRE.1995.512546.

[14] *IEC 61131-3 Programmable controllers - Part 3: Programming languages*. 2003.

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

[16] 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.

[17] Name removed for blind review. *Intrepid Github Repository*. https://github.com/formalmethods/intrepyd.

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

[19] T. Sreemani. "Feasibility of Model Checking Software Requirements: A Case Study". MA thesis. University of Waterloo, 1996.