Hardware Verification and Decision Procedures

fault: A Python Embedded Domain-Specific Language for Metaprogramming Portable Hardware Verification Components

Lenny Truong(✉), Steven Herbst, Rajsekhar Setaluri, Makai Mann, Ross Daly, Keyi Zhang, Caleb Donovick, Daniel Stanley, Mark Horowitz, Clark Barrett, and Pat Hanrahan

Stanford University, Stanford, CA 94305, USA
lenny@stanford.edu

Abstract. While hardware generators have drastically improved design productivity, they have introduced new challenges for the task of verification. To effectively cover the functionality of a sophisticated generator, verification engineers require tools that provide the flexibility of metaprogramming. However, flexibility alone is not enough; components must also be portable in order to encourage the proliferation of verification libraries as well as enable new methodologies. This paper introduces **fault**, a Python embedded hardware verification language that aims to empower design teams to realize the full potential of generators.

1 Introduction

The new golden age of computer architecture relies on advances in the design and implementation of computer-aided design (CAD) tools that enhance productivity [11,21]. While hardware generators have become much more powerful in recent years, the capabilities of verification tools have not improved at the same pace [12]. This paper introduces **fault**,[1] a domain-specific language (DSL) that aims to enable the construction of flexible and portable verification components, thus helping to realize the full potential of hardware generators.

Using flexible hardware generators [1,16] drastically improves the productivity of the hardware design process, but simultaneously increases verification cost. A *generator* is a program that consumes a set of parameters and produces a hardware module. The scope of the verification task grows with the capabilities of the generator, since more sophisticated generators can produce hardware with varying interfaces and behavior. To reduce the cost of attaining functional coverage of a generator, verification components must be as flexible as their design

[1] https://github.com/leonardt/fault.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 403–414, 2020.
https://doi.org/10.1007/978-3-030-53288-8_19

counterparts. To achieve flexibility, hardware verification languages must provide the metaprogramming facilities found in hardware construction languages [1].

However, flexibility alone is not enough to match the power of generators; verification tools must also enable the construction of portable components. Generators facilitate the development of hardware libraries and promote the integration of components from external sources. Underlying the utility of these libraries is the ability for components to be reused in a diverse set of environments. The dominance of commercial hardware verification tools with strict licensing requirements presents a challenge in the development of portable verification components. To encourage the proliferation of verification libraries, hardware verification languages must design for portability across verification tools. Design for portability will also promote innovation in tools by simplifying the adoption of new technologies, as well as enable new verification methodologies based on unified interfaces to multiple technologies.

This paper presents **fault**, a domain-specific language (DSL) embedded in Python designed to enable the flexible construction of portable verification components. As an embedded DSL, **fault** users can employ all of Python's rich metaprogramming capabilities in the description of verification components. Integration with **magma** [15], a hardware construction language embedded in Python, is an essential feature of **fault** that enables full introspection of the hardware circuit under test. By using a staged metaprogramming architecture, **fault** verification components are portable across a wide variety of open-source and commercial verification tools. A key benefit of this architecture is the ability to provide a unified interface to constrained random and formal verification, enabling engineers to reuse the same component in simulation and model checking environments. **fault** is actively used by academic and industrial teams to verify digital, mixed-signal, and analog designs for use in research and production chips. This paper demonstrates **fault**'s capabilities by evaluating the runtime performance of different tools on a variety of applications ranging in complexity from unit tests of a single module to integration tests of a complex design. These experiments leverage **fault**'s portability by reusing the same source input across separate trials for each target tool.

2 Design

We had three goals in designing **fault**: enable the construction of flexible test components through metaprogramming, provide portable abstractions that allow test component reuse across multiple target environments, and support direct integration with standard programming language features. The ability to metaprogram test components is a vital requirement for scaling verification efforts to cover the space of functionality utilized by hardware generators. Portability widens the target audience of a reusable component and enhances a design team's productivity by enabling simple migration to different technologies. Integration with a programming language enables design teams to leverage standard software patterns for reuse as well as feature-rich test automation frameworks.

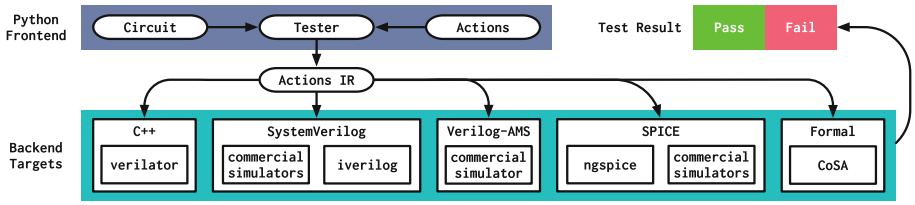


Fig. 1. Architectural overview of the **fault** testing system. In a Python program, the user constructs a `Tester` object with a **magma** `Circuit` and records a sequence of test `Actions`. The compiler uses the action sequence as an intermediate representation (IR). Backend targets lower the actions IR into a format compatible with the corresponding tool and provide an API to run the test and report the results.

Figure 1 provides an overview of the system architecture. **fault** is a DSL embedded in Python, a prolific dynamic language with rich support for metaprogramming and a large ecosystem of libraries. **fault** is designed to work with **magma** [15], a Python embedded hardware construction language which represents circuits as introspectable Python objects containing ports, connections, and instances of other circuits. While **fault** and **magma** separate the concerns of design and verification into separate DSLs, they are embedded in the same host language for simple interoperability. This multi-language design avoids the complexity of specifying and implementing a single general purpose language without sacrificing the benefits of tightly integrating design and verification code.

To construct **fault** test components, the user first instantiates a `Tester` object with a **magma** circuit as an argument. The user then records a sequence of test actions using an API provided by the `Tester` class. Here is an example of constructing a test for a 16-bit `Add` circuit:

```
tester = Tester(Add16)
tester.poke(Add16.in0, 3)
tester.poke(Add16.in1, 2)
tester.eval()
tester.expect(Add16.out, 5)
```

The `poke` action (method) sets an input value, the `eval` action triggers evaluation of the circuit (the effects of `poke` actions are not propagated until an `eval` action occurs), and the `expect` action asserts the value of an output. Attributes of the `Add16` object refer to circuit ports by name.

fault's design is based on the concept of staged metaprogramming [20]; the user writes a program that constructs another program to be executed in a subsequent stage. In **fault**, the first stage executes Python code to construct a test specification; the second stage invokes a target runtime that executes this specification. To run the test for the 16-bit `Add`, the user simply calls a method and provides the desired target:

```
tester.compile_and_run("verilator")
tester.compile_and_run("system-verilog", simulator="iverilog")
```

By applying staged metaprogramming, **fault** allows the user to leverage the full capabilities of the Python host language in the programmatic construction of test components. For example, a test can use a native `for` loop to construct a sequence of actions using the built-in random number library and integer type:

```
for _ in range(32):
   N = (1 << 16) - 1
   in0, in1 = random.randint(0, N), random.randint(0, N)
   tester.poke(Add16.in0, in0)
   tester.poke(Add16.in1, in1)
   tester.eval()
   tester.expect(Add16.out, (in0 + in1) & N)
```

Python `for` loops are executed during the first stage of computation and are effectively "unrolled" into a flat sequence of actions. Other control structures such as `while` loops, `if` statements, and function calls are handled similarly.

Python's object introspection capabilities greatly enhance the flexibility of **fault** tests. For example, the core logic of the above test can be generalized to support an arbitrary width `Add` circuit by inspecting the interface:

```
# compute max value based on port width (length)
N = (1 << len(Add.in0)) - 1
in0, in1 = random.randint(0, N), random.randint(0, N)
tester.poke(Add.in0, in0)
tester.poke(Add.in1, in1)
tester.eval()
tester.expect(Add.out, (in0 + in1) & N)
```

This ability to metaprogram components as a function of the design under test is an essential aspect of **fault**'s design. It allows the construction of generic components that can be reused across designs with varying interfaces and behavior.

fault's embedding in Python's class system provides an opportunity for reuse through inheritance. For example, a design team could subclass the generic `Tester` class and add a new method to perform an asynchronous reset sequence:

```
class ResetTester(Tester):
   def __init__(self, circuit, clock, reset_port):
      super().__init__(self, circuit, clock)
      self.reset_port = reset_port

   def reset(self):
      # asynchronous reset, negative edge
      self.poke(self.reset_port, 1)
      self.eval()
      self.poke(self.reset_port, 0)
      self.eval()
      self.poke(self.reset_port, 1)
      self.eval()
```

Combining inheritance with introspection, we can augment the the `ResetTester` to automatically discover the reset port by inspecting port types:

```
class AutoResetTester(ResetTester):
    def __init__(self, circuit, clock):
        # iterate over interface to find reset (assumes exactly one)
        for port in circuit.interface.ports.values():
            if isinstance(port, AsyncResetN):
                reset_port = port
        super().__init__(self, circuit, clock, reset_port)
```

2.1 Frontend: Tester API

fault's Python embedding is implemented by the `Tester` class which provides various interfaces for recording test actions as well as methods for compiling and running tests using a specific target. By using Python's class system to perform a shallow embedding [5], **fault** avoids the complexity of processing abstract syntax trees and simply uses Python's standard execution to construct test components. As a result, programming in **fault** is much like programming with a standard Python library. This design choice reduces the overhead of learning the DSL and simplifies aspects of implementation such as error messages, but comes at the cost of limited capabilities for describing control flow. The **fault** frontend described in this paper focuses on implementation simplicity, but the system is designed to be easily extended with new frontends using alternative embeddings.

Action Methods. The `Tester` class provides a low-level interface for recording actions using methods. The basic action methods are `poke` (set a port to a value), `expect` (assert a port equals a value), `step` (invert the value of the clock), `peek` (read the value of a port), and `eval` (evaluate the circuit). The `peek` method returns an object containing a reference to the value of a circuit port in the current simulation state. Using logical and arithmetic operators, the user can construct expressions with this object and pass the result to other actions. For example, to expect that the value of the port `O0` is equal to the inverse of the value of port `O1`, the user would write `tester.expect(circuit.O0, ~tester.peek(circuit.O1))`. The `Tester` provides a `print` action to display simulation runtime information included the peeked values.

Metaprogramming Control Flow. Notably absent from the basic method interface described above are control flow abstractions. As noted before, standard Python control structures such as loops and `if` statements are executed in the first stage of computation as part of the metaprogram. However, there are cases where the user intends to preserve the control structure in the generated code, such as long-running loops that should not be unrolled at compile time or loops that are conditioned on dynamic values from the circuit state. For example, consider a `while` loop that executes until it receives a ready signal:

```
# Construct while loop conditioned on circuit.ready.
loop = tester._while(tester.peek(circuit.ready))
loop.expect(circuit.ready, 0) # executes inside loop
loop.step(2)                  # executes inside loop
# Check final state after loop has exited
tester.expect(circuit.count, expected_cycle_count)
```

This logic could not be encoded in the metaprogram, because the metaprogram is evaluated before the test is run, and thus does not know anything about the runtime state of the circuit. To capture this dynamic control flow, the `Tester` provides methods for inserting `if-else` statements, `for` loops, and `while` loops. Each of these methods returns a new instance of the current `Tester` object which provides the same API, allowing the user to record actions corresponding to the body of the control construct. The `Tester` class provides convenience functions for using these control structures to generate common patterns, such as `wait_on`, `wait_until_low`, and `wait_until_posedge`.

Attribute Interface. While the low-level method interface is useful for writing complex metaprograms, simple components are rather verbose to construct. To simplify the handling of basic actions like `poke` and `peek`, the `Tester` object exposes an interface for referring to circuit ports and internal signals using Python's object attribute syntax. For example, to poke the input port `I` of a circuit with value `1`, one would write `tester.circuit.I = 1`. This interface supports referring to internal signals using a hierarchical syntax. For example, referring to port `Q` of an instance `ff` can be done with `tester.circuit.ff.Q`.

Assume/Guarantee. The `Tester` object provides methods for specifying assumptions and guarantees that are abstracted over constrained random and formal model checking runtime environments. An *assumption* is a constraint on input values, and a *guarantee* is an assertion on output values. Assumptions and guarantees are specified using Python `lambda` functions that return symbolic expressions referring to the input and output and ports of a circuit. For example, the guarantee `lambda a, b, c: (c >= a) and (c >= b)` states that the output `c` is always greater than or equal to the inputs `a` and `b`. Here is an example of verifying a simple ALU using the assume/guarantee interface:

```
# Configuration sequence for opcode register
tester.circuit.opcode_en = 1
tester.circuit.opcode = 0 # opcode for add (+)
tester.step(2)
tester.circuit.opcode_en = 0
tester.step(2)
# Verify add does not overflow
tester.circuit.a.assume(lambda a: a < BitVector[16](32768))
tester.circuit.b.assume(lambda b: b < BitVector[16](32768))
tester.circuit.c.guarantee(
   lambda a, b, c: (c >= a) and (c >= b)
)
```

Note that this example demonstrates the use of `poke` and `step` to initialize circuits not only for constrained random testing, but also for formal verification.

2.2 Actions IR

In using the `Tester` API, users construct a sequence of `Action` objects that are used as an intermediate representation (IR) for the compiler. Basic port action

objects, such as `Poke` and `Expect`, simply store references to ports and values. Control flow action objects, such as `While` and `If`, contain sub-sequences of actions, resulting in a hierarchical data-structure similar to an abstract syntax tree. This view of the compiler internals reveals that the metaphor of *recording actions* is really an abstraction over the construction of program fragments.

2.3 Backend Targets

fault supports a variety of open-source and commercial backend targets for running tests. A target is responsible for consuming an action sequence, compiling it into a format compatible with the target runtime, and providing an API for invoking the runtime. Targets must also report the result of the test either by reading the exit code of running the process or processing the test output.

Verilog Simulation Targets. The **fault** compiler includes support for the open-source Verilog simulators **verilator** [17] and **iverilog** [22], plus three commercial simulators. To compile **fault** programs to a **verilator** test bench, the backend lowers the action sequence into a C++ program that interacts with the software simulation object produced by the **verilator** compiler. For **iverilog** and the commercial simulators, the backend lowers the action sequence into a SystemVerilog test bench that interacts with the test circuit through an `initial` block inside the top-level module. One useful aspect of the SystemVerilog backend is its handling of variations in the feature support of target simulators. For example, the commercial simulators use different commands for enabling waveform tracing and **iverilog** uses a non-standard API for interacting with files. Constrained random inputs are generated using rejection or SMT [9] sampling.

CoSA. The CoreIR Symbolic Analyzer (CoSA) is a solver-agnostic SMT-based hardware model checker [13]. **fault**'s CoSA target relies on **magma**'s ability to compile Python circuit descriptions to CoreIR [8], a hardware intermediate representation. CoreIR's formal semantics are based on finite-state machines and the SMT theory of fixed-size bitvectors [3]. **fault** action sequences are lowered into CoSA's custom explicit transition system format (ETS) and combined with the CoreIR representation of the circuit to produce a model. CoSA allows the user to specify assumptions and properties, providing a straightforward lowering of **fault** assumptions and guarantees.

SPICE. In addition to being able to test designs with Verilog simulators, **fault** supports analog and mixed-signal simulators. Compared to the traditional approach of maintaining separate implementations for digital and analog tests, this is a significantly easier way to write tests for mixed-signal circuits. Basic actions such as `poke` and `expect` are supported in the SPICE simulation mode, but they are implemented quite differently than they are in Verilog-based tests. Rather than emitting a sequential list of actions in an `initial` block, **fault**

compiles `poke` actions into piecewise-linear (PWL) waveforms. Other actions, such as `expect`, are implemented by post-processing the simulation data.

Verilog-AMS. For designs containing a mixture of SPICE and Verilog blocks, **fault** supports testing with a Verilog-AMS simulator. This mode is more similar to running SystemVerilog-based tests than SPICE-based tests. In particular, the test bench is implemented using a top-level SystemVerilog module, meaning that a wide range of actions are supported including loops and conditionals. This is a key benefit of using a Verilog-AMS simulator as opposed to a SPICE simulator.

3 Evaluation

To demonstrate **fault**'s capabilities, we evaluate the runtime performance of four different testing tasks from the domain of hardware verification. Each task highlights the utility of **fault**'s portability by reusing the same source input across separate trials of different targets. Due to licensing restrictions, we omit the name of the commercial simulators and replace them with a generic name. The code to reproduce these experiments is available in the artifact.[2] Each experiment involves at least one open-source simulator, but reproducing all the results requires access to commercial simulators.

CGRA Processing Element Unit Tests. To demonstrate the capability of **fault** as a tool for writing portable tests for digital verification, Fig. 2 reports the runtime performance of a subset of the **lassen** test suite. **lassen** [19] is an open-source implementation of a CGRA processing element that contains a large suite of unit tests using **fault**. Interestingly, we see comparable performance between **verilator** and **commercial simulator 1**, while **commercial simulator 2** is consistently ∼5x slower than the others. One important property of the **lassen** test suite is that it generates a new test bench for each operation and input/output pair. This stresses a simulator's ability to efficiently handle incremental changes, since each invocation involves a new top-level test bench file, but an unchanged design under test.

Test	verilator	commercial sim 1	commercial sim 2
`test_unsigned_binary`	94.483	88.700	519.079
`test_smult`	31.439	28.668	170.115
`test_fp_binary_op`	104.117	91.878	571.759
`test_stall`	10.424	9.629	56.458

Fig. 2. Runtime (s) for unit tests of a CGRA processing element collected with a VM running on an Intel(R) Xeon(R) Silver 4214 CPU @ 2.20 GHz with 256 GB of RAM.

[2] https://github.com/leonardt/fault_artifact/blob/master/README.md.

SRAM Array. To demonstrate the capability of **fault** as a tool for writing portable tests for analog and mixed-signal verification, we used **OpenRAM** to generate a 16x16 SRAM and then ran a randomized readback test of the design with SPICE, Verilog-AMS, and SystemVerilog simulators. **OpenRAM** [10] is an open-source memory compiler that produces a SPICE netlist and Verilog model.

The results shown in Fig. 3a reveal two interesting trends. First, as expected, SPICE simulations of the array were significantly slower than Verilog simulations (100-1000x). Since **fault** allows the user to prototype tests with fast Verilog simulations, and then seamlessly switch to SPICE for signoff verification, our tool may reduce the latency in developing mixed-signal tests by orders of magnitude. Second, even for simulations of the same type, there was significant variation in the runtime of different simulators. SPICE simulation time varied by about 2x, while Verilog simulation time varied by about 10x. One of the advantages of using **fault** is that it is easy to switch between simulators to find the one that works best for a particular scenario.

Target	Simulator	Runtime (s)
spice	ngspice	117.660
spice	comm sim 1	199.868
spice	comm sim 2	98.043
system-verilog	iverilog	0.238
system-verilog	comm sim 1	1.081
system-verilog	comm sim 2	2.807
verilog-ams	comm sim 1	228.405

(a) Runtime using a VM on an Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz with 64GB of RAM.

Lines of Code (LoC)	
fault	136
Handwritten SPICE	223
Handwritten SystemVerilog and Verilog-AMS	189

(b) LoC for **fault** and language-specific implementations of the test.

Fig. 3. Results for OpenRAM 16x16 SRAM randomized readback test.

We also looked at the amount of human effort required to use **fault** to implement this test as compared to the traditional approach of writing separate testbenches for each simulation language. Since "human effort" is subjective, we used lines of code as a rough metric, as measured from handwritten implementations of the same test in SystemVerilog, Verilog-AMS, and SPICE. Figure 3b shows the results of this experiment: the **fault**-based approach used 136 LoC as compared to 412 LoC for the traditional approach, a reduction of 3.02x.

CGRA Integration Test Bench. To observe how **fault** scales to more complex testing tasks, we report numbers for an integration test of the Stanford Garnet CGRA [18]. This test generates an instance of the CGRA chip, runs a simulation that programs the chip for an image processing application, streams the input image data onto the chip, and streams the output image data to a file. The output is compared to a reference software model. Running the test

took 232 min with the **verilator** target, 185 min with **commercial simulator 1**, and 221 min with **commercial simulator 2**. Leveraging the portability of **fault**-based tests could save up to 47 min in testing time. These results were collected using the same machine as the SRAM experiment (see Fig. 3a).

Unified Constrained Random and Formal. To demonstrate the utility of the assume/guarantee interface as a unified abstraction for constrained random and formal verification, we compared the runtime performance of using a constrained random target versus a formal model checker to verify the simple ALU property shown in Sect. 2.1. The first test evaluated the runtime performance of verifying correctness of the property on 100 constrained random inputs versus using a formal model checker. The formal model checker provided a complete proof of correctness using interpolation-based model checking [14] in 1.613 s, while constrained random verified 100 samples in 2.269 s (rejection sampling) and 2.799 s (SMT sampling). The second test injected a bug into the ALU by swapping the opcodes for addition and subtraction. The model checker found a counterexample in 1.154 s with bounded model checking [4], while constrained random failed in 2.947 s (rejection sampling) and 1.230 s (SMT sampling). In both cases the model checker was at least as fast as the constrained random equivalent while providing better coverage in the case of no bug. These results were collected using a MacBook Pro (13-in 2017, 4 Thunderbolt, macOS 10.15.2), with a 3.5 GHz Dual-Core Intel i7 CPU, and 16 GB RAM.

4 Related Work

Prior work has leveraged using a generic API to Verilog simulators to build portability into testing infrastructures. The **ChiselTest** library [2] and **cocotb** [7] provide this capability for Scala and Python respectively. Using a generic API offers many of the same advantages with regards to test portability, simplicity, and automation, but the lack of multi-stage execution limits the application to more diverse backend targets such as SPICE simulations and formal model checkers. However, because these libraries interact with the simulator directly, they do allow user code to immediately respond to the simulator state, enabling interactive debugging through the host language. **cocotb** also presents a coroutine abstraction that naturally models the concurrency found in hardware simulation. Future work could investigate using cocotb as a runtime target for **fault**'s frontend, enabling a similar concurrent, interactive style of testing. Another interesting avenue of work would be to extend **fault**'s backend targets to support lowering **cocotb**'s coroutine abstraction.

5 Conclusion

The ethos of **fault** is to enable the construction of flexible, portable test components that are simple to integrate and scale for testing complex applications.

The ability to metaprogram test components is essential for enabling verification teams to match the productivity of design teams using generators. **fault**'s portability enables teams to easily transition to different tools for different use cases, and enables the proliferation of reusable verification libraries that are applicable in a diverse set of tooling environments.

While **fault** has already demonstrated utility to design teams in academia and industry, there remains a bright future filled with opportunity to improve the system. Extending the assume/guarantee interface to support temporal properties/constraints and leverage compositional reasoning [6] is essential for scaling the approach to more complex systems. Adding concurrent programming abstractions such as coroutines are essential for capturing the common patterns used in the testing of parallel hardware. Using a deep embedding architecture could significantly improve the performance of generating **fault** test benches.

Funding. The authors would like to thank the DARPA DSSoC (FA8650-18-2-7861) and POSH (FA8650-18-2-7854) programs, the Stanford AHA and SystemX affiliates, Intel's Agile ISTC, the Hertz Foundation Fellowship, and the Stanford Graduate Fellowship for supporting this work.

References

1. Bachrach, J., et al.: Chisel: constructing hardware in a scala embedded language. In: 2012 DAC Design Automation Conference, pp. 1212–1221, June 2012. https://doi.org/10.1145/2228360.2228584
2. ucb bar: chisel-testers2 (2019). https://github.com/ucb-bar/chisel-testers2
3. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org(2016)
4. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
5. Boulton, R.J., Gordon, A., Gordon, M.J.C., Harrison, J., Herbert, J., Tassel, J.V.: Experience with embedding hardware description languages in HOL. In: Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, pp. 129–156. North-Holland Publishing Co., NLD (1992)
6. Clarke, E.M., Long, D.E., McMillan, K.L.: Compositional model checking. In: LICS, pp. 353–362. IEEE Computer Society (1989)
7. cocotb: cocotb (2019). https://github.com/cocotb/cocotb
8. Daly, R.: CoreIR: A simple LLVM-style hardware compiler (2017). https://github.com/rdaly525/coreir
9. Dutra, R., Bachrach, J., Sen, K.: SMTsampler: efficient stimulus generation from complex SMT constraints. In: 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1–8. IEEE (2018)
10. Guthaus, M.R., Stine, J.E., Ataei, S., Chen, B., Wu, B., Sarwar, M.: OpenRAM: an open-source memory compiler. In: 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1–6, November 2016. https://doi.org/10.1145/2966986.2980098

11. Hennessy, J.L., Patterson, D.A.: A new golden age for computer architecture. Commun. ACM **62**(2), 48–60 (2019). https://doi.org/10.1145/3282307
12. Lockhart, D., et al.: Experiences building edge TPU with chisel. In: 2018 Chisel Community Conference (CCC) (2018)
13. Mattarei, C., Mann, M., Barrett, C., Daly, R.G., Huff, D., Hanrahan, P.: CoSA: integrated verification for agile hardware design. In: 2018 Formal Methods in Computer Aided Design (FMCAD), pp. 1–5, October 2018. https://doi.org/10.23919/FMCAD.2018.8603014
14. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_1
15. phanrahan: magma (2019). https://github.com/phanrahan/magma (2019)
16. Shacham, O., Azizi, O., Wachs, M., Richardson, S., Horowitz, M.: Rethinking digital design: why design must change. IEEE Micro **30**(6), 9–24 (2010). https://doi.org/10.1109/MM.2010.81
17. Snyder, W.: Verilator and systemperl. In: North American SystemC Users' Group, Design Automation Conference (2004)
18. StanfordAHA: Garnetflow (2019). https://github.com/StanfordAHA/GarnetFlow
19. StanfordAHA: lassen (2019). https://github.com/StanfordAHA/lassen
20. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theor. Comput. Sci. **248**(1–2), 211–242 (2000)
21. Truong, L., Hanrahan, P.: A golden age of hardware description languages: applying programming language techniques to improve design productivity. In: Lerner, B.S., Bodík, R., Krishnamurthi, S. (eds.) 3rd Summit on Advances in Programming Languages, SNAPL 2019, 16–17 May 2019, Providence, RI, USA. LIPIcs, vol. 136, pp. 7:1–7:21. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2019). https://doi.org/10.4230/LIPIcs.SNAPL.2019.7
22. Williams, S.: Icarus verilog (2006). http://iverilog.icarus.com

Nonlinear Craig Interpolant Generation

Ting Gan[1], Bican Xia[2(✉)], Bai Xue[3,4], Naijun Zhan[3,4(✉)], and Liyun Dai[5]

[1] School of Computer Science, Wuhan University, Wuhan, China
ganting@whu.edu.cn
[2] LMAM, School of Mathematical Sciences, Peking University, Beijing, China
xbc@math.pku.edu.cn
[3] State Key Laboratory of Computer Science, Institute of Software, CAS, Beijing, China
{xuebai,znj}@ios.ac.cn
[4] University of Chinese Academy of Sciences, Beijing, China
[5] RISE, School of Computer and Information Science, Southwest University, Chongqing, China
dailiyun@swu.edu.cn

Abstract. Craig interpolant generation for non-linear theory and its combination with other theories are still in infancy, although interpolation-based techniques have become popular in the verification of programs and hybrid systems where non-linear expressions are very common. In this paper, we first prove that a polynomial interpolant of the form $h(\mathbf{x}) > 0$ exists for two mutually contradictory polynomial formulas $\phi(\mathbf{x}, \mathbf{y})$ and $\psi(\mathbf{x}, \mathbf{z})$, with the form $f_1 \geq 0 \wedge \cdots \wedge f_n \geq 0$, where f_i are polynomials in $\mathbf{x}, \mathbf{y}$ or $\mathbf{x}, \mathbf{z}$, and the quadratic module generated by f_i is Archimedean. Then, we show that synthesizing such interpolant can be reduced to solving a semi-definite programming problem (SDP). In addition, we propose a verification approach to assure the validity of the synthesized interpolant and consequently avoid the unsoundness caused by numerical error in SDP solving. Besides, we discuss how to generalize our approach to general semi-algebraic formulas. Finally, as an application, we demonstrate how to apply our approach to invariant generation in program verification.

Keywords: Craig interpolant · Archimedean condition · Semi-definite programming · Program verification · Sum of squares

1 Introduction

Interpolation-based techniques have become popular in recent years because of their inherently modular and local reasoning, which can scale up existing formal verification techniques like theorem proving, model-checking, abstract interpretation, and so on, while the scalability is the bottleneck of these techniques. The study of interpolation was pioneered by Krajíček [20] and Pudlák [30] in connection with theorem proving, by McMillan in connection with model-checking

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 415–438, 2020.
https://doi.org/10.1007/978-3-030-53288-8_20

[25], by Graf and Saïdi [14], Henzinger *et al.* [16] and McMillan [26] in connection with abstraction like CEGAR, by Wang *et al.* [17] in connection with machine-learning based program verification.

Craig interpolant generation plays a central role in interpolation-based techniques, and therefore has drawn increasing attention. In the literature, there are various efficient algorithms proposed for automatically synthesizing interpolants for decidable fragments of first-order logic, linear arithmetic, array logic, equality logic with uninterpreted functions (EUF), etc., and their combinations, and their use in verification, e.g., [6,16,18,19,26,27,33,33,37] and the references therein. Additionally, how to compare the strength of different interpolants is investigated in [9]. However, interpolant generation for non-linear theory and its combination with the aforementioned theories is still in infancy, although non-linear polynomials inequalities are quite common in safety-critical software and embedded systems [38,39].

In [7], Dai *et al.* had a first try and gave an algorithm for generating interpolants for conjunctions of mutually contradictory nonlinear polynomial inequalities based on the existence of a witness guaranteed by Stengle's Positivstellensatz [36], which is computable using semi-definite programming (SDP). Their algorithm is incomplete in general but if all variables are bounded (called Archimedean condition), then it becomes complete. A major limitation of their work is that two mutually contradictory formulas ϕ and ψ must have the same set of variables. In [10], Gan *et al.* proposed an algorithm to generate interpolants for quadratic polynomial inequalities. The basic idea is based on the insight that for analyzing the solution space of concave quadratic polynomial inequalities, it suffices to linearize them by proving a generalization of Motzkin's transposition theorem for concave quadratic polynomial inequalities. Moreover, they also discussed how to generate interpolants for the combination of the theory of quadratic concave polynomial inequalities and *EUF* based on the hierarchical calculus proposed in [34] and used in [33]. Obviously, *quadratic concave* polynomial inequalities is a very restrictive class of polynomial formulas, although most of existing abstract domains fall within it as argued in [10]. Meanwhile, in [13], Gao and Zufferey presented an approach to extract interpolants for non-linear formulas possibly containing transcendental functions and differential equations from proofs of unsatisfiability generated by δ-decision procedure [12] based on interval constraint propagation (ICP) [1] by transforming proof traces from δ-complete decision procedures into interpolants that consist of Boolean combinations of linear constraints. Thus, their approach can only find the interpolants between two formulas whenever their conjunction is not δ-satisfiable. Similar idea was also reported in [21]. In [5], Chen *et al.* proposed an approach for synthesizing non-linear interpolants based on counterexample-guided and machine-learning, but it relies on quantifier elimination in order to guarantee the completeness and convergence, which gives rise to the low efficiency of their approach theoretically. In [35], Srikanth *et al.* presented an approach called *CAMPY* to exploit non-linear interpolant generation, which is achieved by abstracting non-linear formulas (possibly with non-polynomial expressions)

to the theory of linear arithmetic with uninterpreted functions, i.e., EUFLIA, to prove and/or disprove if a given program satisfies a given property, that may contain nonlinear expressions.

Example 1. In order to compare the approach proposed in this paper and the ones aforementioned, consider

$$\phi = -2xy^2 + x^2 - 3xz - y^2 - yz + z^2 - 1 \geq 0 \wedge 100 - x^2 - y^2 \geq 0 \wedge$$
$$x^2z^2 + y^2z^2 - x^2 - y^2 + \frac{1}{6}(x^4 + 2x^2y^2 + y^4) - \frac{1}{120}(x^6 + y^6) - 4 \leq 0;$$
$$\psi = 4(x-y)^4 + (x+y)^2 + w^2 - 133.097 \leq 0 \wedge 100(x+y)^2 - w^2(x-y)^4 - 3000 \geq 0.$$

It can be checked that $\phi \wedge \psi \models \bot$.

Obviously, synthesizing interpolants for ϕ and ψ in this example is beyond the ability of the above approaches reported in [7,10]. Using the method in [13] implemented in dReal3 it would return "SAT" with $\delta = 0.001$, i.e., $\phi \wedge \psi$ is δ-satisfiable, and hence it cannot synthesize any interpolant using [12]'s approach with any precision greater than 0.001[1]. While, using our method, an interpolant $h > 0$ with degree 10 can be found as shown in Fig. 1[2]. Additionally, using the symbolic procedure REDUCE, it can be proved that $h > 0$ is indeed an interpolant of ϕ and ψ.

In this paper, we investigate this issue and consider how to synthesize an interpolant for two polynomial formulas $\phi(\mathbf{x}, \mathbf{y})$ and $\psi(\mathbf{x}, \mathbf{z})$ with $\phi(\mathbf{x}, \mathbf{y}) \wedge \psi(\mathbf{x}, \mathbf{z}) \models \bot$, where

$$\phi(\mathbf{x}, \mathbf{y}) : f_1(\mathbf{x}, \mathbf{y}) \geq 0 \wedge \cdots \wedge f_m(\mathbf{x}, \mathbf{y}) \geq 0,$$
$$\psi(\mathbf{x}, \mathbf{z}) : g_1(\mathbf{x}, \mathbf{z}) \geq 0 \wedge \cdots \wedge g_n(\mathbf{x}, \mathbf{z}) \geq 0,$$

$\mathbf{x} \in \mathbb{R}^r$, $\mathbf{y} \in \mathbb{R}^s$, $\mathbf{z} \in \mathbb{R}^t$ are variable vectors, $r, s, t \in \mathbb{N}$, and $f_1, \ldots, f_m, g_1, \ldots, g_n$ are polynomials. In addition, $\mathcal{M}_{\mathbf{x},\mathbf{y}}\{f_1(\mathbf{x}, \mathbf{y}), \ldots, f_m(\mathbf{x}, \mathbf{y})\}$ and $\mathcal{M}_{\mathbf{x},\mathbf{z}}\{g_1(\mathbf{x}, \mathbf{z}), \ldots, g_n(\mathbf{x}, \mathbf{z})\}$ are two Archimedean quadratic modules. Here we allow uncommon variables, that are not allowed in [7], and drop the constraint that polynomials must be concave and quadratic, which is assumed in [10]. The Archimedean condition amounts to that all the variables are bounded, which is reasonable in program verification, as only bounded numbers can be represented in computer in practice. We first prove that there exists a polynomial

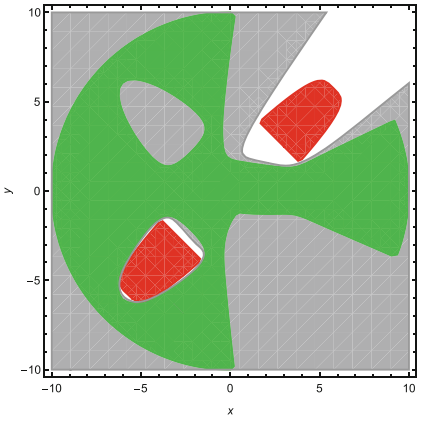


Fig. 1. Example 1. (Green region: the projection of $\phi(x, y, z)$ onto x and y; red region: the projection of $\psi(x, y, w)$ onto x and y; gray region plus the green region: the synthesized interpolant $\{(x, y) \mid h(x, y) > 0\}$.) (Color figure online)

[1] Alternatively, if we try the formula with the latest version of dReal4, it does not produce any output after 20 h.

[2] The mathematical representation of h is given in the full version [11].

$h(\mathbf{x})$ such that $h(\mathbf{x}) = 0$ separates the state space of $\mathbf{x}$ defined by $\phi(\mathbf{x}, \mathbf{y})$ from the one defined by $\psi(\mathbf{x}, \mathbf{z})$ theoretically, and then propose an algorithm to compute such $h(\mathbf{x})$ based on SDP. Furthermore, we propose a verification approach to assure the validity of the synthesized interpolant and consequently avoid the unsoundness caused by numerical error in SDP solving. Finally, we also discuss how to extend our results to general semi-algebraic constraints.

Another contribution of this paper is that as an application, we illustrate how to apply our approach to invariant generation in program verification by revising Lin *et al.*'s framework proposed in [22] for invariant generation based on *weakest precondition*, *strongest postcondition* and *interpolation* by allowing to generate nonlinear invariants.

The paper is organized as follows. Some preliminaries and the problem of interest are introduced in Sect. 2. Section 3 shows the existence of an interpolant for two mutually contradictory polynomial formulas only containing conjunction, and Sect. 4 presents SDP-based methods to compute it. In Sect. 5, we discuss how to avoid unsoundness caused by numerical error in SDP. Section 6 extends our approach to general polynomial formulas. Section 7 demonstrates how to apply our approach to invariant generation in program verification. We conclude this paper in Sect. 8.

2 Preliminaries

In this section, we first give a brief introduction on some notions used throughout this paper and then describe the problem of interest.

2.1 Quadratic Module

$\mathbb{N}$, $\mathbb{Q}$ and $\mathbb{R}$ are the sets of integers, rational numbers and real numbers, respectively. $\mathbb{Q}[\mathbf{x}]$ and $\mathbb{R}[\mathbf{x}]$ denotes the polynomial ring over rational numbers and real numbers in $r \geq 1$ indeterminates $\mathbf{x} : (x_1, \ldots, x_r)$. We use $\mathbb{R}[\mathbf{x}]^2 := \{p^2 \mid p \in \mathbb{R}[\mathbf{x}]\}$ for the set of squares and $\sum \mathbb{R}[\mathbf{x}]^2$ for the set of sums of squares of polynomials in $\mathbf{x}$. Vectors are denoted by boldface letters. $\bot$ and $\top$ stand for **false** and **true**, respectively.

Definition 1 (Quadratic Module [24]). *A subset $\mathcal{M}$ of $\mathbb{R}[\mathbf{x}]$ is called a* quadratic module *if it contains 1 and is closed under addition and multiplication with squares, i.e., $1 \in \mathcal{M}, \mathcal{M} + \mathcal{M} \subseteq \mathcal{M}$, and $p^2\mathcal{M} \subseteq \mathcal{M}$ for all $p \in \mathbb{R}[\mathbf{x}]$.*

Let $\overline{p} := \{p_1, \ldots, p_s\}$ be a finite subset of $\mathbb{R}[\mathbf{x}]$, the quadratic module $\mathcal{M}_{\mathbf{x}}(\overline{p})$ or simply $\mathcal{M}(\overline{p})$ generated by $\overline{p}$ (i.e. the smallest quadratic module containing all p_is) is $\mathcal{M}_{\mathbf{x}}(\overline{p}) = \{\sum_{i=0}^{s} \delta_i p_i \mid \delta_i \in \sum \mathbb{R}[\mathbf{x}]^2\}$, where $p_0 = 1$.

Archimedean condition plays a key role in the study of polynomial optimization.

Definition 2 (Archimedean). *Let $\mathcal{M}$ be a quadratic module of $\mathbb{R}[\mathbf{x}]$. $\mathcal{M}$ is said to be* Archimedean *if there exists some $a > 0$ such that $a - \sum_{i=1}^{r} x_i^2 \in \mathcal{M}$.*

2.2 Problem Description

Craig showed that given two formulas ϕ and ψ in a first-order theory $\mathcal{T}$, if $\phi \models \psi$, then there always exists an *interpolant* I over the common symbols of ϕ and ψ s.t. $\phi \models I$ and $I \models \psi$. In the verification literature, this terminology has been abused following [26], where a *reverse interpolant* (coined by Kovács and Voronkov in [19]) I over the common symbols of ϕ and ψ is defined by

Definition 3 (Interpolant). *Given two formulas ϕ and ψ in a theory $\mathcal{T}$ s.t. $\phi \wedge \psi \models_{\mathcal{T}} \bot$, a formula I is an* interpolant *of ϕ and ψ if (i) $\phi \models_{\mathcal{T}} I$; (ii) $I \wedge \psi \models \bot$; and (iii) I only contains common symbols and free variables shared by ϕ and ψ.*

Definition 4. *A basic semi-algebraic set $\{\mathbf{x} \in \mathbb{R}^n \mid \bigwedge_{i=1}^{s} p_i(\mathbf{x}) \geq 0\}$ is called a set of the* Archimedean form *if $\mathcal{M}_{\mathbf{x}}\{p_1(\mathbf{x}), \ldots, p_s(\mathbf{x})\}$ is Archimedean, where $p_i(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$, $i = 1, \ldots, s$.*

The interpolant synthesis problem of interest in this paper is given in Problem 1.

Problem 1. Let $\phi(\mathbf{x}, \mathbf{y})$ and $\psi(\mathbf{x}, \mathbf{z})$ be two polynomial formulas of the form

$$\begin{aligned} \phi(\mathbf{x}, \mathbf{y}) &: f_1(\mathbf{x}, \mathbf{y}) \geq 0 \wedge \cdots \wedge f_m(\mathbf{x}, \mathbf{y}) \geq 0, \\ \psi(\mathbf{x}, \mathbf{z}) &: g_1(\mathbf{x}, \mathbf{z}) \geq 0 \wedge \cdots \wedge g_n(\mathbf{x}, \mathbf{z}) \geq 0, \end{aligned}$$

where, $\mathbf{x} \in \mathbb{R}^r$, $\mathbf{y} \in \mathbb{R}^s$, $\mathbf{z} \in \mathbb{R}^t$ are variable vectors, $r, s, t \in \mathbb{N}$, and $f_1, \ldots, f_m, g_1, \ldots, g_n$ are polynomials in the corresponding variables. Suppose $\phi \wedge \psi \models \bot$, and $\{(\mathbf{x}, \mathbf{y}) \mid \phi(\mathbf{x}, \mathbf{y})\}$ and $\{(\mathbf{x}, \mathbf{z}) \mid \psi(\mathbf{x}, \mathbf{z})\}$ are semi-algebraic sets of the Archimedean form. Find a polynomial $h(\mathbf{x})$ such that $h(\mathbf{x}) > 0$ is an interpolant for ϕ and ψ.

3 Existence of Interpolants

The basic idea and steps of proving the existence of interpolants are as follows: Because an interpolant of ϕ and ψ contains only the common symbols in ϕ and ψ, it is natural to consider the projections of the sets defined by ϕ and ψ on $\mathbf{x}$, i.e. $P_{\mathbf{x}}(\phi(\mathbf{x}, \mathbf{y})) \hat{=} \{\mathbf{x} \mid \exists \mathbf{y}.\, \phi(\mathbf{x}, \mathbf{y})\}$ and $P_{\mathbf{x}}(\psi(\mathbf{x}, \mathbf{z})) \hat{=} \{\mathbf{x} \mid \exists \mathbf{z}.\, \psi(\mathbf{x}, \mathbf{z})\}$, which are obviously disjoint. We therefore prove that, if $h(\mathbf{x}) = 0$ separates $P_{\mathbf{x}}(\phi(\mathbf{x}, \mathbf{y}))$ and $P_{\mathbf{x}}(\psi(\mathbf{x}, \mathbf{z}))$, then $h(\mathbf{x})$ solves Problem 1 (see Proposition 1). Thus, we only need to prove the existence of such $h(\mathbf{x})$ through the following steps: First, we prove that $P_{\mathbf{x}}(\phi(\mathbf{x}, \mathbf{y}))$ and $P_{\mathbf{x}}(\psi(\mathbf{x}, \mathbf{z}))$ are compact semi-algebraic sets which are unions of finitely many basic closed semi-algebraic sets (see Lemma 1). Second, using Putinar's Positivstellensatz, we prove that, for two disjoint basic closed semi-algebraic sets S_1 and S_2 of the Archimedean form, there exists a polynomial $h_1(\mathbf{x})$ such that $h_1(\mathbf{x}) = 0$ separates S_1 and S_2 (see Lemma 2). This result is then extended to the case that S_2 is a finite union of basic closed semi-algebraic sets (see Lemma 3). Finally, by generalizing Lemma 3 to the case that two compact semi-algebraic sets both are unions of finitely many basic closed semi-algebraic sets and together with Proposition 1, we prove the existence of interpolant in Theorem 2 and Corollary 1.

Proposition 1. *If $h(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ satisfies the following constraints*

$$\forall \mathbf{x} \in P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y})).h(\mathbf{x}) > 0 \quad \textit{and} \quad \forall \mathbf{x} \in P_{\mathbf{x}}(\psi(\mathbf{x},\mathbf{z})).h(\mathbf{x}) < 0, \tag{1}$$

then $h(\mathbf{x}) > 0$ is an interpolant for $\phi(\mathbf{x},\mathbf{y})$ and $\psi(\mathbf{x},\mathbf{z})$, where $\phi(\mathbf{x},\mathbf{y})$ and $\psi(\mathbf{x},\mathbf{z})$ are defined as in Problem 1.

Proof. According to Definition 3, it is enough to prove that $\phi(\mathbf{x},\mathbf{y}) \models h(\mathbf{x}) > 0$ and $\psi(\mathbf{x},\mathbf{z}) \models h(\mathbf{x}) \leq 0$.

Since any $(\mathbf{x}_0,\mathbf{y}_0)$ satisfying $\phi(\mathbf{x},\mathbf{y})$ must imply $\mathbf{x}_0 \in P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y}))$, it follows that $h(\mathbf{x}_0) > 0$ from (1) and $\phi(\mathbf{x},\mathbf{y}) \models h(\mathbf{x}) > 0$. Similarly, we can prove $\psi(\mathbf{x},\mathbf{z}) \models h(\mathbf{x}) < 0$, implying that $\psi(\mathbf{x},\mathbf{z}) \models h(\mathbf{x}) \leq 0$. Therefore, $h(\mathbf{x}) > 0$ is an interpolant for $\phi(\mathbf{x},\mathbf{y})$ and $\psi(\mathbf{x},\mathbf{z})$. □

In order to synthesize such $h(\mathbf{x})$ in Proposition 1, we first dig deeper into the two sets $P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y}))$ and $P_{\mathbf{x}}(\psi(\mathbf{x},\mathbf{z}))$. As shown later, i.e. in Lemma 1, we will find that these two sets are compact semi-algebraic sets of the form $\{\mathbf{x} \mid \bigvee_{i=1}^{c} \bigwedge_{j=1}^{J_i} \alpha_{i,j}(\mathbf{x}) \geq 0\}$. Before this lemma, we introduce Finiteness theorem pertinent to a *basic closed semi-algebraic subset* of $\mathbb{R}^n$, which will be used in the proof of Lemma 1, where a basic closed semi-algebraic subset of $\mathbb{R}^n$ is a set of the form $\{\mathbf{x} \in \mathbb{R}^n \mid \alpha_1(\mathbf{x}) \geq 0, \ldots, \alpha_k(\mathbf{x}) \geq 0\}$ with $\alpha_1, \ldots, \alpha_k \in \mathbb{R}[\mathbf{x}]$.

Theorem 1 (Finiteness Theorem, Theorem 2.7.2 in [3]). *Let $A \subset \mathbb{R}^n$ be a closed semi-algebraic set. Then A is a finite union of basic closed semi-algebraic sets.*

Lemma 1. *The set $P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y}))$ is compact semi-algebraic set of the following form*

$$P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y})) := \{\mathbf{x} \mid \bigvee_{i=1}^{c} \bigwedge_{j=1}^{J_i} \alpha_{i,j}(\mathbf{x}) \geq 0\},$$

where $\alpha_{i,j}(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$, $i = 1, \ldots, c$, $j = 1, \ldots, J_i$. The same claim applies to the set $P_{\mathbf{x}}(\psi(\mathbf{x},\mathbf{z}))$ as well.

Proof. For the sake of simplicity, we denote $\{(\mathbf{x},\mathbf{y}) \mid \phi(\mathbf{x},\mathbf{y})\}$ and $P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y}))$ by S and $\pi(S)$, respectively.

Because S is a compact set and π is a continuous map that maps compact set to compact set, $\pi(S)$, which is the image of a compact set under a continuous map, is compact. Moreover, as S is a semi-algebraic set and the projection of a semi-algebraic set is also a semi-algebraic set by Tarski-Seidenberg theorem [2], this implies that $\pi(S)$ is a semi-algebraic set. Thus, $\pi(S)$ is a compact semi-algebraic set.

Since $\pi(S)$ is a compact semi-algebraic set, and also a closed semi-algebraic set, we have that $\pi(\mathrm{S})$ is a finite union of basic closed semi-algebraic sets from Theorem 1. Hence, there exist a series of polynomials $\alpha_{1,1}(\mathbf{x}), \ldots, \alpha_{1,J_1}(\mathbf{x}), \ldots, \alpha_{c,1}(\mathbf{x}), \ldots, \alpha_{c,J_c}(\mathbf{x})$ such that $\pi(\mathrm{S}) = \bigcup_{i=1}^{c}\{\mathbf{x} \mid \bigwedge_{j=1}^{J_i} \alpha_{i,j}(\mathbf{x}) \geq 0\} = \{\mathbf{x} \mid \bigvee_{i=1}^{c} \bigwedge_{j=1}^{J_i} \alpha_{i,j}(\mathbf{x}) \geq 0\}$. This concludes this lemma. □

After knowing the structure of $P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y}))$ and $P_{\mathbf{x}}(\psi(\mathbf{x},\mathbf{z}))$ being a union of some basic semialgebraic sets as illustrated in Lemma 1, we next prove the existence of $h(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ satisfying (1), as formally stated in Theorem 2.

Theorem 2. *Suppose that $\phi(\mathbf{x},\mathbf{y})$ and $\psi(\mathbf{x},\mathbf{z})$ are defined as in Problem 1, then there exists a polynomial $h(\mathbf{x})$ satisfying (1).*

As pointed out by an anonymous reviewer that Theorem 2 can be obtained by some properties of the ring of Nash functions proved in [29]. In what follows, we give a simpler and more intuitive proof. To the end, it requires some preliminaries first. The main tool in our proof is Putinar's Positivstellensatz, as formulated in Theorem 3.

Theorem 3 (Putinar's Positivstellensatz [31]). *Let $p_1,\ldots,p_k \in \mathbb{R}[\mathbf{x}]$ and $S_1 = \{\mathbf{x} \mid p_1(\mathbf{x}) \geq 0,\ldots,p_k(\mathbf{x}) \geq 0\}$. Assume that the quadratic module $\mathcal{M}(p_1,\ldots,p_k)$ is Archimedean. For $q \in \mathbb{R}[\mathbf{x}]$, if $q > 0$ on S_1 then $q \in \mathcal{M}(p_1,\ldots,p_k)$.*

With Putinar's Positivstellensatz we can draw a conclusion that there exists a polynomial such that its zero level set[3] separates two compact semi-algebraic sets of the Archimedean form, as claimed in Lemmas 2 and 3.

Lemma 2. *Let $S_1 = \{\mathbf{x} \mid p_1(\mathbf{x}) \geq 0,\ldots,p_J(\mathbf{x}) \geq 0\}$, $S_2 = \{\mathbf{x} \mid q_1(\mathbf{x}) \geq 0,\ldots,q_K(\mathbf{x}) \geq 0\}$ be semi-algebraic sets of the Archimedean form and $S_1 \cap S_2 = \emptyset$, then there exists a polynomial $h_1(\mathbf{x})$ such that*

$$\forall \mathbf{x} \in S_1.\ h_1(\mathbf{x}) > 0, \quad \forall \mathbf{x} \in S_2.\ h_1(\mathbf{x}) < 0. \tag{2}$$

Proof. Since $S_1 \cap S_2 = \emptyset$, it follows

$$p_2 \geq 0 \wedge \cdots \wedge p_J \geq 0 \wedge q_1 \geq 0 \wedge \cdots \wedge q_K \geq 0 \models -p_1 > 0.$$

Let $S_3 = \{\mathbf{x} \mid p_2 \geq 0 \wedge \cdots \wedge p_J \geq 0 \wedge q_1 \geq 0 \wedge \cdots \wedge q_K \geq 0\}$, then $-p_1 > 0$ on S_3. Since S_1 and S_2 are semi-algebraic sets of the Archimedean form, it follows $\mathcal{M}_{\mathbf{x}}(p_2(\mathbf{x}),\ldots,p_J(\mathbf{x}),q_1(\mathbf{x}),\ldots,q_K(\mathbf{x}))$ is also Archimedean. Hence, S_3 is compact. From $-p_1 > 0$ on S_3, we further have that there exists some $u_1 \in \sum \mathbb{R}[\mathbf{x}]^2$ such that $-u_1p_1 - 1 > 0$ on S_3. Using Theorem 3, we have that

$$-u_1p_1 - 1 \in \mathcal{M}_{\mathbf{x}}(p_2(\mathbf{x}),\ldots,p_J(\mathbf{x}),q_1(\mathbf{x}),\ldots,q_K(\mathbf{x})),$$

implying that there exists a set of sums of squares polynomials $u_2,\ldots,u_J$ and $v_0,v_1,\ldots,v_K \in \mathbb{R}[\mathbf{x}]$, such that

$$-u_1p_1 - 1 \equiv u_2p_2 + \cdots + u_Jp_J + v_0 + v_1q_1 + \cdots + v_Kq_K.$$

Let $h_1 = \frac{1}{2} + u_1p_1 + \cdots + u_Jp_J$, i.e., $-h_1 = \frac{1}{2} + v_0 + v_1q_1 + \cdots + v_Kq_K$. It is easy to check that (2) holds. □

[3] The zero level set of an n-variate polynomial $h(\mathbf{x})$ is defined as $\{\mathbf{x} \in \mathbb{R}^n \mid h(\mathbf{x}) = 0\}$.

Lemma 3 generalizes the result of Lemma 2 to more general compact semi-algebraic sets of the Archimedean form, which is the union of multiple basic semi-algebraic sets.

Lemma 3. *Assume* $S_0 = \{\mathbf{x} \mid p_1(\mathbf{x}) \geq 0, \ldots, p_J(\mathbf{x}) \geq 0\}$ *and* $S_i = \{\mathbf{x} \mid q_{i,1}(\mathbf{x}) \geq 0, \ldots, q_{i,K_i}(\mathbf{x}) \geq 0\}$, $i = 1, \ldots, b$, *are semi-algebraic sets of the Archimedean form, and* $S_0 \cap \bigcup_{i=1}^{b} S_i = \emptyset$, *then there exists a polynomial* $h_0(\mathbf{x})$ *such that*

$$\forall \mathbf{x} \in S_0.\ h_0(\mathbf{x}) > 0, \quad \forall \mathbf{x} \in \bigcup_{i=1}^{b} S_i.\ h_0(\mathbf{x}) < 0. \tag{3}$$

In order to prove this lemma, we prove the following lemma first.

Lemma 4. *Let* $c, d \in \mathbb{R}$ *with* $0 < c < d$ *and* $U_0 = [c, d]^r$. *There exists a polynomial* $\hat{h}(\mathbf{x})$ *such that*

$$\mathbf{x} \in U_0 \models \hat{h}(\mathbf{x}) > 0 \models \bigwedge_{i=1}^{r} x_i > 0, \tag{4}$$

where $\mathbf{x} = (x_1, \ldots, x_r)$.

Proof. We show that there exists $k \in \mathbb{N}$ such that $\hat{h}(\mathbf{x}) = (\frac{d}{2})^{2k} - (x_1 - \frac{c+d}{2})^{2k} - \cdots - (x_r - \frac{c+d}{2})^{2k}$ satisfies (4). It is evident that $\hat{h}(\mathbf{x}) > 0 \models \bigwedge_{i=1}^{r} x_i > 0$ holds. In the following we just need to verify that $\bigwedge_{i=1}^{r} c \leq x_i \leq d \models \hat{h}(\mathbf{x}) > 0$ holds. Since $c \leq x_i \leq d$, we have $(x_i - \frac{c+d}{2})^{2k} \leq (\frac{d-c}{2})^{2k}$ and $(\frac{d}{2})^{2k} - \sum_{i=1}^{r}(x_i - \frac{c+d}{2})^{2k} \geq (\frac{d}{2})^{2k} - r(\frac{d-c}{2})^{2k}$. Obviously, if an integer k satisfies $(\frac{d}{d-c})^{2k} > r$, then $(\frac{d}{2})^{2k} - \sum_{i=1}^{r}(x_i - \frac{c+d}{2})^{2k} > 0$. The existence of such k satisfying $(\frac{d}{d-c})^{2k} > r$ is assured by $\frac{d}{d-c} > 1$. □

Now we give a proof for Lemma 3 as follows.

Proof (of Lemma 3). For any i with $1 \leq i \leq b$, according to Lemma 2, there exists a polynomial $h_i \in \mathbb{R}[\mathbf{x}]$, satisfying $\forall \mathbf{x} \in S_0.\ h_i(\mathbf{x}) > 0$ and $\forall \mathbf{x} \in S_i.\ h_i(\mathbf{x}) < 0$.

Next, we construct $h_0(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ from $h_1(\mathbf{x}), \ldots, h_b(\mathbf{x})$. Since S_0 is a semi-algebraic set of the Archimedean form, S_0 is compact and thus $h_i(\mathbf{x})$ has minimum value and maximum value on S_0, denoted by c_i and d_i respectively. Let $c = \min(c_1, \ldots, c_b)$ and $d = \max(d_1, \ldots, d_b)$. Clearly, $0 < c < d$.

From Lemma 4 there must exist a polynomial $\hat{h}(w_1, \ldots, w_b)$ such that

$$\bigwedge_{i=1}^{b} c \leq w_i \leq d \models \hat{h}(w_1, \ldots, w_b) > 0, \tag{5}$$

$$\hat{h}(w_1, \ldots, w_b) > 0 \models \bigwedge_{i=1}^{b} w_i > 0. \tag{6}$$

Let $h_0'(\mathbf{x}) = \hat{h}(h_1(\mathbf{x}), \ldots, h_b(\mathbf{x}))$. Obviously, $h_0'(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$. We next prove that $h_0'(\mathbf{x})$ satisfies (3) in Lemma 3.

For all $\mathbf{x}_0 \in S_0$, $c \leq h_i(\mathbf{x}_0) \leq d$, $i = 1, \ldots, b$, $h_0'(\mathbf{x}_0) = \hat{h}(h_1(\mathbf{x}_0), \ldots, h_b(\mathbf{x}_0)) > 0$ by (5). Therefore, the first constraint in (3), i.e. $\forall \mathbf{x}_0 \in S_0.h_0(\mathbf{x}_0) > 0$, holds.

For any $\mathbf{x}_0 \in \bigcup_{i=1}^{b} S_i$, there must exist some i such that $\mathbf{x}_0 \in S_i$, implying that $h_i(\mathbf{x}_0) < 0$. By (6) we have $h_0'(\mathbf{x}_0) = \hat{h}(h_1(\mathbf{x}_0), \ldots, h_b(\mathbf{x}_0)) \leq 0$.

Thus, we obtain the conclusion that there exists a polynomial $h_0'(\mathbf{x})$ such that $\forall \mathbf{x} \in S_0.\ h_0'(\mathbf{x}) > 0$, and $\forall \mathbf{x} \in \bigcup_{i=1}^{b} S_i.\ h_0'(\mathbf{x}) \leq 0$. Also, since S_0 is a compact set, and $h_0'(\mathbf{x}) > 0$ on S_0, there must exist some positive number $\epsilon > 0$ such that $h_0'(\mathbf{x}) - \epsilon > 0$ over S_0. Then $h_0'(\mathbf{x}) - \epsilon < 0$ on $\bigcup_{i=1}^{b} S_i$. Therefore, setting $h_0(\mathbf{x}) := h_0'(\mathbf{x}) - \epsilon$, Lemma 3 is proved. □

In Lemma 3 we proved that there exists a polynomial $h(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ such that its zero level set is a barrier between two semi-algebraic sets of the Archimedean form, of which one set is a union of finitely many basic semi-algebraic sets. In the following we will give a formal proof of Theorem 2, which is a generalization of Lemma 3.

Proof (of Theorem 2). According to Lemma 1 we have that $P_{\mathbf{x}}(\phi(\mathbf{x}, \mathbf{y}))$ and $P_{\mathbf{x}}(\psi(\mathbf{x}, \mathbf{z}))$ are compact sets, and there respectively exists a set of polynomials $p_{i,j}(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$, $i = 1, \ldots, a$, $j = 1, \ldots, J_i$, and $q_{l,k}(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$, $l = 1, \ldots, b$, $k = 1, \ldots, K_i$, such that

$$P_{\mathbf{x}}(\phi(\mathbf{x}, \mathbf{y})) = \{\mathbf{x} \mid \bigvee_{i=1}^{a} \bigwedge_{j=1}^{J_i} p_{i,j}(\mathbf{x}) \geq 0\}, \quad P_{\mathbf{x}}(\psi(\mathbf{x}, \mathbf{z})) = \{\mathbf{x} \mid \bigvee_{l=1}^{b} \bigwedge_{k=1}^{K_l} q_{l,k}(\mathbf{x}) \geq 0\}.$$

Since $P_{\mathbf{x}}(\phi(\mathbf{x}, \mathbf{y}))$ and $P_{\mathbf{x}}(\psi(\mathbf{x}, \mathbf{z}))$ are compact sets, there exists a positive $N \in \mathbb{R}$ such that $f = N - \sum_{i=1}^{r} x_i^2 \geq 0$ over $P_{\mathbf{x}}(\phi(\mathbf{x}, \mathbf{y}))$ and $P_{\mathbf{x}}(\psi(\mathbf{x}, \mathbf{z}))$. For each $i = 1, \ldots, a$ and each $l = 1, \ldots, b$, set $p_{i,0} = q_{l,0} = f$. Denote $\{\mathbf{x} \mid \bigvee_{i=1}^{a} \bigwedge_{j=0}^{J_i} p_{i,j}(\mathbf{x}) \geq 0\} = \bigcup_{i=1}^{a} \{\mathbf{x} \mid \bigwedge_{j=0}^{J_i} p_{i,j}(\mathbf{x}) \geq 0\}$ by P_1 and $\{\mathbf{x} \mid \bigvee_{l=1}^{b} \bigwedge_{k=0}^{K_l} q_{l,k}(\mathbf{x}) \geq 0\} = \bigcup_{l=1}^{b} \{\mathbf{x} \mid \bigwedge_{k=0}^{K_l} q_{l,k}(\mathbf{x}) \geq 0\}$ by P_2. It is easy to see that $P_1 = P_{\mathbf{x}}(\phi(\mathbf{x}, \mathbf{y}))$, $P_2 = P_{\mathbf{x}}(\psi(\mathbf{x}, \mathbf{z}))$.

Since $\phi \wedge \psi \models \bot$, there does not exist $(\mathbf{x}, \mathbf{y}, \mathbf{z}) \in \mathbb{R}^{r+s+t}$ that satisfies $\phi \wedge \psi$, implying that $P_{\mathbf{x}}(\phi(\mathbf{x}, \mathbf{y})) \cap P_{\mathbf{x}}(\psi(\mathbf{x}, \mathbf{z})) = \emptyset$ and thus $P_1 \cap P_2 = \emptyset$. Also, since $\{\mathbf{x} \mid \bigwedge_{j=0}^{J_{i_1}} p_{i_1,j}(\mathbf{x}) \geq 0\} \subseteq P_1$, for each $i_1 = 1, \ldots, a$, $\{\mathbf{x} \mid \bigwedge_{j=0}^{J_{i_1}} p_{i_1,j}(\mathbf{x}) \geq 0\} \cap P_2 = \emptyset$ holds. By Lemma 3 there exists $h_{i_1}(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ such that

$$\forall \mathbf{x} \in \{\mathbf{x} \mid \bigwedge_{j=0}^{J_{i_1}} p_{i_1,j}(\mathbf{x}) \geq 0\}.h_{i_1}(\mathbf{x}) > 0, \quad \forall \mathbf{x} \in P_2.h_{i_1}(\mathbf{x}) < 0.$$

Let $S' = \{\mathbf{x} \mid -h_1(\mathbf{x}) \geq 0, \ldots, -h_a(\mathbf{x}) \geq 0, N - \sum_{i=1}^{r} x_i^2 \geq 0\}$. Obviously, S' is a semialgebraic set of the Archimedean form, $P_2 \subset S'$ and $P_1 \cap S' = \emptyset$. Therefore, according to Lemma 2, there exists a polynomial $\overline{h}(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ such that $\forall \mathbf{x} \in S'.\ \overline{h}(\mathbf{x}) > 0$ and $\forall \mathbf{x} \in P_1.\ \overline{h}(\mathbf{x}) < 0$. Let $h(\mathbf{x}) = -\overline{h}(\mathbf{x})$, then we have $\forall \mathbf{x} \in P_1.\ h(\mathbf{x}) > 0$ and $\forall \mathbf{x} \in P_2.\ h(\mathbf{x}) < 0$, implying that $\forall \mathbf{x} \in P_{\mathbf{x}}(\phi(\mathbf{x}, \mathbf{y})).h(\mathbf{x}) > 0$ and $\forall \mathbf{x} \in P_{\mathbf{x}}(\psi(\mathbf{x}, \mathbf{z})).h(\mathbf{x}) < 0$. Thus, this completes the proof of Theorem 2. □

Consequently, we immediately have the following conclusion.

Corollary 1. *Let $\phi(\mathbf{x},\mathbf{y})$ and $\psi(\mathbf{x},\mathbf{z})$ be defined as in Problem 1. There must exist a polynomial $h(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ such that $h(\mathbf{x}) > 0$ is an interpolant for ϕ and ψ.*

Actually, since $P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y}))$ and $P_{\mathbf{x}}(\psi(\mathbf{x},\mathbf{z}))$ both are compact set by Lemma 1, and $h(\mathbf{x}) > 0$ on $P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y}))$ and $h(\mathbf{x}) < 0$ on $P_{\mathbf{x}}(\psi(\mathbf{x},\mathbf{z}))$, we can obtain $h'(\mathbf{x})$ by giving a small perturbation to the coefficients of $h(\mathbf{x})$ such that $h'(\mathbf{x})$ has the property of $h(\mathbf{x})$. Hence, there should exist a $h(\mathbf{x}) \in \mathbb{Q}[\mathbf{x}]$ such that $h(\mathbf{x}) > 0$ is an interpolant for ϕ and ψ, intuitively.

Theorem 4. *Let $\phi(\mathbf{x},\mathbf{y})$ and $\psi(\mathbf{x},\mathbf{z})$ be defined as in Problem 1. There must exist a polynomial $h(\mathbf{x}) \in \mathbb{Q}[\mathbf{x}]$ such that $h(\mathbf{x}) > 0$ is an interpolant for ϕ and ψ.*

Proof. We just need to prove there exists a polynomial $h(\mathbf{x}) \in \mathbb{Q}[\mathbf{x}]$ satisfying (1).

By Theorem 2, there exists a polynomial $h'(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ satisfying (1). Since $P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y}))$ and $P_{\mathbf{x}}(\psi(\mathbf{x},\mathbf{z}))$ are compact sets, $h'(\mathbf{x}) > 0$ on $P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y}))$ and $h'(\mathbf{x}) < 0$ on $P_{\mathbf{x}}(\psi(\mathbf{x},\mathbf{z}))$, there exist $\eta_1 > 0$ and $\eta_2 > 0$ such that

$$\forall \mathbf{x} \in P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y})).h'(\mathbf{x}) - \eta_1 \geq 0,\ \forall \mathbf{x} \in P_{\mathbf{x}}(\psi(\mathbf{x},\mathbf{z})).h'(\mathbf{x}) + \eta_2 \leq 0.$$

Let $\eta = \min(\frac{\eta_1}{2}, \frac{\eta_2}{2})$. Suppose $h'(\mathbf{x}) \in \mathbb{R}[\mathbf{x}]$ has the form $h'(\mathbf{x}) = \sum_{\alpha\in\Omega} c_\alpha \mathbf{x}^\alpha$, where $\alpha \in \mathbb{N}^r$, $\Omega \subset \mathbb{N}^r$ is a finite set of indices, r is the dimension of $\mathbf{x}$, $\mathbf{x}^\alpha$ is the monomial $\mathbf{x}_1^{\alpha_1} \cdots \mathbf{x}_r^{\alpha_r}$, and $0 \neq c_\alpha \in \mathbb{R}$ is the coefficient of monomial $\mathbf{x}^\alpha$. Let $N = |\Omega|$ be the cardinality of Ω. Since $P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y}))$ and $P_{\mathbf{x}}(\psi(\mathbf{x},\mathbf{z}))$ are compact sets, for any $\alpha \in \Omega$, there exists $M_\alpha > 0$ such that $M_\alpha = \max\{|\mathbf{x}^\alpha| \mid \mathbf{x} \in P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y})) \cup P_{\mathbf{x}}(\psi(\mathbf{x},\mathbf{z}))\}$. Then for any fixed polynomial $\hat{h}(\mathbf{x}) = \sum_{\alpha\in\Omega} d_\alpha \mathbf{x}^\alpha$, with $d_\alpha \in [c_\alpha - \frac{\eta}{NM_\alpha}, c_\alpha + \frac{\eta}{NM_\alpha}]$, and any $\mathbf{x} \in P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y})) \cup P_{\mathbf{x}}(\psi(\mathbf{x},\mathbf{z}))$, we have

$$|\hat{h}(\mathbf{x}) - h'(\mathbf{x})| = |\sum_{\alpha\in\Omega}(d_\alpha - c_\alpha)\mathbf{x}^\alpha| \leq \sum_{\alpha\in\Omega}|(d_\alpha - c_\alpha)| \cdot |\mathbf{x}^\alpha| \leq \sum_{\alpha\in\Omega}\frac{\eta}{NM_\alpha} \cdot M_\alpha = \eta.$$

Since $\eta = \min(\frac{\eta_1}{2}, \frac{\eta_2}{2})$, hence

$$\forall \mathbf{x} \in P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y})).\hat{h}(\mathbf{x}) \geq \frac{\eta_1}{2} > 0, \quad \forall \mathbf{x} \in P_{\mathbf{x}}(\psi(\mathbf{x},\mathbf{z})).\hat{h}(\mathbf{x}) \leq -\frac{\eta_2}{2} < 0. \tag{7}$$

Since for any $d_\alpha \in [c_\alpha - \frac{\eta}{NM_\alpha}, c_\alpha + \frac{\eta}{NM_\alpha}]$ (7) holds, there must exist some rational number $r_\alpha \in \mathbb{Q}$ in $[c_\alpha - \frac{\eta}{NM_\alpha}, c_\alpha + \frac{\eta}{NM_\alpha}]$ satisfying (7) because of the density of rational numbers. Thus, let $h(\mathbf{x}) = \sum_{\alpha\in\Omega} r_\alpha \mathbf{x}^\alpha$. Clearly, it follows that $h(\mathbf{x}) \in \mathbb{Q}[\mathbf{x}]$ and (1) holds. □

So, the existence of $h(\mathbf{x}) \in \mathbb{Q}[\mathbf{x}]$ is guaranteed. Moreover, from the proof of Theorem 4, we know that a small perturbation of $h(\mathbf{x})$ is permitted, which is a good property for computing $h(\mathbf{x})$ in a numeric way. In the subsequent subsection, we recast the problem of finding such $h(\mathbf{x})$ as a semi-definite programming problem.

4 SOS Formulation

Similar to [7], in this section, we discuss how to reduce the problem of finding $h(\mathbf{x})$ satisfying (1) to a sum of squares programming problem.

Theorem 5. *Let $\phi(\mathbf{x},\mathbf{y})$ and $\psi(\mathbf{x},\mathbf{z})$ be defined as in the Problem 1. Then there exist $m+n+2$ SOS (sum of squares) polynomials $u_i(\mathbf{x},\mathbf{y})$ $(i=1,\ldots,m+1)$, $v_j(\mathbf{x},\mathbf{z})$ $(j=1,\ldots,n+1)$ and a polynomial $h(\mathbf{x})$ such that*

$$h-1=\sum_{i=1}^{m}u_i f_i+u_{m+1},\quad -h-1=\sum_{j=1}^{n}v_j g_j+v_{n+1}, \tag{8}$$

and $h(\mathbf{x})>0$ is an interpolant for $\phi(\mathbf{x},\mathbf{y})$ and $\psi(\mathbf{x},\mathbf{z})$.

Proof. By Theorem 2 there exists a polynomial $\hat{h}(\mathbf{x})$ such that

$$\forall \mathbf{x}\in P_{\mathbf{x}}(\phi(\mathbf{x},\mathbf{y})).\hat{h}(\mathbf{x})>0,\quad \forall \mathbf{x}\in P_{\mathbf{x}}(\psi(\mathbf{x},\mathbf{z})).\hat{h}(\mathbf{x})<0.$$

Set $S_1=\{(\mathbf{x},\mathbf{y})\mid f_1\geq 0,\ldots,f_m\geq 0\}$ and $S_2=\{(\mathbf{x},\mathbf{z})\mid g_1\geq 0,\ldots,g_n\geq 0\}$. Since $\hat{h}(\mathbf{x})>0$ on S_1, which is compact, there exist $\epsilon_1>0$ such that $\hat{h}(\mathbf{x})-\epsilon_1>0$ on S_1. Similarly, there exist $\epsilon_2>0$ such that $-\hat{h}(\mathbf{x})-\epsilon_2>0$ on S_2. Let $\epsilon=\min(\epsilon_1,\epsilon_2)$, and $h(\mathbf{x})=\frac{\hat{h}(\mathbf{x})}{\epsilon}$, then $h(\mathbf{x})-1>0$ on S_1 and $-h(\mathbf{x})-1>0$ on S_2. Since $\mathcal{M}_{\mathbf{x},\mathbf{y}}(f_1(\mathbf{x},\mathbf{y}),\ldots,f_m(\mathbf{x},\mathbf{y}))$ is Archimedean, from Theorem 3, we have $h(\mathbf{x})-1\in\mathcal{M}_{\mathbf{x},\mathbf{y}}(f_1(\mathbf{x},\mathbf{y}),\ldots,f_m(\mathbf{x},\mathbf{y}))$. Similarly, $-h(\mathbf{x})-1\in\mathcal{M}_{\mathbf{x},\mathbf{z}}(g_1(\mathbf{x},\mathbf{z}),\ldots,g_n(\mathbf{x},\mathbf{z}))$. That is, there exist $m+n+2$ SOS polynomials u_i,v_j satisfying the following semi-definite constraints:

$$h(\mathbf{x})-1=\sum_{i=1}^{m}u_i f_i+u_{m+1},\quad -h(\mathbf{x})-1=\sum_{j=1}^{n}v_j g_j+v_{n+1}. \qquad \square$$

According to Theorem 5, the problem of finding $h(\mathbf{x})\in\mathbb{R}[\mathbf{x}]$ solving Problem 1 can be equivalently reformulated as the problem of searching for SOS polynomials $u_1(\mathbf{x},\mathbf{y}),\ldots,u_m(\mathbf{x},\mathbf{y})$, $v_1(\mathbf{x},\mathbf{z}),\ldots,v_n(\mathbf{x},\mathbf{z})$ and a polynomial $h(\mathbf{x})$ with appropriate degrees such that

$$\begin{cases} h(\mathbf{x})-1-\sum_{i=1}^{m}u_i f_i\in\sum\mathbb{R}[\mathbf{x},\mathbf{y}]^2,\\ -h(\mathbf{x})-1-\sum_{j=1}^{n}v_j g_j\in\sum\mathbb{R}[\mathbf{x},\mathbf{z}]^2,\\ u_i\in\sum\mathbb{R}[\mathbf{x},\mathbf{y}]^2, i=1,\ldots,m,\\ v_j\in\sum\mathbb{R}[\mathbf{x},\mathbf{z}]^2, j=1,\ldots,n. \end{cases} \tag{9}$$

(9) is SOS constraints over SOS multipliers $u_1(\mathbf{x},\mathbf{y}),\ldots,u_m(\mathbf{x},\mathbf{y})$, $v_1(\mathbf{x},\mathbf{z})$, $\ldots,v_n(\mathbf{x},\mathbf{z})$, polynomial $h(\mathbf{x})$, which is convex and could be solved by many existing semi-definite programming solvers such as the optimization library AiSat [7] built on CSDP [4]. Therefore, according to Theorem 5, $h(\mathbf{x})>0$ is an interpolant for ϕ and ψ, which is formulated in Theorem 6.

Theorem 6 (Soundness). *Suppose that* $\phi(\mathbf{x},\mathbf{y})$ *and* $\psi(\mathbf{x},\mathbf{z})$ *are defined as in Problem 1, and* $h(\mathbf{x})$ *is a feasible solution to (9), then* $h(\mathbf{x})$ *solves Problem 1, i.e.* $h(\mathbf{x})>0$ *is an interpolant for* ϕ *and* ψ.

Moreover, we have the following completeness theorem stating that if the degrees of $h(\mathbf{x})\in\mathbb{R}[\mathbf{x}]$ and $u_i(\mathbf{x},\mathbf{y})\in\sum\mathbb{R}[\mathbf{x},\mathbf{y}]^2, v_j(\mathbf{x},\mathbf{z})\in\sum\mathbb{R}[\mathbf{x},\mathbf{z}]^2$, $i=1,\ldots,m$, $j=1,\ldots,n$, are large enough, $h(\mathbf{x})$ can be synthesized definitely via solving (9).

Theorem 7 (Completeness). *For Problem 1, there must be polynomials* $u_i(\mathbf{x},\mathbf{y})\in\mathbb{R}_N[\mathbf{x},\mathbf{y}]$ $(i=1,\ldots,m)$, $v_j(\mathbf{x},\mathbf{z})\in\mathbb{R}_N[\mathbf{x},\mathbf{z}]$ $(j=1,\ldots,n)$ *and* $h(\mathbf{x})\in\mathbb{R}_N[\mathbf{x}]$ *satisfying (11) for some positive integer* N, *where* $\mathbb{R}_k[\cdot]$ *stands for the family of polynomials of degree no more than* k.

Proof. This is an immediate result of Theorem 5. □

Example 2. Consider two contradictory formulas ϕ and ψ defined by

$$f_1(x,y,z,a_1,b_1,c_1,d_1)\geq 0\wedge f_2(x,y,z,a_1,b_1,c_1,d_1)\geq 0\wedge f_3(x,y,z,a_1,b_1,c_1,d_1)\geq 0,$$
$$g_1(x,y,z,a_2,b_2,c_2,d_2)\geq 0\wedge g_2(x,y,z,a_2,b_2,c_2,d_2)\geq 0\wedge g_3(x,y,z,a_2,b_2,c_2,d_2)\geq 0,$$

respectively, where

$$f_1=4-x^2-y^2-z^2-a_1^2-b_1^2-c_1^2-d_1^2,\quad f_2=-y^4+2x^4-a_1^4-1/100,$$
$$f_3=z^2-b_1^2-c_1^2-d_1^2-x-1,\quad g_1=4-x^2-y^2-z^2-a_2^2-b_2^2-c_2^2-d_2^2,$$
$$g_2=x^2-y-a_2-b_2-d_2^2-3,\quad g_3=x.$$

It is easy to observe that ϕ and ψ satisfy the conditions in Problem 1. Since there are local variables in ϕ and ψ and the degree of f_2 is 4, the interpolant generation methods in [7] and [10] are not applicable. We get a concrete SDP problem of the form (9) by setting the degree of the polynomial $h(x,y,z)$ in (9) to be 2. Using the MATLAB package YALMIP [23] and Mosek [28], we obtain

$$h(x,y,z)=-416.7204-914.7840x+472.6184y+199.8985x^2+190.2252y^2$$
$$+690.4208z^2-187.1592xy.$$

Pictorially, we plot $P_{x,y,z}(\phi(x,y,z,a_1,b_1,c_1,d_1))$, $P_{x,y,z}(\psi(x,y,z,a_2,b_2,c_2,d_2))$ and $\{(x,y,z)\mid h(x,y,z)>0\}$ in Fig. 2. It is evident that $h(x,y,z)$ as presented above for $d_h=2$ is a real interpolant for $\phi(x,y,z,a,b,c,d)$ and $\psi(x,y,z,a,b,c,d)$.

5 Avoidance of the Unsoundness Due to Numerical Error in SDP

In this section, we discuss how to avoid the unsoundness of our approach caused by numerical error in SDP based on the work in [32].

A square matrix A is *positive semidefinite* if A is real symmetric and all its eigenvalues are nonnegative, denote by $A \succeq 0$.

In order to solve formula (9) to obtain $h(\mathbf{x})$, we first need to fix a degree bound of u_i, v_j and h, say $2d$, $d \in \mathbb{N}$. It is well-known that any $u(\mathbf{x}) \in \sum \mathbb{R}[\mathbf{x}]^2$ with degree $2d$ can be represented by

$$u(\mathbf{x}) \equiv E_d(\mathbf{x})^T C_u E_d(\mathbf{x}), \tag{10}$$

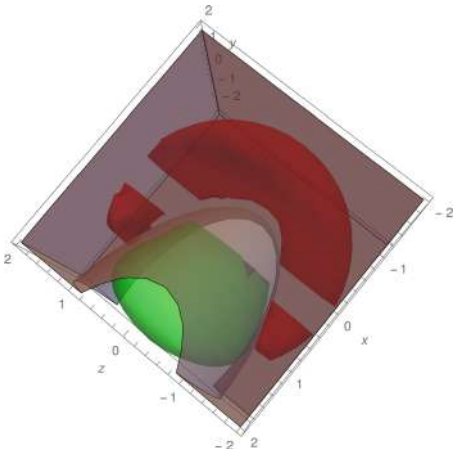


Fig. 2. Example 2. (Red region: $P_{x,y,z}(\phi(x,y,z,a_1,b_1,c_1,d_1))$; green region: $P_{x,y,z}(\psi(x,y,z,a_2,b_2,c_2,d_2))$; gray region: $\{(x,y,z) \mid h(x,y,z) > 0\}$.) (Color figure online)

where $C_u \in \mathbb{R}^{\binom{r+d}{d} \times \binom{r+d}{d}}$ with $C_u \succeq 0$, $E_d(\mathbf{x})$ is a column vector with all monomials in $\mathbf{x}$, whose total degree is not greater than d, and $E_d(\mathbf{x})^T$ stands for the transposition of $E_d(\mathbf{x})$. Equaling the corresponding coefficient of each monomial whose degree is less than or equal to $2d$ at the two sides of (10), we can get a linear equation system as

$$\mathtt{tr}(A_{u,k} C_u) = b_{u,k},\ k = 1, \ldots, K_u, \tag{11}$$

where $A_{u,k} \in \mathbb{R}^{\binom{r+d}{d} \times \binom{r+d}{d}}$ is constant matrix, $b_{u,k} \in \mathbb{R}$ is constant, $\mathtt{tr}(A)$ stands for the trace of matrix A. Thus, searching for u_i, v_j and h satisfying (9) can be reduced to the following SDP problem:

$$\begin{aligned}
\mathtt{find}:\ & C_{u_1}, \ldots, C_{u_m}, C_{v_1}, \ldots, C_{v_n}, C_h, \\
\mathtt{s.t.}\ & \mathtt{tr}(A_{u_i,k} C_{u_i}) = b_{u_i,k},\ i = 1, \ldots, m, k = 1, \ldots, K_{u_i}, \\
& \mathtt{tr}(A_{v_j,k} C_{v_j}) = b_{v_j,k},\ j = 1, \ldots, n, k = 1, \ldots, K_{v_j}, \\
& \mathtt{tr}(A_{h,k} C_h) = b_{h,k},\ k = 1, \ldots, K_h, \\
& \mathtt{diag}(C_{u_1}, \ldots, C_{u_m}, C_{v_1}, \ldots, C_{v_n}, C_{h-1-uf}, C_{-h-1-vg}) \succeq 0,
\end{aligned} \tag{12}$$

where C_{h-1-uf} is the matrix corresponding to polynomial $h - 1 - \sum_{i=1}^{m} u_i f_i$, which is a linear combination of $C_{u_1}, \ldots, C_{u_m}$ and C_h; similarly, $C_{-h-1-vg}$ is the matrix corresponding to polynomial $-h - 1 - \sum_{j=1}^{n} v_j g_j$, which is a linear combination of $C_{v_1}, \ldots, C_{v_n}$ and C_h; and $\mathtt{diag}(C_1, \ldots, C_k)$ is a block-diagonal matrix of $C_1, \ldots, C_k$.

Let D be the dimension of $C = \mathtt{diag}(C_{u_1}, \ldots, C_{-h-1-vg})$, i.e., $\mathtt{diag}(C_{u_1}, \ldots, C_{-h-1-vg}) \in \mathbb{R}^{D\times D}$ and $\widehat{C}$ be the approximate solution to (12) returned by calling a numerical SDP solver, the following theorem is proved in [32].

Theorem 8 ([32], Theorem 3). *$C \succeq 0$ if there exists $\widetilde{C} \in \mathbb{F}^{D\times D}$ such that the following conditions hold: 1. $\widetilde{C}_{ij} = C_{ij}$, for any $i \neq j$; 2. $\widetilde{C}_{ii} \leq C_{ii} - \alpha$, for any i; and 3. the Cholesky algorithm implemented in floating-point arithmetic can conclude that $\widetilde{C}$ is positive semi-definite, where $\mathbb{F}$ is a floating-point format, $\alpha = \frac{(D+1)\kappa}{1-(2D+2)\kappa}\mathtt{tr}(C) + 4(D+1)(2(D+2) + \max_i\{C_{ii}\})\eta$, in which κ is the unit roundoff of $\mathbb{F}$ and η is the underflow unit of $\mathbb{F}$.*

Corollary 2. *Let $\widetilde{C} \in \mathbb{F}^{D\times D}$. Suppose that $\frac{(D+1)D\kappa}{1-(2D+2)\kappa} + 4(D+1)\eta \leq \frac{1}{2}$, $\beta = \frac{(D+1)\kappa}{1-(2D+2)\kappa}\mathtt{tr}(\widetilde{C}) + 4(D+1)(2(D+2) + \max_i\{\widetilde{C}_{ii}\})\eta > 0$, where $\mathbb{F}$ is a floating-point format. Then $\widetilde{C} + 2\beta I \succeq 0$ if the Cholesky algorithm based on floating-point arithmetic succeeds on $\widetilde{C}$, i.e., concludes that $\widetilde{C}$ is positive semi-definite.*

According to Remark 5 in [32], for IEEE 754 binary64 format with rounding to nearest, $\kappa = 2^{-53}(\simeq 10^{-16})$ and $\eta = 2^{-1075}(\simeq 10^{-323})$. In this case, the order of magnitude of β is 10^{-10} and $\frac{(D+1)D\kappa}{1-(2D+2)\kappa} + 4(D+1)\eta$ is 10^{-13}, much less than $\frac{1}{2}$. Obviously, β becomes smaller when the length of binary format becomes longer. W.l.o.g., we suppose that the Cholesky algorithm succeed in computing $\widehat{C}$ the solution of (12), which is reasonable as if an SDP solver returns a solution $\widehat{C}$, then $\widehat{C}$ should be considered to be positive semi-definite in the sense of numeric computation.

So, by Corollary 2, we have $\widehat{C} + 2\beta I \succeq 0$ holds, where I is the identity matrix with the corresponding dimension. Then we have

$$\mathtt{diag}(\widehat{C}_{u_1}, \ldots, \widehat{C}_{u_m}, \widehat{C}_{v_1}, \ldots, \widehat{C}_{v_n}, \widehat{C}_{h-1-uf}, \widehat{C}_{-h-1-vg}) + 2\beta I \succeq 0.$$

Let $\epsilon = \max_{p\in P, 1\leq i\leq K_p} |\mathtt{tr}(A_{p,i}\widehat{C}_p) - b_{p,i}|$, where $P = \{u_1, \ldots, u_m, v_1, \ldots, v_n, h\}$, which can be regarded as the tolerance of the SDP solver. Since $|\mathtt{tr}(A_{p,i}C_p) - b_{p,i}|$ is the error term for each monomial of p, i.e., ϵ can be considered as the error bound on the coefficients of polynomials u_i, v_j and h, for any polynomial $\hat{u}_i$ ($\hat{v}_j$ and $\hat{h}$), computed from (11) by replacing C_u with the corresponding $\widehat{C_u}$, there exists a corresponding remainder term R_{u_i} (resp. R_{v_j} and R_h) with degree not greater than $2d$, whose coefficients are bounded by ϵ. Hence, we have

$$\begin{aligned}
&\widehat{u}_i + R_{u_i} + 2\beta E_d(\mathbf{x},\mathbf{y})^T E_d(\mathbf{x},\mathbf{y}) \in \sum \mathbb{R}[\mathbf{x},\mathbf{y}]^2, i = 1, \ldots, m,\\
&\widehat{v}_j + R_{v_j} + 2\beta E_d(\mathbf{x},\mathbf{z})^T E_d(\mathbf{x},\mathbf{z}) \in \sum \mathbb{R}[\mathbf{x},\mathbf{z}]^2, j = 1, \ldots, n,\\
&\widehat{h} + R_h - 1 - \sum_{i=1}^{m}(\widehat{u}_i + R'_{u_i})f_i + 2\beta E_d(\mathbf{x},\mathbf{y})^T E_d(\mathbf{x},\mathbf{y}) \in \sum \mathbb{R}[\mathbf{x},\mathbf{y}]^2, \quad (13)\\
&-\widehat{h} + R'_h - 1 - \sum_{j=1}^{m}(\widehat{v}_j + R'_{v_j})g_j + 2\beta E_d(\mathbf{x},\mathbf{z})^T E_d(\mathbf{x},\mathbf{z}) \in \sum \mathbb{R}[\mathbf{x},\mathbf{z}]^2.
\end{aligned}$$

Now, in order to avoid unsoundness of our approach caused by the numerical issue due to SDP, we have to prove

$$f_1 \geq 0 \wedge \cdots \wedge f_m \geq 0 \Rightarrow \widehat{h} > 0, \tag{14}$$

$$g_1 \geq 0 \wedge \cdots \wedge g_n \geq 0 \Rightarrow \widehat{h} < 0. \tag{15}$$

Regarding (14), let $R_{2d,\mathbf{x}}$ be a polynomial in $\mathbb{R}[|\mathbf{x}|]$, whose total degree is $2d$, and all coefficients are 1, e.g., $R_{2,x,y} = 1 + |x| + |y| + |x^2| + |xy| + |y^2|$. Since $S = \{(\mathbf{x}, \mathbf{y}) \mid f_1 \geq 0 \wedge \cdots \wedge f_m \geq 0\}$ is a compact set, then for any polynomial $p \in \mathbb{R}[\mathbf{x}, \mathbf{y}]$, $|p|$ is bounded on S. Let M_1 be an upper bound of $R_{2d,\mathbf{x},\mathbf{y}}$ on S, M_2 an upper bound of $E_d(\mathbf{x}, \mathbf{y})^T E_d(\mathbf{x}, \mathbf{y})$, and M_{f_i} an upper bound of f_i on S. Then, $|R_{u_i}|$, $|R'_{u_i}|$ and $|R_h|$ are bounded by ϵM_1. Let $E_{\mathbf{xy}} = E_d(\mathbf{x}, \mathbf{y})^T E_d(\mathbf{x}, \mathbf{y})$. So for any $(\mathbf{x}_0, \mathbf{y}_0) \in S$, considering the polynomials below at $(\mathbf{x}_0, \mathbf{y}_0) \in S$, by the first and third line in (13),

$$\begin{aligned}
\widehat{h} \geq\ & 1 - R_h + \sum_{i=1}^{m} (\widehat{u_i} + R'_{u_i}) f_i - 2\beta E_{\mathbf{xy}} \\
\geq\ & 1 - \epsilon M_1 + \sum_{i=1}^{m} (\widehat{u_i} + R_{u_i} + 2\beta E_{\mathbf{xy}} + R'_{u_i} - R_{u_i} - 2\beta E_{xy}) f_i - 2\beta M_2 \\
=\ & 1 - \epsilon M_1 - 2\beta M_2 + \sum_{i=1}^{m} (\widehat{u_i} + R_{u_i} + 2\beta E_{\mathbf{xy}}) f_i + \sum_{i=1}^{m} (R'_{u_i} - R_{u_i} - 2\beta E_{\mathbf{xy}}) f_i \\
\geq\ & 1 - \epsilon M_1 - 2\beta M_2 + 0 - \sum_{i=1}^{m} (\epsilon M_1 + \epsilon M_1 + 2\beta M_2) M_{f_i} \\
=\ & 1 - (2\sum_{i=1}^{m} M_{f_i} + 1) M_1 \epsilon - 2(\sum_{i=1}^{m} M_{f_i} + 1) M_2 \beta.
\end{aligned}$$

Whence,

$$f_1 \geq 0 \wedge \cdots \wedge f_m \geq 0 \Rightarrow \widehat{h} \geq 1 - (2\sum_{i=1}^{m} M_{f_i} + 1) M_1 \epsilon - 2(\sum_{i=1}^{m} M_{f_i} + 1) M_2 \beta.$$

Let $S' = \{(\mathbf{x}, \mathbf{z}) \mid g_1 \geq 0 \wedge \cdots \wedge g_n \geq 0\}$, M_3 be an upper bound of $R_{2d,\mathbf{x},\mathbf{z}}$ on S', M_4 an upper bound of $E_d(\mathbf{x}, \mathbf{z})^T E_d(\mathbf{x}, \mathbf{z})$ on S', and M_{g_j} an upper bound of g_j on S'. Similarly, it follows

$$g_1 \geq 0 \wedge \cdots \wedge g_n \geq 0 \Rightarrow \quad -\widehat{h} \geq 1 - (2\sum_{j=1}^{n} M_{g_j} + 1) M_3 \epsilon - 2(\sum_{j=1}^{n} M_{g_j} + 1) M_4 \beta.$$

So, the following proposition is immediately.

Proposition 2. *There exist two positive constants γ_1 and γ_2 such that*

$$f_1 \geq 0 \wedge \cdots \wedge f_m \geq 0 \Rightarrow \widehat{h} \geq 1 - \gamma_1 \epsilon - \gamma_2 \beta, \tag{16}$$

$$g_1 \geq 0 \wedge \cdots \wedge g_n \geq 0 \Rightarrow -\widehat{h} \geq 1 - \gamma_1 \epsilon - \gamma_2 \beta. \tag{17}$$

Since ϵ and β heavily rely on the numerical tolerance and the floating point representation, it is easy to see that ϵ and β become small enough with $\gamma_1\epsilon < \frac{1}{2}$ and $\gamma_2\beta < \frac{1}{2}$, if the numerical tolerance is small enough and the length of the floating point representation is long enough. This implies

$$f_1 \geq 0 \wedge \cdots \wedge f_m \geq 0 \Rightarrow \widehat{h} > 0, \quad g_1 \geq 0 \wedge \cdots \wedge g_n \geq 0 \Rightarrow -\widehat{h} > 0.$$

If so, any numerical result $\widehat{h} > 0$ returned by calling an SDP solver to (12) is guaranteed to be a real interpolant for ϕ and ψ, i.e., a correct solution to Problem 1.

Example 3. Consider the numerical result for Example 2 in Sect. 4. Let M_{f_1}, M_{f_2}, M_{f_3}, M_{g_1}, M_{g_2}, M_{g_3}, M_1, M_2, M_3, M_4 are defined as above. It is easy to see that

$$f_1 \geq 0 \Rightarrow |x| \leq 2 \wedge |y| \leq 2 \wedge |z| \leq 2 \wedge |a_1| \leq 2 \wedge |b_1| \leq 2 \wedge |c_1| \leq 2 \wedge |d_1| \leq 2.$$

Then, by simple calculations, we obtain $M_{f_1} = 4, M_{f_2} = 32, M_{f_3} = 3, M_1 = 83, M_2 = 29$. Thus,

$$(2\sum_{i=1}^{m} M_{f_i} + 1)M_1 = 6557, \quad 2(\sum_{i=1}^{m} M_{f_i} + 1)M_2 = 2320.$$

Also, since

$$g_1 \geq 0 \Rightarrow |x| \leq 2 \wedge |y| \leq 2 \wedge |z| \leq 2 \wedge |a_2| \leq 2 \wedge |b_2| \leq 2 \wedge |c_2| \leq 2 \wedge |d_2| \leq 2,$$

we obtain $M_{g_1} = 4, M_{g_2} = 7, M_{g_3} = 2, M_3 = 83, M_4 = 29$. Thus,

$$(2\sum_{i=1}^{m} M_{g_i} + 1)M_3 = 2241, \quad 2(\sum_{i=1}^{m} M_{g_i} + 1)M_4 = 812.$$

Consequently, we have $\gamma_1 = 6557$ and $\gamma_2 = 2320$ in Proposition 2.

Due to the fact that the default error tolerance is 10^{-8} in the SDP solver Mosek and h is rounding to 4 decimal places, we have $\epsilon = \frac{10^{-4}}{2}$. In addition, as the absolute value of each element in $\widehat{C}$ is less than 10^3, and the dimension of D is less than 10^3, we obtain

$$\beta = \frac{(D+1)\kappa}{1-(2D+2)\kappa}\mathtt{tr}(\widetilde{C}) + 4(D+1)(2(D+2) + \max_i(\widetilde{C}_{ii}))\eta \leq 10^{-6}.$$

Consequently, $\gamma_1\epsilon \leq 6557 \cdot \frac{10^{-4}}{2} < \frac{1}{2}$, $\gamma_2\beta \leq 2320 \cdot 10^{-6} < \frac{1}{2}$, which imply that $h(x, y, z) > 0$ presented in Example 2 is indeed a real interpolant.

Remark 1. Besides, the result could be verified by the following symbolic computation procedure instead: computing $P_{\mathbf{x}}(\phi)$ and $P_{\mathbf{x}}(\psi)$ first by some symbolic tools, such as Redlog [8] which is a package that extends the computer algebra system REDUCE to a computer logic system; then verifying

$\mathbf{x} \in P_{\mathbf{x}}(\phi) \Rightarrow h(\mathbf{x}) > 0$ and $\mathbf{x} \in P_{\mathbf{x}}(\psi) \Rightarrow h(\mathbf{x}) < 0$. For this example, $P_{x,y,z}(\phi)$ and $P_{x,y,z}(\psi)$ obtained by Redlog are too complicated and therefore not presented here. The symbolic computation can verify that $h(x, y, z)$ in this example is exactly an interpolant, which confirms our conclusion. Alternatively, we can also solve the SDP in (9) using a SDP solver with infinite precision [15], and obtain an exact result. But this only works for problems with small size because a SDP solver with infinite precision is essentially based on symbolic computation as commented in [15].

6 Generalizing to General Polynomial Formulas

Problem 2. Let $\phi(\mathbf{x}, \mathbf{y})$ and $\psi(\mathbf{x}, \mathbf{z})$ be two polynomial formulas defined as follows,

$$\phi(\mathbf{x}, \mathbf{y}) : \bigvee_{i=1}^{m} \phi_i,\ \phi_i = \bigwedge_{k=1}^{K_i} f_{i,k}(\mathbf{x}, \mathbf{y}) \geq 0; \quad \psi(\mathbf{x}, \mathbf{z}) : \bigvee_{j=1}^{n} \psi_j,\ \psi_j = \bigwedge_{s=1}^{S_j} g_{j,s}(\mathbf{x}, \mathbf{z}) \geq 0,$$

where all $f_{i,k}$ and $g_{j,s}$ are polynomials. Suppose $\phi \wedge \psi \models \bot$, and for $i = 1, \ldots, m$, $j = 1, \ldots, n$, $\{(\mathbf{x}, \mathbf{y}) \mid \phi_i(\mathbf{x}, \mathbf{y})\}$ and $\{(\mathbf{x}, \mathbf{z}) \mid \psi_j(\mathbf{x}, \mathbf{z})\}$ are all semi-algebraic sets of the Archimedean form. Find a polynomial $h(\mathbf{x})$ such that $h(\mathbf{x}) > 0$ is an interpolant for ϕ and ψ.

Theorem 9. *For Problem 2, there exists a polynomial $h(\mathbf{x})$ satisfying*

$$\forall \mathbf{x} \in P_{\mathbf{x}}(\phi(\mathbf{x}, \mathbf{y})).h(\mathbf{x}) > 0, \quad \forall \mathbf{x} \in P_{\mathbf{x}}(\psi(\mathbf{x}, \mathbf{z})).h(\mathbf{x}) < 0.$$

Proof. We just need to prove that Lemma 1 holds for Problem 2 as well. Since $\{(\mathbf{x}, \mathbf{y}) \mid \phi_i(\mathbf{x}, \mathbf{y})\}$ and $\{(\mathbf{x}, \mathbf{z}) \mid \psi_j(\mathbf{x}, \mathbf{z})\}$ are all semi-algebraic sets of the Archimedean form, then $\{(\mathbf{x}, \mathbf{y}) \mid \phi(\mathbf{x}, \mathbf{y})\}$ and $\{(\mathbf{x}, \mathbf{z}) \mid \psi(\mathbf{x}, \mathbf{z})\}$ both are compact. See $\{(\mathbf{x}, \mathbf{y}) \mid \phi(\mathbf{x}, \mathbf{y})\}$ or $\{(\mathbf{x}, \mathbf{z}) \mid \psi(\mathbf{x}, \mathbf{z})\}$ as S in the proof of Lemma 1, then Lemma 1 holds for Problem 2. Thus, the rest of proof is same as that forTheorem 2. □

Corollary 3. *Let $\phi(\mathbf{x}, \mathbf{y})$ and $\psi(\mathbf{x}, \mathbf{z})$ be defined as in Problem 2. There must exist a polynomial $h(\mathbf{x})$ such that $h(\mathbf{x}) > 0$ is an interpolant for ϕ and ψ.*

Theorem 10. *Let $\phi(\mathbf{x}, \mathbf{y})$ and $\psi(\mathbf{x}, \mathbf{z})$ be defined as in Problem 2. Then there exists a polynomial $h(\mathbf{x})$ and $\sum_{i=1}^{m}(K_i + 1) + \sum_{j=1}^{n}(S_j + 1)$ sum of squares polynomials $u_{i,k}(\mathbf{x}, \mathbf{y})$ $(i = 1, \ldots, m,\ k = 1, \ldots, K_i + 1)$, $v_{j,s}(\mathbf{x}, \mathbf{z})$ $(j = 1, \ldots, n,\ s = 1, \ldots, S_j)$ satisfying the following semi-definite constraints such that $h(\mathbf{x}) > 0$ is an interpolant for $\phi(\mathbf{x}, \mathbf{y})$ and $\psi(\mathbf{x}, \mathbf{z})$:*

$$h - 1 = \sum_{k=1}^{K_i} u_{i,k} f_{i,k} + u_{i,K_i+1}, \quad i = 1, \ldots, m; \tag{18}$$

$$-h - 1 = \sum_{s=1}^{S_j} v_{j,s} g_{j,s} + v_{j,S_j+1}, \quad j = 1, \ldots, n. \tag{19}$$

Proof. By the property of Archimedean, the proof is same as that for T heorem 5. □

Similarly, Problem 2 can be equivalently reformulated as the problem of searching for sum of squares polynomials satisfying

$$\begin{cases} h(\mathbf{x}) - 1 - \sum_{k=1}^{K_i} u_{i,k} f_{i,k} \in \sum \mathbb{R}[\mathbf{x}, \mathbf{y}]^2, i = 1, \ldots, m; \\ -h(\mathbf{x}) - 1 - \sum_{s=1}^{S_j} v_{j,s} g_{j,s} \in \sum \mathbb{R}[\mathbf{x}, \mathbf{z}]^2, j = 1, \ldots, n; \\ u_{i,k} \in \sum \mathbb{R}[\mathbf{x}, \mathbf{y}]^2, i = 1, \ldots, m, k = 1, \ldots, K_i; \\ v_{j,s} \in \sum \mathbb{R}[\mathbf{x}, \mathbf{z}]^2, j = 1, \ldots, n, s = 1, \ldots, S_j. \end{cases} \tag{20}$$

Example 4. Consider

$$\phi(x, y, a_1, a_2, b_1, b_2) : (f_1 \geq 0 \wedge f_2 \geq 0) \vee (f_3 \geq 0 \wedge f_4 \geq 0),$$
$$\psi(x, y, c_1, c_2, d_1, d_2) : (g_1 \geq 0 \wedge g_2 \geq 0) \vee (g_3 \geq 0 \wedge g_4 \geq 0),$$

where

$$\begin{aligned} f_1 &= 16 - (x + y - 4)^2 - 16(x - y)^2 - a_1^2, & f_2 &= x + y - a_2^2 - (2 - a_2)^2, \\ f_3 &= 16 - (x + y + 4)^2 - 16(x - y)^2 - b_1^2, & f_4 &= -x - y - b_2^2 - (2 - b_2)^2, \\ g_1 &= 16 - 16(x + y)^2 - (x - y + 4)^2 - c_1^2, & g_2 &= y - x - c_2^2 - (1 - c_2)^2, \\ g_3 &= 16 - 16(x + y)^2 - (x - y - 4)^2 - d_1^2, & g_4 &= x - y - d_2^2 - (1 - d_2)^2. \end{aligned}$$

We get a concrete SDP problem of the form (20) by setting the degree of $h(x, y)$ in (20) to be 2. Using the MATLAB package YALMIP and Mosek, we obtain

$$h(x, y) = -2.3238 + 0.6957x^2 + 0.6957y^2 + 7.6524xy.$$

The result is plotted in Fig. 3, and can be verified either by numerical error analysis as in Example 2 or by a symbolic procedure like REDUCE as described in Remark 1.

Example 5 (Ultimate). Consider the following example taken from [5], which is a challenging benchmark to existing approaches for nonlinear interpolant generation.

$$\phi = (f_1 \geq 0 \wedge f_2 \geq 0 \vee f_3 \geq 0) \wedge f_4 \geq 0 \wedge f_5 \geq 0 \vee f_6 \geq 0,$$
$$\psi = (g_1 \geq 0 \wedge g_2 \geq 0 \vee g_3 \geq 0) \wedge g_4 \geq 0 \wedge g_5 \geq 0 \vee g_6 \geq 0,$$

where

$$\begin{aligned} f_1 &= 3.8025 - x^2 - y^2, & f_2 &= y, \\ f_3 &= 0.9025 - (x - 1)^2 - y^2, & f_4 &= (x - 1)^2 + y^2 - 0.09, \\ f_5 &= (x + 1)^2 + y^2 - 1.1025, & f_6 &= 0.04 - (x + 1)^2 - y^2, \\ g_1 &= 3.8025 - x^2 - y^2, & g_2 &= -y, \\ g_3 &= 0.9025 - (x + 1)^2 - y^2, & g_4 &= (x + 1)^2 + y^2 - 0.09, \\ g_5 &= (x - 1)^2 + y^2 - 1.1025, & g_6 &= 0.04 - (x - 1)^2 - y^2. \end{aligned}$$

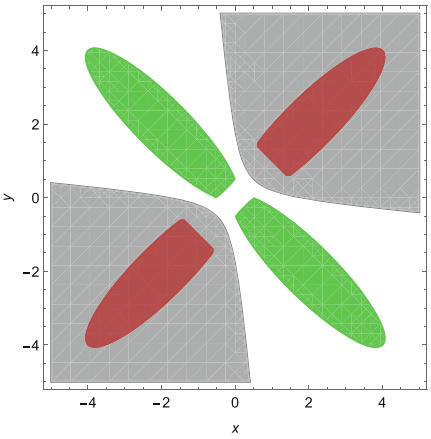


Fig. 3. Example 4. (Red region: $P_{x,y}(\phi(x, y, a_1, a_2, b_1, b_2))$; green region: $P_{x,y}(\psi(x, y, c_1, c_2, d_1, d_2))$; gray region: $\{(x, y) \mid h(x, y) > 0\}$.) (Color figure online)

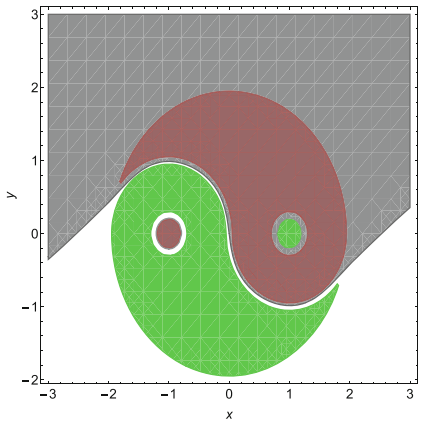


Fig. 4. Example 5. (Red region: $P_{x,y}(\phi(x, y))$; green region: $P_{x,y}(\psi(x, y))$; gray region: $\{(x, y) \mid h(x, y) > 0\}$.) (Color figure online)

We first convert ϕ and ψ to the disjunction normal form as:

$$\phi = (f_1 \geq 0 \wedge f_2 \geq 0 \wedge f_4 \geq 0 \wedge f_5 \geq 0) \vee (f_3 \geq 0 \wedge f_4 \geq 0 \wedge f_5 \geq 0) \vee (f_6 \geq 0),$$
$$\psi = (g_1 \geq 0 \wedge g_2 \geq 0 \wedge g_4 \geq 0 \wedge g_5 \geq 0) \vee (g_3 \geq 0 \wedge g_4 \geq 0 \wedge g_5 \geq 0) \vee (g_6 \geq 0).$$

We get a concrete SDP problem of the form (20) by setting the degree of $h(x, y)$ in (20) to be 7. Using the MATLAB package YALMIP and Mosek, keeping the decimal to four, we obtain

$$\begin{aligned} h(x, y) = {} & 1297.5980x + 191.3260y - 3172.9653x^3 + 196.5763x^2y + 2168.1739xy^2 \\ & + 1045.7373y^3 + 1885.8986x^5 - 1009.6275x^4y + 3205.3793x^3y^2 - 1403.5431x^2y^3 \\ & + 1842.0669xy^4 + 1075.2003y^5 - 222.0698x^7 + 547.9542x^6y - 704.7474x^5y^2 \\ & + 1724.7008x^4y^3 - 728.2229x^3y^4 + 1775.7548x^2y^5 - 413.3771xy^6 + 1210.2617y^7. \end{aligned}$$

The result is plotted in Fig. 4, and can be verified either by numerical error analysis as in Example 2 or by a symbolic procedure like REDUCE as described in Remark 1.

7 Application to Invariant Generation

In this section, as an application, we sketch how to apply our approach to invariant generation in program verification, the details can be found in [11].

In [22], Lin *et al.* proposed a framework for invariant generation using *weakest precondition*, *strongest postcondition* and *interpolation*, which consists of two procedures, i.e., synthesizing invariants by forward interpolation based on *strongest*

postcondition and *interpolant generation*, and by backward interpolation based on *weakest precondition* and *interpolant generation*. In [22], only linear invariants can be synthesized as no powerful approaches are available to synthesize nonlinear interpolants. Obviously, our results can strengthen their framework by allowing to generate nonlinear invariants. For example, we can revise the procedure Squeezing Invariant - Forward in their framework and obtain Algorithm 1.

The major revisions include:

- firstly, we exploit our method to synthesize interpolants see line 4 in Algorithm 1;
- secondly, we add a conditional statement for A_{i+1} at line 7–10 in Algorithm 1 in order to make A_{i+1} to be Archimedean.

The procedure Squeezing Invariant - backward can be revised similarly.

Algorithm 1. Revised Squeezing Invariant - Forward

Input: An annotated loop: $\{P\}$ while ρ do C $\{Q\}$, where P and Q are Archimedean
Output: (yes/no, $\mathcal{I}$), where $\mathcal{I}$ is a loop invariant

1: $A_0 \leftarrow P$; $B_0 \leftarrow (\neg\rho \wedge \neg Q)$; $i \leftarrow 0$; $j \leftarrow 0$;
2: **while** $\top$ **do**
3: **if** $(\bigvee_{k=0}^{i} A_i) \wedge B_j$ is not satisfiable, $(\bigvee_{k=0}^{i} A_i)$ and B_j are Archimedean **then**
4: call our method to synthesize an interpolant for $(\bigvee_{k=0}^{i} A_i)$ and B_j, say $\mathcal{I}_i$; {Use our method to generate interpolant}
5: **if** $\{\mathcal{I}_i \wedge \rho\}\, C\, \{\mathcal{I}_i\}$ **then**
6: **return** (yes, $\mathcal{I}_i$);
7: **else if** $\mathcal{I}_i$ is Archimedean **then**
8: $A_{i+1} \leftarrow \mathsf{sp}(\mathcal{I}_i \wedge \rho, C)$;
9: **else**
10: $A_{i+1} \leftarrow \mathsf{sp}(A_i \wedge \rho, C)$;
11: **end if**
{sp: a predicate transformer to compute the strongest postcondition of C w.r.t. $\mathcal{I}_i \wedge \rho$}
12: $i \leftarrow i+1$; $B_{j+1} \leftarrow B_0 \vee (\rho \wedge \mathsf{wp}(C, B_j))$;
{wp: a predicate transformer to compute the weakest precondition of C w.r.t. B_j}
13: $j \leftarrow j+1$;
14: **else if** A_i is concrete **then**
15: **return** (no, $\bot$);
16: **else**
17: **while** A_i is not concrete **do**
18: $i \leftarrow i-1$;
19: **end while**
20: $A_{i+1} \leftarrow \mathsf{sp}(A_i \wedge \rho, C)$; $i \leftarrow i+1$;
21: **end if**
22: **end while**

Example 6. Consider a loop program given in Algorithm 2 for controlling the acceleration of a car adapted from [21]. Suppose we know that vc is in $[0, 40]$ at the beginning of the loop, we would like to prove that $vc < 49.61$ holds after the loop. Since the loop guard is unknown, it means that the loop may terminate after any number of iterations.

We apply Algorithm 1 to the computation of an invariant to ensure that $vc < 49.61$ holds. Since vc is the velocity of car, $0 \leq vc < 49.61$ is required to hold in order to maintain safety. Via Algorithm 1, we have $A_0 = \{vc \mid vc(40-vc) \geq 0\}$ and $B = \{vc \mid vc < 0\} \cup \{vc \mid vc \geq 49.61\}$. Here, we replace B with $B' = [-2, -1] \cup [49.61, 55])$, i.e., $B' = \{vc \mid (vc+2)(-1-vc) \geq 0 \vee (vc-49.61)(55-vc) \geq 0\}$, in order to make it with Archimedean form.

Firstly, it is evident that $A_0 : vc(40 - vc) \geq 0$ implies $A_0 \wedge B' \models \bot$. By applying our approach, we obtain an interpolant

$$\mathcal{I}_0 : 1.4378 + 3.3947 * vc - 0.083 * vc^2 > 0$$

for A_0 and B'. It can be verified that $\{\mathcal{I}_0\}\, C\, \{\mathcal{I}_0\}$ (line 5) does not hold, where C stands for the loop body.

Secondly, by setting $A_1 = sp(\mathcal{I}_0, C)$ (line 8) and re-calling our approach, we obtain an interpolant

$$\mathcal{I}_1 : 2.0673 + 3.0744 * vc - 0.0734 * vc^2 > 0$$

for $A_0 \cup A_1$ and B'. Likewise, it can be verified that $\{\mathcal{I}_1\}\, C\, \{\mathcal{I}_1\}$ (line 5) does not hold.

Algorithm 2. Control code for accelerating a car

```
1: /* Pre: vc ∈ [0, 40] */
2: while unknown do
3:     fa ← 0.5418 * vc * vc;
4:     fr ← 1000 − fa;
5:     ac ← 0.0005 * fr;
6:     vc ← vc + ac;
7: end while
8: /* Post: vc < 49.61 */
```

Thirdly, repeating the above procedure again, we obtain an interpolant

$$\mathcal{I}_2 : 2.2505 + 2.7267 * vc - 0.063 * vc^2 > 0,$$

and it can be verified that $\{\mathcal{I}_2\}\, C\, \{\mathcal{I}_2\}$ holds, implying that $\mathcal{I}_2$ is an invariant. Moreover, it is trivial to verify that $\mathcal{I}_2 \Rightarrow vc < 49.61$.

Consequently, we have the conclusion that $\mathcal{I}_2$ is an inductive invariant which witnesses the correctness of the loop.

8 Conclusion

In this paper we propose a sound and complete method to synthesize Craig interpolants for mutually contradictory polynomial formulas $\phi(\mathbf{x}, \mathbf{y})$ and $\psi(\mathbf{x}, \mathbf{z})$, with the form $f_1 \geq 0 \wedge \cdots \wedge f_n \geq 0$, where f_i's are polynomials in $\mathbf{x}, \mathbf{y}$ or $\mathbf{x}, \mathbf{z}$ and the quadratic module generated by f_i's is Archimedean. The interpolant is generated by solving a semi-definite programming problem, which is a generalization of the method in [7] dealing with mutually contradictory formulas with the same set of variables and the method in [10] dealing with mutually contradictory formulas with concave quadratic polynomial inequalities. As an application, we apply our approach to invariant generation in program verification.

As a future work, we would like to consider interpolant synthesizing for formulas with strict polynomial inequalities. Also, it deserves to consider how to synthesize interpolants for the combination of non-linear formulas and other theories based on our approach and other existing ones, as well as further applications to the verification of programs and hybrid systems.

Acknowledgments. We thank Dr. Sicun Gao, Dr. Damien Zufferey and Dr. Mingshuai Chen for their help on using dReal. We are indebted to anonymous reviewers for their detailed and constructive criticisms and comments on the preliminary version, which help to improve the presentation of this paper very much. This work is financially supported by NSFC under grants 61902284 (the first author), 61732001 (the second and fourth authors), 61532019 (the second author), 61836005 (the third author), 61625206 (the fourth author) and 61802318 (the fifth author), and the third author is also in part supported by Hundred Talents Program under grant No. Y8YC235015.

References

1. Benhamou, F., Granvilliers, L.: Continuous and interval constraints. In: Handbook of Constraint Programming. Foundations of Artificial Intelligence, vol. 2, pp. 571–603 (2006)
2. Bierstone, E., Milman, P.D.: Semianalytic and subanalytic sets. Publications Mathematiques de l'IHÉS **67**, 5–42 (1988)
3. Bochnak, J., Coste, M., Roy, M.: Real Algebraic Geometry. Springer, Heidelberg (1998). https://doi.org/10.1007/978-3-662-03718-8
4. Borchers, B.: CSDP, a C library for semidefinite programming. Optim. Methods Softw. **11**(1–4), 613–623 (1999). http://projects.coin-or.org/csdp/
5. Chen, M., Wang, J., An, J., Zhan, B., Kapur, D., Zhan, N.: NIL: learning nonlinear interpolants. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 178–196. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_11
6. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient interpolant generation in satisfiability modulo theories. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 397–412. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_30
7. Dai, L., Xia, B., Zhan, N.: Generating non-linear interpolants by semidefinite programming. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 364–380. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_25

8. Dolzmann, A., Sturm, T.: REDLOG: computer algebra meets computer logic. ACM SIGSAM Bull. **31**(2), 2–9 (1997)
9. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 129–145. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11319-2_12
10. Gan, T., Dai, L., Xia, B., Zhan, N., Kapur, D., Chen, M.: Interpolant synthesis for quadratic polynomial inequalities and combination with *EUF*. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 195–212. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_14
11. Gan, T., Xia, B., Xue, B., Zhan, N.: Nonlinear Craig interpolant generation. CoRR, abs/1903.01297 (2019)
12. Gao, S., Kong, S., Clarke, E.: Proof generation from delta-decisions. In: SYNASC 2014, pp. 156–163 (2014)
13. Gao, S., Zufferey, D.: Interpolants in nonlinear theories over the reals. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 625–641. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_41
14. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_10
15. Henrion, D., Naldi, S., Safey El Din, M.: Exact algorithms for semidefinite programs with degenerate feasible set. In: ISSAC 2018, pp. 191–198 (2018)
16. Henzinger, T., Jhala, R., Majumdar, R., McMillan, K.: Abstractions from proofs. In: POPL 2004, pp. 232–244 (2004)
17. Jung, Y., Lee, W., Wang, B.-Y., Yi, K.: Predicate generation for learning-based quantifier-free loop invariant inference. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 205–219. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_17
18. Kapur, D., Majumdar, R., Zarba, C.: Interpolation for data structures. In: FSE 2006, pp. 105–116 (2006)
19. Kovács, L., Voronkov, A.: Interpolation and symbol elimination. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 199–213. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_17
20. Krajíček, J.: Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. J. Symbol. Logic **62**(2), 457–486 (1997)
21. Kupferschmid, S., Becker, B.: Craig interpolation in the presence of non-linear constraints. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 240–255. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24310-3_17
22. Lin, S., Sun, J., Xiao, H., Sanán, D., Hansen, H.: FiB: squeezing loop invariants by interpolation between forward/backward predicate transformers. In: ASE 2017, pp. 793–803 (2017)
23. Lofberg., J.: YALMIP: a toolbox for modeling and optimization in MATLAB. In: CACSD 2004, pp. 284–289. IEEE (2004)
24. Marshall, M.: Positive Polynomials and Sums of Squares. American Mathematical Society, Providence (2008)
25. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_1
26. McMillan, K.: An interpolating theorem prover. Theoret. Comput. Sci. **345**(1), 101–121 (2005)

27. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_31
28. Mosek, A.: The MOSEK optimization toolbox for MATLAB manual. Version 7.1 (Revision 28), p. 17 (2015)
29. Mostowski, T.: Some properties of the ring of nash functions. Annali della Scuola Normale Superiore di Pisa **3**(2), 245–266 (1976)
30. Pudlăk, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. J. Symbol. Logic **62**(3), 981–998 (1997)
31. Putinar, M.: Positive polynomials on compact semi-algebraic sets. Indiana Univ. Math. J. **42**(3), 969–984 (1993)
32. Roux, P., Voronin, Y.-L., Sankaranarayanan, S.: Validating numerical semidefinite programming solvers for polynomial invariants. Formal Methods Syst. Des. **53**(2), 286–312 (2017). https://doi.org/10.1007/s10703-017-0302-y
33. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint solving for interpolation. J. Symb. Comput. **45**(11), 1212–1233 (2010)
34. Sofronie-Stokkermans, V.: Interpolation in local theory extensions. In: Logical Methods in Computer Science, vol. 4, no. 4 (2008)
35. Srikanth, A., Sahin, B., Harris, W.: Complexity verification using guided theorem enumeration. In: POPL 2017, pp. 639–652 (2017)
36. Stengle, G.: A nullstellensatz and a positivstellensatz in semialgebraic geometry. Ann. Math. **207**, 87–97 (1974)
37. Yorsh, G., Musuvathi, M.: A combination method for generating interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 353–368. Springer, Heidelberg (2005). https://doi.org/10.1007/11532231_26
38. Zhan, N., Wang, S., Zhao, H.: Formal Verification of Simulink/Stateflow Diagrams. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-47016-0
39. Zhao, H., Zhan, N., Kapur, D., Larsen, K.G.: A "hybrid" approach for synthesizing optimal controllers of hybrid systems: a case study of the oil pump industrial example. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 471–485. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_38

Approximate Counting of Minimal Unsatisfiable Subsets

Jaroslav Bendík[1(✉)] and Kuldeep S. Meel[2]

[1] Masaryk University, Brno, Czech Republic
xbendik@gmail.com
[2] National University of Singapore, Singapore, Singapore

Abstract. Given an unsatisfiable formula F in CNF, i.e. a set of clauses, the problem of Minimal Unsatisfiable Subset (MUS) seeks to identify a minimal subset of clauses $N \subseteq F$ such that N is unsatisfiable. The emerging viewpoint of MUSes as the root causes of unsatisfiability has led MUSes to find applications in a wide variety of diagnostic approaches. Recent advances in identification and enumeration of MUSes have motivated researchers to discover applications that can benefit from rich information about the set of MUSes. One such extension is that of counting the number of MUSes. The current best approach for MUS counting is to employ a MUS enumeration algorithm, which often does not scale for the cases with a reasonably large number of MUSes.

Motivated by the success of hashing-based techniques in the context of model counting, we design the first approximate MUS counting procedure with (ε, δ) guarantees, called AMUSIC. Our approach avoids exhaustive MUS enumeration by combining the classical technique of universal hashing with advances in QBF solvers along with a novel usage of union and intersection of MUSes to achieve runtime efficiency. Our prototype implementation of AMUSIC is shown to scale to instances that were clearly beyond the realm of enumeration-based approaches.

1 Introduction

Given an unsatisfiable Boolean formula F as a set of clauses $\{f_1, f_2, \ldots f_n\}$, also known as conjunctive normal form (CNF), a set N of clauses is a Minimal Unsatisfiable Subset (MUS) of F iff $N \subseteq F$, N is unsatisfiable, and for each $f \in N$ the set $N \setminus \{f\}$ is satisfiable. Since MUSes can be viewed as representing the *minimal reasons* for unsatisfiability of a formula, MUSes have found applications in wide variety of domains ranging from diagnosis [45], ontologies debugging [1], spreadsheet debugging [29], formal equivalence checking [20], constrained counting and sampling [28], and the like. As the scalable techniques for identification of MUSes appeared only about decade and half ago, the earliest applications primarily focused on a reduction to the identification of a single MUS or a small set of MUSes. With an improvement in the scalability of MUS identification techniques, researchers have now sought to investigate extensions of MUSes

Work done in part while the first author visited National University of Singapore.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 439–462, 2020.
https://doi.org/10.1007/978-3-030-53288-8_21

and their corresponding applications. One such extension is MUS counting, i.e., counting the number of MUSes of F. Hunter and Konieczny [26], Mu [45], and Thimm [56] have shown that the number of MUSes can be used to compute different inconsistency metrics for general propositional knowledge bases.

In contrast to the progress in the design of efficient MUS identification techniques, the work on MUS counting is still in its nascent stages. Reminiscent of the early days of model counting, the current approach for MUS counting is to employ a complete MUS enumeration algorithm, e.g., [3,12,34,55], to explicitly identify all MUSes. As noted in Sect. 2, there can be up to exponentially many MUSes of F w.r.t. $|F|$, and thus their complete enumeration can be practically intractable. Indeed, contemporary MUS enumeration algorithms often cannot complete the enumeration within a reasonable time [10,12,34,47]. In this context, one wonders: *whether it is possible to design a scalable MUS counter without performing explicit enumeration of MUSes?*

The primary contribution of this paper is a probabilistic counter, called AMUSIC, that takes in a formula F, tolerance parameter ε, confidence parameter δ, and returns an estimate guaranteed to be within $(1+\varepsilon)$-multiplicative factor of the exact count with confidence at least $1-\delta$. Crucially, for F defined over n clauses, AMUSIC explicitly identifies only $\mathcal{O}(\log n \cdot \log(1/\delta) \cdot (\varepsilon)^{-2})$ many MUSes even though the number of MUSes can be exponential in n.

The design of AMUSIC is inspired by recent successes in the design of efficient XOR hashing-based techniques [15,17] for the problem of model counting, i.e., given a Boolean formula G, compute the number of models (also known as solutions) of G. We observe that both the problems are defined over a power-set structure. In MUS counting, the goal is to count MUSes in the power-set of F, whereas in model counting, the goal is to count models in the power-set that represents all valuations of variables of G. Chakraborty et al. [18,52] proposed an algorithm, called ApproxMC, for approximate model counting that also provides the (ϵ, δ) guarantees. ApproxMC is currently in its third version, ApproxMC3 [52]. The base idea of ApproxMC3 is to partition the power-set into *nCells* small cells, then pick one of the cells, and count the number *inCell* of models in the cell. The total model count is then estimated as $nCells \times inCell$. Our algorithm for MUS counting is based on ApproxMC3. We adopt the high-level idea to partition the power-set of F into small cells and then estimate the total MUS count based on a MUS count in a single cell. The difference between ApproxMC3 and AMUSIC lies in the way of counting the target elements (models vs. MUSes) in a single cell; we propose novel MUS specific techniques to deal with this task. In particular, our contribution is the following:

- We introduce a QBF (quantified Boolean formula) encoding for the problem of counting MUSes in a single cell and use a Σ_3^P oracle to solve it.
- Let $\texttt{UMU}_F$ and $\texttt{IMU}_F$ be the union and the intersection of all MUSes of F, respectively. We observe that every MUS of F (1) contains $\texttt{IMU}_F$ and (2) is contained in $\texttt{UMU}_F$. Consequently, if we determine the sets $\texttt{UMU}_F$ and $\texttt{IMU}_F$, then we can significantly speed up the identification of MUSes in a cell.

- We propose a novel approaches for computing the union $\texttt{UMU}_F$ and the intersection $\texttt{IMU}_F$ of all MUSes of F.
- We implement AMUSIC and conduct an extensive empirical evaluation on a set of *scalable* benchmarks. We observe that AMUSIC is able to compute estimates for problems clearly beyond the reach of existing enumeration-based techniques. We experimentally evaluate the *accuracy* of AMUSIC. In particular, we observe that the estimates computed by AMUSIC are significantly closer to true count than the theoretical guarantees provided by AMUSIC.

Our work opens up several new interesting avenues of research. From a theoretical perspective, we make polynomially many calls to a Σ_3^P oracle while the problem of finding a MUS is known to be in FP^{NP}, i.e. a MUS can be found in polynomial time by executing a polynomial number of calls to an NP-oracle [19,39]. Contrasting this to model counting techniques, where approximate counter makes polynomially many calls to an NP-oracle when the underlying problem of finding satisfying assignment is NP-complete, a natural question is to close the gap and seek to design a MUS counting algorithm with polynomially many invocations of an FP^{NP} oracle. From a practitioner perspective, our work calls for a design of MUS techniques with native support for XORs; the pursuit of native support for XOR in the context of SAT solvers have led to an exciting line of work over the past decade [52,53].

2 Preliminaries and Problem Formulation

A Boolean formula $F = \{f_1, f_2, \ldots, f_n\}$ in a conjunctive normal form (CNF) is a set of Boolean clauses over a set of Boolean variables $Vars(F)$. A Boolean clause is a set $\{l_1, l_2, \ldots, l_k\}$ of literals. A literal is either a variable $x \in Vars(F)$ or its negation $\neg x$. A truth assignment I to the variables $Vars(F)$ is a mapping $Vars(F) \to \{1, 0\}$. A clause $f \in F$ is satisfied by an assignment I iff $I(l) = 1$ for some $l \in f$ or $I(k) = 0$ for some $\neg k \in f$. The formula F is satisfied by I iff I satisfies every $f \in F$; in such a case I is called a *model* of F. Finally, F is *satisfiable* if it has a model; otherwise F is *unsatisfiable*.

A QBF is a Boolean formula where each variable is either universally ($\forall$) or existentially ($\exists$) quantified. We write $Q_1 \cdots Q_k$-QBF, where $Q_1, \ldots Q_k \in \{\forall, \exists\}$, to denote the class of QBF with a particular type of alternation of the quantifiers, e.g., $\exists\forall$-QBF or $\exists\forall\exists$-QBF. Every QBF is either true (valid) or false (invalid). The problem of deciding validity of a formula in $Q_1 \cdots Q_k$-QBF where $Q_1 = \exists$ is Σ_k^P-complete [43].

When it is clear from the context, we write just *formula* to denote either a QBF or a Boolean formula in CNF. Moreover, throughout the whole text, we use F to denote the input Boolean Formula in CNF. Furthermore, we will use capital letters, e.g., S, K, N, to denote other CNF formulas, small letters, e.g., f, f_1, f_i, to denote clauses, and small letters, e.g., x, x', y, to denote variables.

Given a set X, we write $\mathcal{P}(X)$ to denote the power-set of X, and $|X|$ to denote the cardinality of X. Finally, we write $Pr[O : \mathbb{P}]$ to denote the probability of an

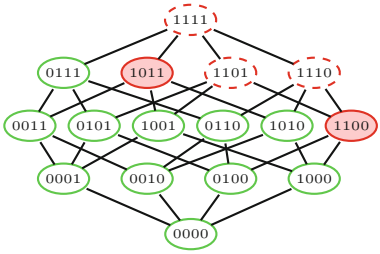


Fig. 1. Illustration of the power set of the formula F from the Example 1. We denote individual subsets of F using the bit-vector representation. The subsets with a dashed border are the unsatisfiable subsets, and the others are satisfiable subsets. The MUSes are filled with a background color. (Color figure online)

outcome O when sampling from a probability space $\mathbb{P}$. When $\mathbb{P}$ is clear from the context, we write just $Pr[O]$.

Minimal Unsatisfiability

Definition 1 (MUS). *A set N, $N \subseteq F$, is a* minimal unsatisfiable subset (MUS) *of F iff N is unsatisfiable and for all $f \in N$ the set $N \setminus \{f\}$ is satisfiable.*

Note that the minimality concept used here is set minimality, not minimum cardinality. Therefore, there can be MUSes with different cardinalities. In general, there can be up to exponentially many MUSes of F w.r.t. $|F|$ (see the Sperner's theorem [54]). We use $\texttt{AMU}_F$ to denote the set of all MUSes of F. Furthermore, we write $\texttt{UMU}_F$ and $\texttt{IMU}_F$ to denote the union and the intersection of all MUSes of F, respectively. Finally, note that every subset S of F can be expressed as a bit-vector over the alphabet $\{0, 1\}$; for example, if $F = \{f_1, f_2, f_3, f_4\}$ and $S = \{f_1, f_4\}$, then the bit-vector representation of S is 1001.

Definition 2. *Let N be an unsatisfiable subset of F and $f \in N$. The clause f is* necessary *for N iff $N \setminus \{f\}$ is satisfiable.*

The necessary clauses are sometimes also called *transition* [6] or *critical* [2] clauses. Note that a set N is a MUS iff every $f \in N$ is necessary for N. Also, note that a clause $f \in F$ is necessary for F iff $f \in \texttt{IMU}_F$.

Example 1. We demonstrate the concepts on an example, illustrated in Fig. 1. Assume that $F = \{f_1 = \{x_1\}, f_2 = \{\neg x_1\}, f_3 = \{x_2\}, f_4 = \{\neg x_1, \neg x_2\}\}$. In this case, $\texttt{AMU}_F = \{\{f_1, f_2\}, \{f_1, f_3, f_4\}\}$, $\texttt{IMU}_F = \{f_1\}$, and $\texttt{UMU}_F = F$.

Hash Functions

Let n and m be positive integers such that $m < n$. By $\{1, 0\}^n$ we denote the set of all bit-vectors of length n over the alphabet $\{1, 0\}$. Given a vector $v \in \{1, 0\}^n$

and $i \in \{1, \ldots, n\}$, we write $v[i]$ to denote the i-th bit of v. A hash function h from a family $H_{xor}(n, m)$ of hash functions maps $\{1, 0\}^n$ to $\{1, 0\}^m$. The family $H_{xor}(n, m)$ is defined as $\{h \mid h(y)[i] = a_{i,0} \oplus (\bigoplus_{k=1}^{n}(a_{i,k} \wedge y[k]))$ *for all* $1 \leq i \leq m\}$, where $\oplus$ and $\wedge$ denote the Boolean XOR and AND operators, respectively, and $a_{i,k} \in \{1, 0\}$ for all $1 \leq i \leq m$ and $1 \leq k \leq n$.

To choose a hash function uniformly at random from $H_{xor}(n, m)$, we randomly and independently choose the values of $a_{i,k}$. It has been shown [24] that the family $H_{xor}(n, m)$ is pairwise independent, also known as strongly 2-universal. In particular, let us by $h \leftarrow H_{xor}(n, m)$ denote the probability space obtained by choosing a hash function h uniformly at random from $H_{xor}(n, m)$. The property of pairwise independence guarantees that for all $\alpha_1, \alpha_2 \in \{1, 0\}^m$ and for all distinct $y_1, y_2 \in \{1, 0\}^n$, $Pr[\bigwedge_{i=1}^{2} h(y_i) = \alpha_i : h \leftarrow H_{xor}(n, m)] = 2^{-2m}$.

We say that a hash function $h \in H_{xor}(n, m)$ *partitions* $\{0, 1\}^n$ into 2^m *cells*. Furthermore, given a hash function $h \in H_{xor}(n, m)$ and a cell $\alpha \in \{1, 0\}^m$ of h, we define their *prefix-slices*. In particular, for every $k \in \{1, \ldots, m\}$, the k^{th} *prefix* of h, denoted $h^{(k)}$, is a map from $\{1, 0\}^n$ to $\{1, 0\}^k$ such that $h^{(k)}(y)[i] = h(y)[i]$ for all $y \in \{1, 0\}^n$ and for all $i \in \{1, \ldots, k\}$. Similarly, the k^{th} prefix of α, denoted $\alpha^{(k)}$, is an element of $\{1, 0\}^k$ such that $\alpha^{(k)}[i] = \alpha[i]$ for all $i \in \{1, \ldots, k\}$. Intuitively, a cell $\alpha^{(k)}$ of $h^{(k)}$ originates by merging the two cells of $h^{(k+1)}$ that differ only in the last bit.

In our work, we use hash functions from the family $H_{xor}(n, m)$ to partition the power-set $\mathcal{P}(F)$ of the given Boolean formula F into 2^m cells. Furthermore, given a cell $\alpha \in \{0, 1\}^m$, let us by $\texttt{AMU}_{\langle F,h,\alpha\rangle}$ denote the set of all MUSes in the cell α; formally, $\texttt{AMU}_{\langle F,h,\alpha\rangle} = \{M \in \texttt{AMU}_F \mid h(bit(M)) = \alpha\}$, where $bit(M)$ is the bit-vector representation of M. The following observation is crucial for our work.

Observation 1. *For every formula* F, $m \in \{1, \ldots, |F| - 1\}$, $h \in H_{xor}(|F|, m)$, *and* $\alpha \in \{0, 1\}^m$ *it holds that:* $\texttt{AMU}_{\langle F,h^{(i)},\alpha^{(i)}\rangle} \supseteq \texttt{AMU}_{\langle F,h^{(j)},\alpha^{(j)}\rangle}$ *for every* $i < j$.

Example 2. Assume that we are given a formula F such that $|F| = 4$ and a hash function $h \in H_{xor}(4, 2)$ that is defined via the following values of individual $a_{i,k}$:

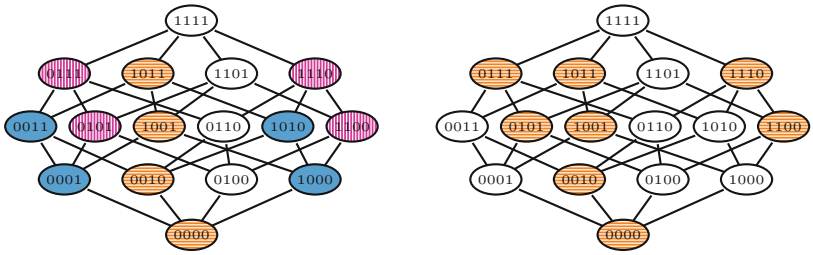
$$\begin{array}{lllll} a_{1,0} = 0, & a_{1,1} = 1, & a_{1,2} = 1, & a_{1,3} = 0, & a_{1,4} = 1 \\ a_{2,0} = 0, & a_{2,1} = 1, & a_{2,2} = 0, & a_{2,3} = 0, & a_{2,4} = 1 \end{array}$$

The hash function partitions $\mathcal{P}(F)$ into 4 cells. For example, $h(1100) = 01$ since $h(1100)[1] = 0 \oplus (1 \wedge 1) \oplus (1 \wedge 1) \oplus (0 \wedge 0) \oplus (1 \wedge 0) = 0$ and $h(1100)[2] = 0 \oplus (1 \wedge 1) \oplus (0 \wedge 1) \oplus (0 \wedge 0) \oplus (1 \wedge 0) = 1$. Figure 2 illustrates the whole partition and also illustrates the partition given by the prefix $h^{(1)}$ of h.

2.1 Problem Definitions

In this paper, we are concerned with the following problems.

Name: (ϵ, δ)-#MUS problem
Input: A formula F, a tolerance $\epsilon > 0$, and a confidence $1 - \delta \in (0, 1]$.
Output: A number c such that $Pr[|\texttt{AMU}_F|/(1+\epsilon) \leq c \leq |\texttt{AMU}_F| \cdot (1+\epsilon)] \geq 1 - \delta$.



(a) Illustration of $h^{(2)} = h$ with 4 cells: $\alpha_1 = 00$, $\alpha_2 = 01$, $\alpha_3 = 10$, $\alpha_4 = 11$.

(b) Illustration of $h^{(1)}$ with 2 cells: $\alpha_1 = 0$, $\alpha_2 = 1$.

Fig. 2. Illustration of the partition of $\mathcal{P}(F)$ by $h = h^{(2)}$ and $h^{(1)}$ from Example 2. In the case of h, we use 4 colors, orange, pink, white, and blue, to highlight its four cells. In case of $h^{(1)}$, there are only two cells: the white and the blue cells are merged into a white cell, and the pink and the orange cells are merged into an orange cell. (Color figure online)

Name: `MUS-membership` problem
Input: A formula F and a clause $f \in F$.
Output: *True* if there is a MUS $M \in \texttt{AMU}_F$ such that $f \in M$ and *False* otherwise.

Name: `MUS-union` problem
Input: A formula F.
Output: The union $\texttt{UMU}_F$ of all MUSes of F.

Name: `MUS-intersection` problem
Input: A formula F.
Output: The intersection $\texttt{IMU}_F$ of all MUSes of F.

Name: (ϵ, δ)`-#SAT` problem
Input: A formula F, a tolerance $\epsilon > 0$, and a confidence $1 - \delta \in (0, 1]$.
Output: A number m such that $Pr[m/(1+\epsilon) \leq c \leq m \cdot (1+\epsilon)] \geq 1 - \delta$, where m is the number of models of F.

The main goal of this paper is to provide a solution to the (ϵ, δ)`-#MUS` problem. We also deal with the `MUS-membership`, `MUS-union` and `MUS-intersection` problems since these problems emerge in our approach for solving the (ϵ, δ)`-#MUS` problem. Finally, we do not focus on solving the (ϵ, δ)`-#SAT` problem, however the problem is closely related to the (ϵ, δ)`-#MUS` problem.

3 Related Work

It is well-known (see e.g., [21,36,51]) that a clause $f \in F$ belongs to $\texttt{IMU}_F$ iff f is necessary for F. Therefore, to compute $\texttt{IMU}_F$, one can simply check each $f \in F$ for being necessary for F. We are not aware of any work that has focused on the `MUS-intersection` problem in more detail.

The MUS-union problem was recently investigated by Mencia et al. [42]. Their algorithm is based on gradually refining an *under*-approximation of $\texttt{UMU}_F$ until the exact $\texttt{UMU}_F$ is computed. Unfortunately, the authors experimentally show that their algorithm often fails to find the exact $\texttt{UMU}_F$ within a reasonable time even for relatively small input instances (only an under-approximation is computed). In our work, we propose an approach that works in the other way: we start with an *over-approximation* of $\texttt{UMU}_F$ and gradually refine the approximation to eventually get $\texttt{UMU}_F$. Another related research was conducted by Janota and Marques-Silva [30] who proposed several QBF encodings for solving the MUS-membership problem. Although they did not focus on finding $\texttt{UMU}_F$, one can clearly identify $\texttt{UMU}_F$ by solving the MUS-membership problem for each $f \in F$.

As for counting the number of MUSes of F, we are not aware of any previous work dedicated to this problem. Yet, there have been proposed plenty of algorithms and tools (e.g., [3,9,11,12,35,47]) for enumerating/identifying all MUSes of F. Clearly, if we enumerate all MUSes of F, then we obtain the exact value of $|\texttt{AMU}_F|$, and thus we also solve the (ϵ, δ)-#MUS problem. However, since there can be up to exponentially many of MUSes w.r.t. $|F|$, MUS enumeration algorithms are often not able to complete the enumeration in a reasonable time and thus are not able to find the value of $|\texttt{AMU}_F|$.

Very similar to the (ϵ, δ)-#MUS problem is the (ϵ, δ)-#SAT problem. Both problems involve the same probabilistic and approximation guarantees. Moreover, both problems are defined over a power-set structure. In MUS counting, the goal is to count MUSes in $\mathcal{P}(F)$, whereas in model counting, the goal is to count models in $\mathcal{P}(\mathit{Vars}(F))$. In this paper, we propose an algorithm for solving the (ϵ, δ)-#MUS problem that is based on ApproxMC3 [15,17,52]. In particular, we keep the high-level idea of ApproxMC3 for processing/exploring the power-set structure, and we propose new low-level techniques that are specific for MUS counting.

4 AMUSIC: A Hashing-Based MUS Counter

We now describe AMUSIC, a hashing-based algorithm designed to solve the (ε, δ)-#MUS problem. The name of the algorithm is an acronym for Approximate Minimal Unsatisfiable Subsets Implicit Counter. AMUSIC is based on ApproxMC3, which is a hashing-based algorithm to solve (ε, δ)-#SAT problem. As such, while the high-level structure of AMUSIC and ApproxMC3 share close similarities, the two algorithms differ significantly in the design of core technical subroutines.

We first discuss the high-level structure of AMUSIC in Sect. 4.1. We then present the key technical contributions of this paper: the design of core subroutines of AMUSIC in Sects. 4.3, 4.4 and 4.5.

4.1 Algorithmic Overview

The main procedure of AMUSIC is presented in Algorithm 1. The algorithm takes as an input a Boolean formula F in CNF, a tolerance $\epsilon\,(> 0)$, and a confidence

Algorithm 1: AMUSIC(F, ϵ, δ)

```
1  threshold ← 1 + 9.84(1 + ε/(1+ε))(1 + 1/ε)^2
2  Y ← FindMUSes(F, threshold)
3  if |Y| < threshold then return |Y|
4  G ← getUMU(F)
5  I_G ← getIMU(G)
6  nCells ← 2; C ← emptyList; iter ← 0
7  while iter < ⌈17 log2(3/δ)⌉ do
8      iter ← iter + 1
9      (nCells, nSols) ← AMUSICCore(G, I_G, threshold, nCells)
10     if nCells ≠ null then AddToList(C, nCells × nSols)
11 return FindMedian(C)
```

parameter $\delta \in (0, 1]$, and returns an estimate of $|\mathtt{AMU}_F|$ within tolerance ϵ and with confidence at least $1 - \delta$. Similar to ApproxMC3, we first check whether $|\mathtt{AMU}_F|$ is smaller than a specific threshold that is a function of ε. This check is carried out via a MUS enumeration algorithm, denoted FindMUSes, that returns a set Y of MUSes of F such that $|Y| = \mathtt{min}(\mathsf{threshold}, |\mathtt{AMU}_F|)$. If $|Y| < \mathsf{threshold}$, the algorithm terminates while identifying the exact value of $|\mathtt{AMU}_F|$. In a significant departure from ApproxMC3, AMUSIC subsequently computes the union ($\mathtt{UMU}_F$) and the intersection ($\mathtt{IMU}_F$) of all MUSes of F by invoking the subroutines GetUMU and GetIMU, respectively. Through the lens of set representation of the CNF formulas, we can view $\mathtt{UMU}_F$ as another CNF formula, G. Our key observation is that $\mathtt{AMU}_F = \mathtt{AMU}_G$ (see Sect. 4.2), thus instead of working with the whole F, we can focus only on G. The rest of the main procedure is similar to ApproxMC3, i.e., we repeatedly invoke the core subroutine called AMUSICCore. The subroutine attempts to find an estimate c of $|\mathtt{AMU}_G|$ within the tolerance ϵ. Briefly, to find the estimate, the subroutine partitions $\mathcal{P}(G)$ into nCells cells, then picks one of the cells, and counts the number nSols of MUSes in the cell. The pair (nCells, nSols) is returned by AMUSICCore, and the estimate c of $|\mathtt{AMU}_G|$ is then computed as nSols $\times$ nCells. There is a small chance that AMUSICCore fails to find the estimate; it such a case nCells $=$ nSols $=$ null. Individual estimates are stored in a list C. After the final invocation of AMUSICCore, AMUSIC computes the median of the list C and returns the median as the final estimate of $|\mathtt{AMU}_G|$. The total number of invocations of AMUSICCore is in $\mathcal{O}(\log(1/\delta))$ which is enough to ensure the required confidence $1 - \delta$ (details on assurance of the (ϵ, δ) guarantees are provided in Sect. 4.2).

We now turn to AMUSICCore which is described in Algorithm 2. The partition of $\mathcal{P}(G)$ into nCells cells is made via a hash function h from $H_{xor}(|G|, m)$, i.e. nCells $= 2^m$. The choice of m is a crucial part of the algorithm as it regulates the size of the cells. Intuitively, it is easier to identify all MUSes of a small cell; however, on the contrary, the use of small cells does not allow to achieve a reasonable tolerance. Based on ApproxMC3, we choose m such that a cell given by a hash function $h \in H_{xor}(|G|, m)$ contains almost threshold many MUSes. In

Algorithm 2: AMUSICCore($G, \mathtt{I}_G$, threshold, *prevNCells*)

```
1 Choose h at random from H_xor(|G|, |G| − 1)
2 Choose α at random from {0,1}^{|G|−1}
3 nSols ← CountInCell(G, I_G, h, α, threshold)
4 if nSols = threshold then return (null, null)
5 mPrev ← log_2 prevNCells
6 (nCells, nSols) ← LogMUSSearch(G, I_G, h, α, threshold, mPrev)
7 return (nCells, nSols)
```

particular, the computation of AMUSICCore starts by choosing at random a hash function h from $H_{xor}(|G|, |G|-1)$ and a cell α at random from $\{0,1\}^{|G|-1}$. Subsequently, the algorithm tends to identify m^{th} prefixes $h^{(m)}$ and $\alpha^{(m)}$ of h and α, respectively, such that $|\mathtt{AMU}_{\langle G,h^{(m)},\alpha^{(m)}\rangle}| < \mathsf{threshold}$ and $|\mathtt{AMU}_{\langle G,h^{(m-1)},\alpha^{(m-1)}\rangle}| \geq \mathsf{threshold}$. Recall that $\mathtt{AMU}_{\langle G,h^{(1)},\alpha^{(1)}\rangle} \supseteq \cdots \supseteq \mathtt{AMU}_{\langle G,h^{(|G|-1)},\alpha^{(|G|-1)}\rangle}$ (Observation 1, Sect. 2). We also know that the cell $\alpha^{(0)}$, i.e. the whole $\mathcal{P}(G)$, contains at least threshold MUSes (see Algorithm 1, line 3). Consequently, there can exist at most one such m, and it exists if and only if $|\mathtt{AMU}_{\langle G,h^{(|G|-1)},\alpha^{(|G|-1)}\rangle}| < \mathsf{threshold}$. Therefore, the algorithm first checks whether $|\mathtt{AMU}_{\langle G,h^{(|G|-1)},\alpha^{(|G|-1)}\rangle}| < \mathsf{threshold}$. The check is carried via a procedure CountInCell that returns the number $\mathsf{nSols} = \mathtt{min}(|\mathtt{AMU}_{\langle G,h^{(|G|-1)},\alpha^{(|G|-1)}\rangle}|, \mathsf{threshold})$. If $\mathsf{nSols} = \mathsf{threshold}$, then AMUSICCore fails to find the estimate of $|\mathtt{AMU}_G|$ and terminates. Otherwise, a procedure LogMUSSearch is used to find the required value of m together with the number nSols of MUSes in $\alpha^{(m)}$. The implementation of LogMUSSearch is directly adopted from ApproxMC3 and thus we do not provide its pseudocode here (note that in ApproxMC3 the procedure is called LogSATSearch). We only briefly summarize two main ingredients of the procedure. First, it has been observed that the required value of m is often similar for repeated calls of AMUSICCore. Therefore, the algorithm keeps the value *mPrev* of m from previous iteration and first test values near *mPrev*. If none of the near values is the required one, the algorithm exploits that $\mathtt{AMU}_{\langle G,h^{(1)},\alpha^{(1)}\rangle} \supseteq \cdots \supseteq \mathtt{AMU}_{\langle G,h^{(|G|-1)},\alpha^{(|G|-1)}\rangle}$, which allows it to find the required value of m via the galloping search (variation of binary search) while performing only $\log|G|$ calls of CountInCell.

Note that in ApproxMC3, the procedure CountInCell is called BSAT and it is implemented via an NP oracle, whereas we use a Σ_3^P oracle to implement the procedure (see Sect. 4.3). The high-level functionality is the same: the procedures use up to threshold calls of the oracle to check whether the number of the target elements (models vs. MUSes) in a cell is lower than threshold.

4.2 Analysis and Comparison with ApproxMC3

Following from the discussion above, there are three crucial technical differences between AMUSIC and ApproxMC3: (1) the implementation of the subroutine CountInCell in the context of MUS, (2) computation of the intersection $\mathtt{IMU}_F$ of all MUSes of F and its usage in CountInCell, and (3) computation of the union

$\texttt{UMU}_F$ of all MUSes of F and invocation of the underlying subroutines with G (i.e., $\texttt{UMU}_F$) instead of F. The usage of CountInCell can be viewed as domain-specific instantiation of BSAT in the context of MUSes. Furthermore, we use the computed intersection of MUSes to improve the runtime efficiency of CountInCell. It is perhaps worth mentioning that prior studies have observed that over 99% of the runtime of ApproxMC3 is spent inside the subroutine BSAT [52]. Therefore, the runtime efficiency of CountInCell is crucial for the runtime performance of AMUSIC, and we discuss in detail, in Sect. 4.3, algorithmic contributions in the context of CountInCell including usage of $\texttt{IMU}_F$. We now argue that the replacement of F with G in line 4 in Algorithm 1 does not affect correctness guarantees, which is stated formally below:

Lemma 1. *For every G' such that $\texttt{UMU}_F \subseteq G' \subseteq F$, the following hold:*

$$\texttt{AMU}_F = \texttt{AMU}_{G'} \tag{1}$$

$$\texttt{IMU}_F = \texttt{IMU}_{G'} \tag{2}$$

Proof. (1) Since $G' \subseteq F$ then every MUS of G' is also a MUS of F. In the other direction, every MUS of F is contained in the union $\texttt{UMU}_F$ of all MUSes of F, and thus every MUS of F is also a MUS of G' ($\supseteq \texttt{UMU}_F$).

(2) $\texttt{IMU}_F = \bigcap_{M \in \texttt{AMU}_F} = \bigcap_{M \in \texttt{AMU}_{G'}} = \texttt{IMU}_{G'}$.

Equipped with Lemma 1, we now argue that each run of AMUSIC can be simulated by a run of ApproxMC3 for an appropriately chosen formula. Given an unsatisfiable formula $F = \{f_1, \ldots, f_{|F|}\}$, let us by B_F denote a satisfiable formula such that: (1) $Vars(B_F) = \{x_1, \ldots, x_{|F|}\}$ and (2) an assignment $I : Vars(B_F) \to \{1, 0\}$ is a model of B_F iff $\{f_i | I(x_i) = 1\}$ is a MUS of F. Informally, models of B_F one-to-one map to MUSes of F. Hence, the size of sets returned by CountInCell for F is identical to the corresponding BSAT for B_F. Since the analysis of ApproxMC3 only depends on the correctness of the size of the set returned by BSAT, we conclude that the answer computed by AMUSIC would satisfy (ε, δ) guarantees. Furthermore, observing that CountInCell makes threshold many queries to Σ_3^P-oracle, we can bound the time complexity. Formally,

Theorem 1. *Given a formula F, a tolerance $\varepsilon > 0$, and a confidence $1 - \delta \in (0, 1]$, let* AMUSIC(F, ε, δ) *return c. Then $Pr[|\texttt{AMU}_F|/(1+\epsilon) \leq c \leq |\texttt{AMU}_F| \cdot (1+\epsilon)] \geq 1 - \delta$. Furthermore,* AMUSIC *makes $\mathcal{O}(\log |F| \cdot \frac{1}{\varepsilon^2} \cdot \log(1/\delta))$ calls to Σ_3^P oracle.*

Few words are in order concerning the complexity of AMUSIC. As noted in Sect. 1, for a formula on n variables, approximate model counters make $\mathcal{O}(\log n \cdot \frac{1}{\varepsilon^2} \cdot \log(1/\delta))$ calls to an NP oracle, whereas the complexity of finding a satisfying assignment is NP-complete. In our case, we make calls to a Σ_3^P oracle while the problem of finding a MUS is in FP^{NP}. Therefore, a natural direction of future work is to investigate the design of a hashing-based technique that employs an FP^{NP} oracle.

Algorithm 3: CountInCell$(G, \mathtt{I}_G, h, \alpha, \mathsf{threshold})$

```
1 c ← 0; M ← {}
2 while c < threshold do
3     M ← GetMUS(G, I_G, M, h, α)
4     if M = null then return c
5     M ← M ∪ {M}
6     c ← c + 1
7 return c
```

4.3 Counting MUSes in a Cell: CountInCell

In this section, we describe the procedure `CountInCell`. The input of the procedure is the formula G (i.e., $\mathtt{UMU}_F$), the set $\mathtt{I}_G = \mathtt{IMU}_G$, a hash function $h \in H_{xor}(|G|, m)$, a cell $\alpha \in \{0,1\}^m$, and the threshold value. The output is $c = min(\mathsf{threshold}, |\mathtt{AMU}_{\langle G,h,\alpha\rangle}|)$.

The description is provided in Algorithm 3. The algorithm iteratively calls a procedure `GetMUS` that returns either a MUS M such that $M \in (\mathtt{AMU}_{\langle G,h,\alpha\rangle} \setminus \mathcal{M})$ or `null` if there is no such MUS. For each M, the value of c is increased and M is added to $\mathcal{M}$. The loop terminates either when c reaches the value of threshold or when `GetMUS` fails to find a new MUS (i.e., returns `null`). Finally, the algorithm returns c.

GetMUS. To implement the procedure `GetMUS`, we build an $\exists\forall\exists$-QBF formula `MUSInCell` such that each witness of the formula corresponds to a MUS from $\mathtt{AMU}_{\langle G,h,\alpha\rangle} \setminus \mathcal{M}$. The formula consists of several parts and uses several sets of variables that are described in the following.

The main part of the formula, shown in Eq. (3), introduces the first existential quantifier and a set $P = \{p_1, \ldots, p_{|G|}\}$ of variables that are quantified by the quantifier. Note that each valuation I of P corresponds to a subset S of G; in particular let us by $I_{P,G}$ denote the set $\{f_i \in G \mid I(p_i) = 1\}$. The formula is build in such a way that a valuation I is a witness of the formula if and only if $I_{P,G}$ is a MUS from $\mathtt{AMU}_{\langle G,h,\alpha\rangle} \setminus \mathcal{M}$. This property is expressed via three conjuncts, denoted $\mathtt{inCell}(P)$, $\mathtt{unexplored}(P)$, and $\mathtt{isMUS}(P)$, encoding that (i) $I_{P,G}$ is in the cell α, (ii) $I_{P,G}$ is not in $\mathcal{M}$, and (iii) $I_{P,G}$ is a MUS, respectively.

$$\mathtt{MUSInCell} = \exists P.\, \mathtt{inCell}(P) \wedge \mathtt{unexplored}(P) \wedge \mathtt{isMUS}(P) \tag{3}$$

Recall that the family $H_{xor}(n,m)$ of hash functions is defined as $\{h \mid h(y)[i] = a_{i,0} \oplus (\bigoplus_{k=1}^{n} a_{i,k} \wedge y[k])\ \textit{for all}\ 1 \le i \le m\}$, where $a_{i,k} \in \{0,1\}$ (Sect. 2). A hash function $h \in H_{xor}(n,m)$ is given by fixing the values of individual $a_{i,k}$ and a cell α of h is a bit-vector from $\{0,1\}^m$. The formula $\mathtt{inCell}(P)$ encoding that the set $I_{P,G}$ is in the cell α of h is shown in Eq. (4).

$$\mathtt{inCell}(P) = \bigwedge_{i=1}^{m} (a_{i,0} \oplus (\bigoplus_{p \in \{p_k \mid a_{i,k}=1\}} p) \oplus \neg\alpha[i]) \tag{4}$$

To encode that we are not interested in MUSes from $\mathcal{M}$, we can simply block all the valuations of P that correspond to these MUSes. However, we can do better. In particular, recall that if M is a MUS, then no proper subset and no proper superset of M can be a MUS; thus, we prune away all these sets from the search space. The corresponding formula is shown in Eq. (5).

$$\texttt{unexplored}(P) = \bigwedge_{M \in \mathcal{M}} ((\bigvee_{f_i \in M} \neg p_i) \wedge (\bigvee_{f_i \notin M} p_i)) \tag{5}$$

The formula $\texttt{isMUS}(P)$ encoding that $I_{P,G}$ is a MUS is shown in Eq. (6). Recall that $I_{P,G}$ is a MUS if and only if $I_{P,G}$ is unsatisfiable and for every *closest subset* S of $I_{P,G}$ it holds that S is satisfiable, where *closest subset* means that $|I_{P,G} \setminus S| = 1$. We encode these two conditions using two subformulas denoted by $\texttt{unsat}(P)$ and $\texttt{noUnsatSubset}(P)$.

$$\texttt{isMUS}(P) = \texttt{unsat}(P) \wedge \texttt{noUnsatSubset}(P) \tag{6}$$

The formula $\texttt{unsat}(P)$, shown in Eq. (7), introduces the set $\mathit{Vars}(G)$ of variables that appear in G and states that every valuation of $\mathit{Vars}(G)$ falsifies at least one clause contained in $I_{P,G}$.

$$\texttt{unsat}(P) = \forall \mathit{Vars}(G).\ \bigvee_{f_i \in G} (p_i \wedge \neg f_i) \tag{7}$$

The formula $\texttt{noUnsatSubset}(P)$, shown in Eq. (8), introduces another set of variables: $Q = \{q_1, \ldots, q_{|G|}\}$. Similarly as in the case of P, each valuation I of Q corresponds to a subset of G defined as $I_{Q,G} = \{f_i \in G \mid I(q_i) = 1\}$. The formula expresses that for every valuation I of Q it holds that $I_{Q,G}$ is satisfiable or $I_{Q,G}$ is not a closest subset of $I_{P,G}$.

$$\texttt{noUnsatSubset}(P) = \forall Q.\, \texttt{sat}(Q) \vee \neg \texttt{subset}(Q, P) \tag{8}$$

The requirement that $I_{Q,G}$ is satisfiable is encoded in Eq. (9). Since we are already reasoning about the satisfiability of G's clauses in Eq. (7), we introduce here a copy G' of G where each variable x_i of G is substituted by its primed copy x'_i. Equation (9) states that there exists a valuation of $\mathit{Vars}(G')$ that satisfies $I_{Q,G}$.

$$\texttt{sat}(Q) = \exists \mathit{Vars}(G').\ \bigwedge_{f_i \in G'} (\neg q_i \vee f_i) \tag{9}$$

Equation (10) encodes that $I_{Q,G}$ is a closest subset of $I_{P,G}$. To ensure that $I_{Q,G}$ is a *subset* of $I_{P,G}$, we add the clauses $q_i \rightarrow p_i$. To ensure the *closeness*, we use cardinality constraints. In particular, we introduce another set $R = \{r_1, \ldots, r_{|G|}\}$ of variables and enforce their values via $r_i \leftrightarrow (p_i \wedge \neg q_i)$. Intuitively, the number of variables from R that are set to 1 equals to $|I_{P,G} \setminus I_{Q,G}|$. Finally, we add cardinality constraints, denoted by $\texttt{exactlyOne}(R)$, ensuring that exactly one r_i is set to 1.

$$\texttt{subset}(Q,P) = \exists R.\ \bigwedge_{p_i \in P} ((q_i \rightarrow p_i) \wedge (r_i \leftrightarrow (p_i \wedge \neg q_i)) \wedge \texttt{exactlyOne}(R) \quad (10)$$

Note that instead of encoding a *closest subset* in Eq. 10, we could just encode that $I_{Q,G}$ is an arbitrary proper subset of $I_{P,G}$ as it would still preserve the meaning of Eq. 6 that $I_{P,G}$ is a MUS. Such an encoding would not require introducing the set R of variables and also, at the first glance, would save a use of one existential quantifier. The thing is that the whole formula would still be in the form of $\exists\forall\exists$-QBF due to Eq. 9 (which introduces the second existential quantifier). The advantage of using a closet subset is that we significantly prune the search space of the QBF solver. It is thus matter of contemporary QBF solvers whether it is more beneficial to reduce the number of variables (by removing R) or to prune the searchspace via R.

For the sake of lucidity, we have not exploited the knowledge of $\texttt{IMU}_G$ (I_G) while presenting the above equations. Since we know that every clause $f \in \texttt{IMU}_G$ has to be contained in every MUS of G, we can fix the values of the variables $\{p_i \mid f_i \in \texttt{IMU}_G\}$ to 1. This, in turn, significantly simplifies the equations and prunes away exponentially many (w.r.t. $|\texttt{IMU}_G|$) valuations of P, Q, and R, that need to be assumed. To solve the final formula, we employ a $\exists\forall\exists$-QBF solver, i.e., a Σ_3^P oracle.

Finally, one might wonder why we use our custom solution for identifying MUSes in a cell instead of employing one of existing MUS extraction techniques. Conventional MUS extraction algorithms cannot be used to identify MUSes that are in a cell since the cell is not "continuous" w.r.t. the set containment. In particular, assume that we have three sets of clauses, K, L, M, such that $K \subset L \subset M$. It can be the case that K and M are in the cell, but L is not in the cell. Contemporary MUS extraction techniques require the search space to be continuous w.r.t. the set containment and thus cannot be used in our case.

4.4 Computing $\texttt{UMU}_F$

We now turn our attention to computing the union $\texttt{UMU}_F$ (i.e., G) of all MUSes of F. Let us start by describing well-known concepts of *autark variables* and a *lean kernel*. A set $A \subseteq \mathit{Vars}(F)$ of variables is an *autark* of F iff there exists a truth assignment to A such that every clause of F that contains a variable from A is satisfied by the assignment [44]. It holds that the union of two autark sets is also an autark set, thus there exists a unique largest autark set (see, e.g., [31,32]). The *lean kernel* of F is the set of all clauses that do not contain any variable from the largest autark set. It is known that the *lean kernel* of F is an over-approximation of $\texttt{UMU}_F$ (see e.g., [31,32]), and there were proposed several algorithms, e.g., [33,38], for computing the lean kernel.

Algorithm. Our approach for computing $\texttt{UMU}_F$ consists of two parts. First, we compute the lean kernel K of F to get an over-approximation of $\texttt{UMU}_F$, and

Algorithm 4: `getUMU(`F`)`

```
1 K ← the lean kernel of F; M ← {}
2 for f ∈ K \ {f ∈ M | M ∈ M} do
3 |   W ← checkNecessity(f, K)
4 |   if W ≠ null then M ← M ∪ { a MUS of W}
5 |   else K ← K \ {f}
6 return K
```

then we gradually refine the over-approximation K until K is exactly the set $\texttt{UMU}_F$. The refinement is done by solving the MUS-membership problem for each $f \in K$. To solve the MUS-membership problem efficiently, we reveal a connection to necessary clauses, as stated in the following lemma.

Lemma 2. *A clause $f \in F$ belongs to $\texttt{UMU}_F$ iff there is a subset W of F such that W is unsatisfiable and f is necessary for W (i.e., $W \setminus \{f\}$ is satisfiable).*

Proof. $\Rightarrow$: Let $f \in \texttt{UMU}_F$ and $M \in \texttt{AMU}_F$ such that $f \in M$. Since M is a MUS then $M \setminus \{f\}$ is satisfiable; thus f is necessary for M.
$\Leftarrow$: If W is a subset of F and $f \in W$ a necessary clause for W then f has to be contained in every MUS of W. Moreover, W has at least one MUS and since $W \subseteq F$, then every MUS of W is also a MUS of F.

Our approach for computing $\texttt{UMU}_F$ is shown in Algorithm 4. It takes as an input the formula F and outputs $\texttt{UMU}_F$ (denoted K). Moreover, the algorithm maintains a set $\mathcal{M}$ of MUSes of F. Initially, $\mathcal{M} = \emptyset$ and K is set to the lean kernel of F; we use an approach by Marques-Silva et al. [38] to compute the lean kernel. At this point, we know that $K \supseteq \texttt{UMU}_F \supseteq \{f \in M \mid M \in \mathcal{M}\}$. To find $\texttt{UMU}_F$, the algorithm iteratively determines for each $f \in K \setminus \{f \in M \mid M \in \mathcal{M}\}$ if $f \in \texttt{UMU}_F$. In particular, for each f, the algorithm checks whether there exists a subset W of K such that f is necessary for W (Lemma 2). The task of finding W is carried out by a procedure `checkNecessity(`f, K`)`. If there is no such W, then the algorithm removes f from K. In the other case, if W exists, the algorithm finds a MUS of W and adds the MUS to the set $\mathcal{M}$. Any available single MUS extraction approach, e.g., [2,5,7,46], can be used to find the MUS.

To implement the procedure `checkNecessity(`f, K`)` we build a QBF formula that is true iff there exists a set $W \subseteq K$ such that W is unsatisfiable and f is necessary for W. To represent W we introduce a set $S = \{s_g \mid g \in K\}$ of Boolean variables; each valuation I of S corresponds to a subset $I_{S,K}$ of K defined as $I_{S,K} = \{g \in K \mid I(s_g) = 1\}$. Our encoding is shown in Eq. 11.

$$\exists S, Vars(K). \forall Vars(K'). s_f \wedge \Big(\bigwedge_{g \in K \setminus \{f\}} (g \vee \neg s_g) \Big) \wedge \Big(\bigvee_{g \in K'} (\neg g \wedge s_g) \Big) \tag{11}$$

The formula consists of three main conjuncts. The first conjunct ensures that f is present in $I_{S,K}$. The second conjunct states that $I_{S,K} \setminus \{f\}$ is satisfiable,

i.e., that there exists a valuation of $Vars(K)$ that satisfies $I_{S,K} \setminus \{f\}$. Finally, the last conjunct express that $I_{S,K}$ is unsatisfiable, i.e., that every valuation of $Vars(K)$ falsifies at least one clause of $I_{S,K}$. Since we are already reasoning about variables of K in the second conjunct, in the third conjunct, we use a primed version (a copy) K' of K.

Alternative QBF Encodings. Janota and Marques-Silva [30] proposed three other QBF encodings for the MUS-membership problem, i.e., for deciding whether a given $f \in F$ belongs to $\texttt{UMU}_F$. Two of the three proposed encodings are typically inefficient; thus, we focus on the third encoding, which is the most concise among the three. The encoding, referred to as JM encoding (after the initials of the authors), uses only two quantifiers in the form of $\exists\forall$-QBF and it is only linear in size w.r.t. $|F|$. The underlying ideas by JM encoding and our encoding differ significantly. Our encoding is based on necessary clauses (Lemma 2), whereas JM exploits a connection to so-called *Maximal Satisfiable Subsets*. Both the encodings use the same quantifiers; however, our encoding is smaller. In particular, the JM uses $2 \times (Vars(F) + |F|)$ variables whereas our encoding uses only $|F| + 2 \times Vars(F)$ variables, and leads to smaller formulas.

Implementation. Recall that we compute $\texttt{UMU}_F$ to reduce the search space, i.e. instead of working with the whole F, we work only with $G = \texttt{UMU}_F$. The soundness of this reduction is witnessed in Lemma 1 (Sect. 4.2). In fact, Lemma 1 shows that it is sound to reduce the search space to any G' such that $\texttt{UMU}_F \subseteq G' \subseteq F$. Since our algorithm for computing $\texttt{UMU}_F$ subsumes repeatedly solving a Σ_2^P-complete problem, it can be very time-consuming. Therefore, instead of computing the exact $\texttt{UMU}_F$, we optionally compute only an over-approximation G' of $\texttt{UMU}_F$. In particular, we set a (user-defined) time limit for computing the lean kernel K of F. Moreover, we use a time limit for executing the procedure $\texttt{checkNecessity}(f, K)$; if the time limit is exceeded for a clause $f \in K$, we conservatively assume that $f \in \texttt{UMU}_F$, i.e., we over-approximate.

Sparse Hashing and $\texttt{UMU}_F$. The approach of computation of $\texttt{UMU}_F$ is similar to, in spirit, computation of independent support of a formula to design sparse hash functions [16,28]. Briefly, given a Boolean formula H, an *independent support* of H is a set $\mathcal{I} \subseteq Vars(H)$ such that in every model of H, the truth assignment to $\mathcal{I}$ uniquely determines the truth assignment to $Vars(H) \setminus \mathcal{I}$. Practically, independent support can be used to reduce the search space where a model counting algorithm searches for models of H. It is interesting to note that the state of the art technique reduces the computation of independent support of a formula in the context of model counting to that of computing (Group) Minimal Unsatisfiable Subset (GMUS). Thus, a formal study of computation of independent support in the context of MUSes is an interesting direction of future work.

Algorithm 5: `getIMU`(G)

```
1  C ← G
2  K ← ∅
3  while C ≠ ∅ do
4      f ← choose f ∈ C
5      (sat?, I, core) ← checkSAT(G \ {f})
6      if sat? then
7          R ← RMR(G, f, I)
8          K ← K ∪ {f} ∪ R
9          C ← C \ ({f} ∪ R)
10     else
11         C ← C ∩ core
12 return K
```

4.5 Computing $\texttt{IMU}_G$

Our approach to compute the intersection $\texttt{IMU}_G$ (i.e., I_G) of all MUSes of G is composed of several ingredients. First, recall that a clause $f \in G$ belongs to $\texttt{IMU}_G$ iff f is necessary for G. Another ingredient is the ability of contemporary SAT solvers to provide either a model or an *unsat core* of a given unsatisfiable formula $N \subseteq G$, i.e., a small, yet not necessarily minimal, unsatisfiable subset of N. The final ingredient is a technique called *model rotation*. The technique was originally proposed by Marques-Silva and Lynce [40], and it serves to explore necessary clauses based on other already known necessary clauses. In particular, let f be a necessary clause for G and $I : \mathit{Vars}(G) \to \{0, 1\}$ a model of $G \setminus \{f\}$. Since G is unsatisfiable, the model I does not satisfy f. The model rotation attempts to alter I by switching, one by one, the Boolean assignment to the variables $\mathit{Vars}(\{f\})$. Each variable assignment I' that originates from such an alternation of I necessarily satisfies f and does not satisfy at least one $f' \in G$. If it is the case that there is exactly one such f', then f' is necessary for G. An improved version of model rotation, called *recursive model rotation*, was later proposed by Belov and Marques-Silva [6] who noted that the model rotation could be recursively performed on the newly identified necessary clauses.

Our approach for computing $\texttt{IMU}_G$ is shown in Algorithm 5. To find $\texttt{IMU}_G$, the algorithm decides for each f whether f is necessary for G. In particular, the algorithm maintains two sets: a set C of *candidates* on necessary clauses and a set K of already known necessary clauses. Initially, K is empty and $C = G$. At the end of computation, C is empty and K equals to $\texttt{IMU}_G$. The algorithm works iteratively. In each iteration, the algorithm picks a clause $f \in C$ and checks $G \setminus \{f\}$ for satisfiability via a procedure `checkSAT`. Moreover, `checkSAT` returns either a model I or an unsat core *core* of $G \setminus \{f\}$. If $G \setminus \{f\}$ is satisfiable, i.e. f is necessary for G, the algorithm employs the recursive model rotation, denoted by $\texttt{RMR}(G, f, I)$, to identify a set R of additional necessary clauses. Subsequently, all the newly identified necessary clauses are added to K and removed from C.

In the other case, when $G \setminus \{f\}$ is unsatisfiable, the set C is reduced to $C \cap core$ since every necessary clause of G has to be contained in every unsatisfiable subset of G. Note that $f \notin core$, thus at least one clause is removed from C.

5 Experimental Evaluation

We employed several external tools to implement AMUSIC. In particular, we use the QBF solver CAQE [49] for solving the QBF formula `MUSInCell`, the 2QBF solver CADET [50] for solving our $\exists\forall$-QBF encoding while computing $\texttt{UMU}_F$, and the QBF preprocessor QRATPre+ [37] for preprocessing/simplifying our QBF encodings. Moreover, we employ muser2 [7] for a single MUS extraction while computing $\texttt{UMU}_F$, a MaxSAT solver UWrMaxSat [48] to implement the algorithm by Marques-Silva et al. [38] for computing the lean kernel of F, and finally, we use a toolkit called pysat [27] for encoding cardinality constraints used in the formula `MUSInCell`. The tool along with all benchmarks that we used is available at https://github.com/jar-ben/amusic.

Objectives. As noted earlier, AMUSIC is the first technique to (approximately) count MUSes without explicit enumeration. We demonstrate the efficacy of our approach via a comparison with two state of the art techniques for MUS enumeration: MARCO [35] and MCSMUS [3]. Within a given time limit, a MUS enumeration algorithm either identifies the whole $\texttt{AMU}_F$, i.e., provides the exact value of $|\texttt{AMU}_F|$, or identifies just a subset of $\texttt{AMU}_F$, i.e., provides an under-approximation of $|\texttt{AMU}_F|$ with no approximation guarantees.

The objective of our empirical evaluation was two-fold: First, we experimentally examine the scalability of AMUSIC, MARCO, and MCSMUS w.r.t. $|\texttt{AMU}_F|$. Second, we examine the *empirical accuracy* of AMUSIC.

Benchmarks and Experimental Setup. Given the lack of dedicated counting techniques, there is no sufficiently large set of publicly available benchmarks to perform critical analysis of counting techniques. To this end, we focused on a recently emerging theme of evaluation of SAT-related techniques on *scalable benchmarks*[1]. In keeping with prior studies employing empirical methodology based on scalable benchmarks [22,41], we generated a custom collection of CNF benchmarks. The benchmarks mimic requirements on multiprocessing systems. Assume that we are given a system with two groups (kinds) of processes, $A = \{a_1, \ldots, a_{|A|}\}$ and $B = \{b_1, \ldots, b_{|B|}\}$, such that $|A| \geq |B|$. The processes require resources of the system; however, the resources are limited. Therefore, there are restrictions on which processes can be active simultaneously. In particular, we have the following three types of mutually independent restrictions on the system:

[1] M. Y. Vardi, in his talk at BIRS CMO 18w5208 workshop, called on the SAT community to focus on scalable benchmarks in lieu of competition benchmarks. Also, see: https://gitlab.com/satisfiability/scalablesat (Accessed: May 10, 2020).

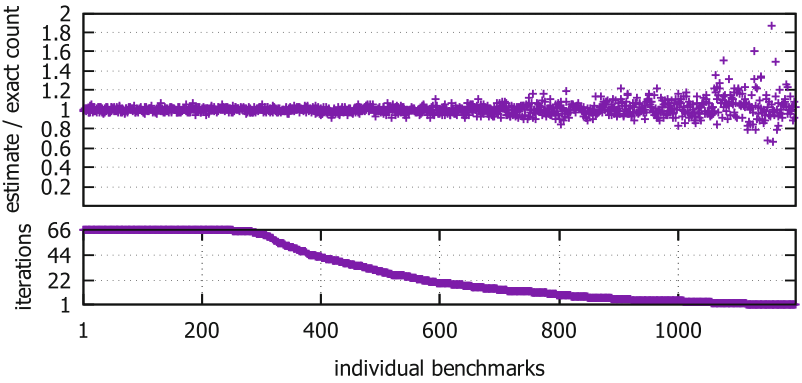


Fig. 3. The number of completed iterations and the accuracy of the final MUS count estimate for individual benchmarks.

- The first type of restriction states that "at most $k-1$ processes from the group A can be active simultaneously", where $k \leq |A|$.
- The second type of restriction enforces that "if no process from B is active then at most $k-1$ processes from A can be active, and if at least one process from B is active then at most $l-1$ processes from A can be active", where $k, l \leq |A|$.
- The third type of restriction includes the second restriction. Moreover, we assume that a process from B can activate a process from A. In particular, for every $b_i \in B$, we assume that when b_i is active, then a_i is also active.

We encode the three restrictions via three Boolean CNF formulas, R_1, R_2, R_3. The formulas use three sets of variables: $X = \{x_1, \ldots, x_{|A|}\}$, $Y = \{y_1, \ldots, y_{|B|}\}$, and Z. The sets X and Y represent the Boolean information about activity of processes from A and B: a_i is active iff $x_i = 1$ and b_j is active iff $y_j = 1$. The set Z contains additional auxiliary variables. Moreover, we introduce a formula $\texttt{ACT} = (\bigwedge_{x_i \in X} x_i) \wedge (\bigwedge_{y_i \in Y} y_i)$ encoding that all processes are active. For each $i \in \{1, 2, 3\}$, the conjunction $G_i = R_i \wedge \texttt{ACT}$ is unsatisfiable. Intuitively, every MUS of G_i represents a minimal subset of processes that need to be active to violate the restriction. The number of MUSes in G_1, G_2, and G_3 is $\binom{|A|}{k}$, $\binom{|A|}{k} + |B| \times \binom{|A|}{l}$, and $\binom{|A|}{k} + \sum_{i=1}^{|B|} (\binom{|B|}{i} \times \binom{|A|-1}{l-i})$, respectively. We generated G_1, G_2, and G_3 for these values: $10 \leq |A| \leq 30$, $2 \leq |B| \leq 6$, $\lfloor \frac{|A|}{2} \rfloor \leq k \leq \lfloor \frac{3 \times |A|}{2} \rfloor$, and $l = k - 1$. In total, we obtained 1353 benchmarks (formulas) that range in their size from 78 to 361 clauses, use from 40 to 152 variables, and contain from 120 to 1.7×10^9 MUSes.

All experiments were run using a time limit of 7200 s and computed on an AMD EPYC 7371 16-Core Processor, 1 TB memory machine running Debian Linux 4.19.67-2. The values of ϵ and δ were set to 0.8 and 0.2, respectively.

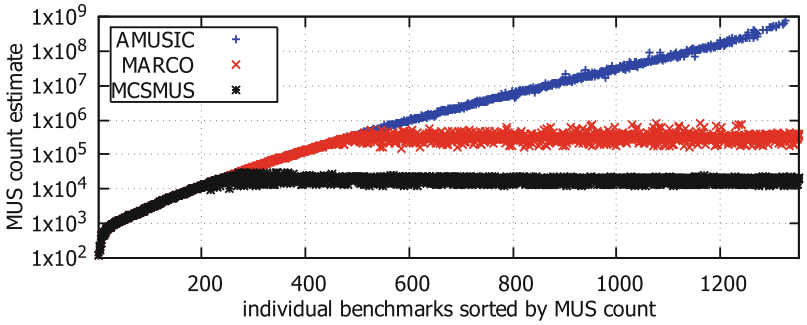


Fig. 4. Scalability of AMUSIC, MARCO, and MCSMUS w.r.t. $|\mathtt{AMU}_F|$.

Accuracy. Recall that to compute an estimate c of $|\mathtt{AMU}_F|$, AMUSIC performs multiple iteration of executing AMUSICCore to get a list C of multiple estimates of $|\mathtt{AMU}_F|$, and then use the median of C as the final estimate c. The more iterations are performed, the higher is the confidence that c is within the required tolerance $\epsilon = 0.8$, i.e., that $\frac{|\mathtt{AMU}_F|}{1.8} \leq c \leq 1.8 \cdot |\mathtt{AMU}_F|$. To achieve the confidence $1 - \delta = 0.8$, 66 iterations need to be performed. In case of 157 benchmarks, the algorithm was not able to finish even a single iteration, and only in case of 251 benchmarks, the algorithm finished all the 66 iterations. For the remaining 945 benchmarks, at least some iterations were finished, and thus at least an estimate with a lower confidence was determined.

We illustrate the achieved results in Fig. 3. The figure consists of two plots. The plot at the bottom of the figure shows the number of finished iterations (y-axis) for individual benchmarks (x-axis). The plot at the top of the figure shows how accurate were the MUS count estimates. In particular, for each benchmark (formula) F, we show the number $\frac{c}{|\mathtt{AMU}_F|}$ where c is the final estimate (median of estimates from finished iterations). For benchmarks where all iterations were completed, it was always the case that the final estimate is within the required tolerance, although we had only 0.8 theoretical confidence that it would be the case. Moreover, the achieved estimate never exceeded a tolerance of 0.1, which is much better than the required tolerance of 0.8. As for the benchmarks where only some iterations were completed, there is only a single benchmark where the tolerance of 0.8 was exceeded.

Scalability. The scalability of AMUSIC, MARCO, and MCSMUS w.r.t. the number of MUSes ($|\mathtt{AMU}_F|$) is illustrated in Fig. 4. In particular, for each benchmark (x-axis), we show in the plot the estimate of the MUS count that was achieved by the algorithms (y-axis). The benchmarks are sorted by the exact count of MUSes in the benchmarks. MARCO and MCSMUS were able to finish the MUS enumeration, and thus to provide the count, only for benchmarks that contained at most 10^6 and 10^5 MUSes, respectively. AMUSIC, on the other hand, was able to provide estimates on the MUS count even for benchmarks that contained up to

10^9 MUSes. Moreover, as we have seen in Fig. 3, the estimates are very accurate. Only in the case of 157 benchmarks where AMUSIC finished no iteration, it could not provide any estimate.

6 Summary and Future Work

We presented a probabilistic algorithm, called AMUSIC, for approximate MUS counting that needs to explicitly identify only logarithmically many MUSes and yet still provides strong theoretical guarantees. The high-level idea is adopted from a model counting algorithm ApproxMC3: we partition the search space into small cells, then count MUSes in a single cell, and estimate the total count by scaling the count from the cell. The novelty lies in the low-level algorithmic parts that are specific for MUSes. Mainly, (1) we propose QBF encoding for counting MUSes in a cell, (2) we exploit MUS intersection to speed-up localization of MUSes, and (3) we utilize MUS union to reduce the search space significantly. Our experimental evaluation showed that the scalability of AMUSIC outperforms the scalability of contemporary enumeration-based counters by several orders of magnitude. Moreover, the practical accuracy of AMUSIC is significantly better than what is guaranteed by the theoretical guarantees.

Our work opens up several questions at the intersection of theory and practice. From a theoretical perspective, the natural question is to ask if we can design a scalable algorithm that makes polynomially many calls to an *NP* oracle. From a practical perspective, our work showcases interesting applications of QBF solvers with native XOR support. Since approximate counting and sampling are known to be inter-reducible, another line of work would be to investigate the development of an almost-uniform sampler for MUSes, which can potentially benefit from the framework proposed in UniGen [14,16]. Another line of work is to extend our MUS counting approach to other constraint domains where MUSes find an application, e.g., F can be a set of SMT [25] or LTL [4,8] formulas or a set of transition predicates [13,23].

Acknowledgments. This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004] and AI Singapore Programme [AISG-RP-2018-005], and NUS ODPRT Grant [R-252-000-685-13]. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore https://www.nscc.sg.

References

1. Arif, M.F., Mencía, C., Ignatiev, A., Manthey, N., Peñaloza, R., Marques-Silva, J.: BEACON: an efficient SAT-based tool for debugging $\mathcal{EL}^+$ ontologies. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 521–530. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_32
2. Bacchus, F., Katsirelos, G.: Using minimal correction sets to more efficiently compute minimal unsatisfiable sets. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 70–86. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_5

3. Bacchus, F., Katsirelos, G.: Finding a collection of MUSes incrementally. In: Quimper, C.-G. (ed.) CPAIOR 2016. LNCS, vol. 9676, pp. 35–44. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33954-2_3
4. Barnat, J., Bauch, P., Beneš, N., Brim, L., Beran, J., Kratochvíla, T.: Analysing sanity of requirements for avionics systems. Formal Aspects Comput. **28**(1), 45–63 (2015). https://doi.org/10.1007/s00165-015-0348-9
5. Belov, A., Heule, M.J.H., Marques-Silva, J.: MUS extraction using clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 48–57. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_5
6. Belov, A., Marques-Silva, J.: Accelerating MUS extraction with recursive model rotation. In: FMCAD, pp. 37–40. FMCAD Inc. (2011)
7. Belov, A., Marques-Silva, J.: MUSer2: an efficient MUS extractor. JSAT **8**, 123–128 (2012)
8. Bendík, J.: Consistency checking in requirements analysis. In: ISSTA, pp. 408–411. ACM (2017)
9. Bendík, J., Beneš, N., Černá, I., Barnat, J.: Tunable online MUS/MSS enumeration. In: FSTTCS. LIPIcs, vol. 65, pp. 50:1–50:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)
10. Bendík, J., Černá, I.: Evaluation of domain agnostic approaches for enumeration of minimal unsatisfiable subsets. In: LPAR. EPiC Series in Computing, vol. 57, pp. 131–142. EasyChair (2018)
11. Bendík, J., Černá, I.: MUST: minimal unsatisfiable subsets enumeration tool. TACAS 2020. LNCS, vol. 12078, pp. 135–152. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_8
12. Bendík, J., Černá, I., Beneš, N.: Recursive online enumeration of all minimal unsatisfiable subsets. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 143–159. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_9
13. Bendík, J., Ghassabani, E., Whalen, M., Černá, I.: Online enumeration of all minimal inductive validity cores. In: Johnsen, E.B., Schaefer, I. (eds.) SEFM 2018. LNCS, vol. 10886, pp. 189–204. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92970-5_12
14. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: On parallel scalable uniform SAT witness generation. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 304–319. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_25
15. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable approximate model counter. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 200–216. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_18
16. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Balancing scalability and uniformity in SAT witness generator. In: Proceedings of DAC (2014)
17. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Algorithmic improvements in approximate counting for probabilistic inference: from linear to logarithmic SAT calls. In: Proceedings of IJCAI (2016)
18. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Algorithmic improvements in approximate counting for probabilistic inference: from linear to logarithmic SAT calls. In: IJCAI, pp. 3569–3576. IJCAI/AAAI Press (2016)
19. Chen, Z.-Z., Toda, S.: The complexity of selecting maximal solutions. Inf. Comput. **119**(2), 231–239 (1995)
20. Orly, C., Moran, G., Michael, L., Alexander, N., Vadim, R.: Designers work less with quality formal equivalence checking. In: DVCon. Citeseer (2010)

21. de Kleer, J., Williams, B.C.: Diagnosing multiple faults. Artif. Intell. **32**(1), 97–130 (1987)
22. Elffers, J., Giráldez-Cru, J., Gocht, S., Nordström, J., Simon, L.: Seeking practical CDCL insights from theoretical sat benchmarks. In: IJCAI, pp. 1300–1308. International Joint Conferences on Artificial Intelligence Organization, July 2018
23. Ghassabani, E., Gacek, A., Whalen, M.W., Heimdahl, M.P.E., Wagner, L.G.: Proof-based coverage metrics for formal verification. In: ASE, pp. 194–199. IEEE Computer Society (2017)
24. Gomes, C.P., Sabharwal, A., Selman, B.: Near-uniform sampling of combinatorial spaces using XOR constraints. In: NIPS, pp. 481–488. MIT Press (2006)
25. Guthmann, O., Strichman, O., Trostanetski, A.: Minimal unsatisfiable core extraction for SMT. In: FMCAD, pp. 57–64. IEEE (2016)
26. Hunter, A., Konieczny, S.: Measuring inconsistency through minimal inconsistent sets. In: KR, pp. 358–366. AAAI Press (2008)
27. Ignatiev, A., Morgado, A., Marques-Silva, J.: PySAT: a Python toolkit for prototyping with SAT Oracles. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) SAT 2018. LNCS, vol. 10929, pp. 428–437. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_26
28. Ivrii, A., Malik, S., Meel, K.S., Vardi, M.Y.: On computing minimal independent support and its applications to sampling and counting. Constraints **21**(1), 41–58 (2015). https://doi.org/10.1007/s10601-015-9204-z
29. Jannach, D., Schmitz, T.: Model-based diagnosis of spreadsheet programs: a constraint-based debugging approach. Autom. Softw. Eng. **23**(1), 105–144 (2014). https://doi.org/10.1007/s10515-014-0141-7
30. Janota, M., Marques-Silva, J.: On deciding MUS membership with QBF. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 414–428. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23786-7_32
31. Büning, H.K., Kullmann, O.: Minimal unsatisfiability and autarkies. In: Handbook of Satisfiability. FAIA, vol. 185, pp. 339–401. IOS Press (2009)
32. Kullmann, O.: Investigations on autark assignments. Discrete Appl. Math. **107**(1–3), 99–137 (2000)
33. Kullmann, O., Marques-Silva, J.: Computing maximal autarkies with few and simple Oracle queries. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 138–155. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_11
34. Liffiton, M.H., Malik, A.: Enumerating infeasibility: finding multiple MUSes quickly. In: Gomes, C., Sellmann, M. (eds.) CPAIOR 2013. LNCS, vol. 7874, pp. 160–175. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38171-3_11
35. Liffiton, M.H., Previti, A., Malik, A., Marques-Silva, J.: Fast, flexible MUS enumeration. Constraints **21**(2), 223–250 (2016). https://doi.org/10.1007/s10601-015-9183-0
36. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reasoning **40**(1), 1–33 (2008). https://doi.org/10.1007/s10817-007-9084-z
37. Lonsing, F., Egly, U.: QRATPre+: effective QBF preprocessing via strong redundancy properties. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 203–210. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_14
38. Marques-Silva, J., Ignatiev, A., Morgado, A., Manquinho, V.M., Lynce, I.: Efficient autarkies. In: ECAI. FAIA, vol. 263, pp. 603–608. IOS Press (2014)
39. Marques-Silva, J., Janota, M.: On the query complexity of selecting few minimal sets. Electron. Colloquium Comput. Complex. (ECCC) **21**, 31 (2014)

40. Marques-Silva, J., Lynce, I.: On improving MUS extraction algorithms. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 159–173. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21581-0_14
41. Meel, K.S., Shrotri, A.A., Vardi, M.Y.: Not all FPRASs are equal: demystifying FPRASs for DNF-counting. Constraints **24**(3), 211–233 (2018). https://doi.org/10.1007/s10601-018-9301-x
42. Mencía, C., Kullmann, O., Ignatiev, A., Marques-Silva, J.: On computing the union of MUSes. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 211–221. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_15
43. Meyer, A.R., Stockmeyer, L.J.: The equivalence problem for regular expressions with squaring requires exponential space. In: SWAT (FOCS), pp. 125–129. IEEE Computer Society (1972)
44. Monien, B., Speckenmeyer, E.: Solving satisfiability in less than 2^n steps. Discrete Appl. Math. **10**(3), 287–295 (1985)
45. Kedian, M.: Formulas free from inconsistency: an atom-centric characterization in priest's minimally inconsistent LP. J. Artif. Intell. Res. **66**, 279–296 (2019)
46. Nadel, A., Ryvchin, V., Strichman, O.: Accelerated deletion-based extraction of minimal unsatisfiable cores. JSAT **9**, 27–51 (2014)
47. Narodytska, N., Bjørner, N., Marinescu, M.-C., Sagiv, M.: Core-guided minimal correction set and core enumeration. In: IJCAI, pp. 1353–1361 (2018). ijcai.org
48. Piotrów, M.: Uwrmaxsat-a new minisat+-based solver in maxsat evaluation 2019. In: MaxSAT Evaluation 2019, p. 11 (2019)
49. Rabe, M.N., Tentrup, L.: CAQE: a certifying QBF solver. In: FMCAD, pp. 136–143. IEEE (2015)
50. Rabe, M.N., Tentrup, L., Rasmussen, C., Seshia, S.A.: Understanding and extending incremental determinization for 2QBF. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 256–274. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_17
51. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **32**(1), 57–95 (1987)
52. Soos, M., Meel, K.S.: BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In: Proceedings of the AAAI (2019)
53. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_24
54. Sperner, E.: Ein satz über untermengen einer endlichen menge. Math. Z. **27**(1), 544–548 (1928). https://doi.org/10.1007/BF01171114
55. Stern, R.T., Kalech, M., Feldman, A., Provan, G.M.: Exploring the duality in conflict-directed model-based diagnosis. In: AAAI. AAAI Press (2012)
56. Thimm, M.: On the evaluation of inconsistency measures. Meas. Inconsistency Inf. **73**, 19–60 (2018)

Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling

Mate Soos[1], Stephan Gocht[2], and Kuldeep S. Meel[1(✉)]

[1] School of Computing, National University of Singapore, Singapore, Singapore
meel@comp.nus.edu.sg
[2] Lund University, Lund, Sweden

Abstract. Given a Boolean formula, the problem of counting seeks to estimate the number of solutions of F while the problem of uniform sampling seeks to sample solutions uniformly at random. Counting and uniform sampling are fundamental problems in computer science with a wide range of applications ranging from constrained random simulation, probabilistic inference to network reliability and beyond. The past few years have witnessed the rise of hashing-based approaches that use XOR-based hashing and employ SAT solvers to solve the resulting CNF formulas conjuncted with XOR constraints. Since over 99% of the runtime of hashing-based techniques is spent inside the SAT queries, improving CNF-XOR solvers has emerged as a key challenge.

In this paper, we identify the key performance bottlenecks in the recently proposed BIRD architecture, and we focus on overcoming these bottlenecks by accelerating the XOR handling within the SAT solver and on improving the solver integration through a smarter use of (partial) solutions. We integrate the resulting system, called BIRD2, with the state of the art approximate model counter, ApproxMC3, and the state of the art almost-uniform model sampler UniGen2. Through an extensive evaluation over a large benchmark set of over 1896 instances, we observe that BIRD2 leads to consistent speed up for both counting and sampling, and in particular, we solve 77 and 51 more instances for counting and sampling respectively.

1 Introduction

A CNF-XOR formula φ is represented as conjunction of two Boolean formulas $\varphi_{\text{CNF}} \wedge \varphi_{\text{XOR}}$ wherein φ_{CNF} is represented in Conjunctive Normal Form (CNF) and φ_{XOR} is represented as conjunction of XOR constraints. While owing to the NP-completeness of CNF, every CNF-XOR formula can be represented as a CNF formula with only a linear increase in the size of the resulting formula, such a transformation may not be ideal in several scenarios. In particular, it is

The resulting tools ApproxMC4 and UniGen3 are available open source at https://github.com/meelgroup/approxmc and https://github.com/meelgroup/unigen.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 463–484, 2020.
https://doi.org/10.1007/978-3-030-53288-8_22

well known that modern Conflict Driven Clause Learning (CDCL) SAT solvers perform poorly on XOR formulas represented in CNF form despite the existence of efficient polynomial time decision procedures for XOR constraints. Furthermore, constraints arising from domains such as cryptanalysis and circuits can be naturally described as CNF-XOR formulas and these domains served as the early inspiration for design of SAT solvers with native support for XORs through the usage of Gaussian Elimination. These efforts lead to the development of CryptoMiniSat, a SAT solver that sought to perform Conflict Driven Clause Learning and Gaussian Elimination in tandem. The architecture of the early verisons of CryptoMiniSat sought to employ disjoint storage of CNF and XOR clauses – reminiscent to the architecture of SMT solvers.

While CryptoMiniSat was originally designed for cryptanalysis, its ability to handle XORs natively has led it to be a fundamental building block of the hashing-based techniques for approximate model counting and sampling. Model counting, also known as #SAT, and uniform sampling of solutions for Boolean formulas are two fundamental problems in computer science with a wide variety of applications [1,11,18]. The core idea of hashing-based techniques for approximate counting and almost-uniform sampling is to employ XOR-based 3-wise independent hash functions[1] to partition the solution space of F into *roughly equal small* cells of solutions. The usage of XOR-based hash functions allows us to represent a cell as conjunction of a Boolean formula in conjunctive normal form (CNF) and XOR constraints, and a SAT solver is invoked to enumerate solutions inside a randomly chosen cell. The corresponding counting and sampling algorithms typically employ the underlying solver in an incremental fashion and invoke the solver thousands of times, thereby necessitating the need for runtime efficiency. In this context, Soos and Meel [19] observed that the original architecture of CryptoMiniSat did not allow a straightforward integration of pre- and in-processing which of late has emerged to be key techniques in SAT solving. Accordingly, Soos and Meel [19] proposed a new architecture, called BIRD, that relied on the key idea of keeping the XOR constraints in both CNF form and XOR form. Soos and Meel integrated BIRD into CryptoMiniSat, and showed that state of the art approximate model counter, ApproxMC, when integrated with the new version of CryptoMiniSat achieves significant runtime improvements. The resulting version of ApproxMC was called ApproxMC3.

Motivated by the success of BIRD in achieving significant runtime performance improvements, we sought to investigate the key bottlenecks in the runtime performance of CryptoMiniSat when handling CNF+XOR formulas. Given the prominent usage of CNF-XOR formulas by the hashing based techniques, we study the runtime behavior of CryptoMiniSat for the the queries issued by the hashing-based approximate counters and samplers, ApproxMC3 and UniGen2 respectively. Our investigation leads us to make five core technical contributions. The first four contributions contribute towards architectural advances in han-

[1] While approximate counting techniques [10] only require 2-wise independent hash functions, hashing-based sampling techniques [6,9] require 3-wise independent hash functions.

dling of CNF-XOR formulas while the fifth contribution focuses on algorithmic improvements in the hashing-based techniques for counting and sampling:

1. **Matrix row handling improvements** for efficient propagation and conflict checking of XOR constraints
2. **XOR constraint detaching** from the standard unit propagation system for higher unit propagation speed
3. **Lazy reason clause generation** to reduce reason generation overhead for unused reasons generated from XOR constraints
4. **Allowing partial solution extraction** by the SAT solver
5. **Intelligent reuse of solutions** by hashing-based techniques to reduce the number of SAT calls

We integrate these improvements into the BIRD framework, the resulting framework is called BIRD2. The BIRD2 framework is applied to state of the art approximate model counter, ApproxMC3, and to the almost-uniform sampler UniGen2 [6,9]. The resulting counter and sampler are called ApproxMC4 and UniGen3 respectively. We conducted an extensive empirical evaluation with over 1800 benchmarks arising from diverse domains with computational effort totalling 50,000 CPU hours. With a timeout of 5000 s, ApproxMC3 and UniGen2+BIRD were able to solve only 1148 and 1012 benchmarks, while ApproxMC4 and UniGen3 solved 1225 and 1063 benchmarks respectively. Furthermore, we observe a consistent speedup for most of the benchmarks that could be solved by ApproxMC3 and UniGen2+BIRD. In particular, the PAR-2[2] score improved from 4146 with ApproxMC3 to 3701 with ApproxMC4. Similarly, the corresponding PAR-2 scores for UniGen3 and UniGen2+BIRD improved to 4574 from 4878.

2 Notations and Preliminaries

Let F be a Boolean formula in conjunctive normal form (CNF) and $\mathsf{Vars}(F)$ the set of variables in F. Unless otherwise stated, we use n to denote the number of variables in F i.e., $n = |\mathsf{Vars}(F)|$. An assignment of truth values to the variables in $\mathsf{Vars}(F)$ is called a *satisfying assignment* or *witness* of F if it makes F evaluate to true. We denote the set of all witnesses of F by $sol(F)$. If we are only interested in a subset of variables $S \subseteq \mathsf{Vars}(F)$ we will use $sol(F)_{\downarrow S}$ to indicate the projection of $sol(F)$ on S.

The problem of *propositional model counting* is to compute $|sol(F)|$ for a given CNF formula F. A *probably approximately correct* (or PAC) counter is a probabilistic algorithm $\mathsf{ApproxCount}(\cdot,\cdot,\cdot)$ that takes as inputs a formula F, a tolerance $\varepsilon > 0$, and a confidence $1-\delta \in (0,1]$, and returns a count c with (ε,δ)-guarantees, i.e., $\mathsf{Pr}\Big[|sol(F)|/(1+\varepsilon) \leq c \leq (1+\varepsilon)|sol(F)|\Big] \geq 1-\delta$. Projected

[2] PAR-2 score, that is, penalized average runtime, assigns a runtime of two times the time limit (instead of a "not solved" status) for each benchmark not solved by a tool.

model counting is defined analogously using $sol(F)_{\downarrow S}$ instead of $sol(F)$, for a given sampling set $S \subseteq \mathsf{Vars}(F)$.

A *uniform sampler* outputs a solution $y \in sol(F)$ such that $\mathsf{Pr}[y \text{ is output}] = \frac{1}{|sol(F)|}$. An *almost-uniform sampler* relaxes the guarantee of uniformity and in particular, ensures that $\frac{1}{(1+\varepsilon)|sol(F)|} \leq \mathsf{Pr}[y \text{ is output}] \leq \frac{1+\varepsilon}{|sol(F)|}$.

Universal Hash Functions. Let $n, m \in \mathbb{N}$ and $\mathcal{H}(n,m) \triangleq \{h : \{0,1\}^n \to \{0,1\}^m\}$ be a family of hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$. We use $h \xleftarrow{R} \mathcal{H}(n,m)$ to denote the probability space obtained by choosing a function h uniformly at random from $\mathcal{H}(n,m)$. To measure the quality of a hash function we are interested in the set of elements of S mapped to α by h, denoted $\mathsf{Cell}_{\langle S,h,\alpha\rangle}$ and its cardinality, i.e., $|\mathsf{Cell}_{\langle S,h,\alpha\rangle}|$. To avoid cumbersome terminology, we abuse notation slightly and we use $\mathsf{Cell}_{\langle F,m\rangle}$ (resp. $\mathsf{Cnt}_{\langle F,m\rangle}$) as shorthand for $\mathsf{Cell}_{\langle sol(F),h,\alpha\rangle}$ (resp. $|\mathsf{Cell}_{\langle sol(F),h,\alpha\rangle}|$).

Definition 1. *A family of hash functions $\mathcal{H}(n,m)$ is k-wise independent*[3] *if $\forall \alpha_1, \alpha_2, \ldots \alpha_k \in \{0,1\}^m$ and for distinct $\mathbf{y}_1, \mathbf{y}_2, \ldots \mathbf{y}_k \in \{0,1\}^n$, $h \xleftarrow{R} \mathcal{H}(n,m)$,*

$$\mathsf{Pr}\left[(h(\mathbf{y}_1) = \alpha_1) \wedge (h(\mathbf{y}_2) = \alpha_2) \ldots \wedge (h(\mathbf{y}_k) = \alpha_k)\right] = \left(\frac{1}{2^m}\right)^k \tag{1}$$

Note that every k-wise independent hash family is also $k-1$ wise independent.

Prefix Slicing. While universal hash families have nice concentration bounds, they are not adaptive, in the sense that one cannot build on previous queries. In several applications of hashing, the dependence between different queries can be exploited to extract improvements in theoretical complexity and runtime performance. Thus, we are typically interested in prefix slices of hash functions [10] as follows.

Definition 2. *For every $m \in \{1, \ldots n\}$, the m^{th} prefix-slice of h, denoted $h^{(m)}$, is a map from $\{0,1\}^n$ to $\{0,1\}^m$, such that $h^{(m)}(\mathbf{y})[i] = h(\mathbf{y})[i]$, for all $y \in \{0,1\}^n$ and for all $i \in \{1, \ldots m\}$. Similarly, the m^{th} prefix-slice of α, denoted $\alpha^{(m)}$, is an element of $\{0,1\}^m$ such that $\alpha^{(m)}[i] = \alpha[i]$ for all $i \in \{1, \ldots m\}$.*

Explicit Hash Functions. The most common explicit hash family used in state of the art sampling and counting techniques is based on random XOR constraints. Viewing $\mathsf{Vars}(F)$ as a vector $\boldsymbol{x}$ of dimension $n \times 1$, we can represent the hash family as follows: Let $\mathcal{H}_{xor}(n,m) \triangleq \{h : \{0,1\}^n \to \{0,1\}^m\}$ be the family of functions of the form $h(x) = \boldsymbol{M}\boldsymbol{x} + \boldsymbol{b}$ with $\boldsymbol{M} \in \mathbb{F}_2^{m \times n}$ and $\boldsymbol{b} \in \mathbb{F}_2^{m \times 1}$ where the entries of $\boldsymbol{M}$ and b are independently generated according to the

[3] The phrase *strongly 2-universal* is also used to refer to 2-wise independent as noted by Vadhan in [23], although the concept of 2-universal hashing proposed by Carter and Wegman [4] only required that $\mathsf{Pr}[h(x) = h(y)] \leq \frac{1}{2^m}$.

Bernoulli distribution with probability 1/2. Observe that $h^{(m)}(x)$ can be written as $h^{(m)}(\boldsymbol{x}) = \boldsymbol{M}^{(m)}\boldsymbol{x} + \boldsymbol{b}^{(m)}$, where $\boldsymbol{M}^{(m)}$ denotes the submatrix formed by the first m rows and n columns of $\boldsymbol{M}$ and $\boldsymbol{b}^{(m)}$ is the first m entries of the vector $\boldsymbol{b}$. It is well known that $\mathcal{H}_{xor}$ is 3-wise independent [9].

3 Background

The general idea of hashing-based model counting and sampling is to use a hash function from a suitable family, e.g. $\mathcal{H}_{xor}$, to divide the solution space into cells that are sufficiently small such that all solutions within a cell can be enumerated efficiently. Given such a cell, its size can then be used to estimate the total count of solutions or we can return a random element of this small cell to produce a sample. Hence, hashing-based sampling and counting are closely related.

3.1 Hashing-Based Model Counting

The seminal work of Valiant [24] established that #SAT is #P-complete. Toda [22] showed that the entire polynomial hierarchy is contained inside the complexity class defined by a polynomial time Turing machine equipped with #P oracle. Building on Carter and Wegman's [4] seminal work of universal hash functions, Stockmeyer [21] proposed a probabilistic polynomial time procedure relative to an NP oracle to obtain an (ε, δ)-approximation of F.

The core theoretical idea of the hashing-based approximate solution counting framework proposed in ApproxMC [8], building on Stockmeyer [21], is to employ 2-universal hash functions to partition the solution space, denoted by $sol(F)$ for a formula F, into *roughly equal small* cells, wherein a cell is called *small* if it has solutions less than or equal to a pre-computed threshold, thresh. An NP oracle is employed to check if a cell is small by enumerating solutions one-by-one until either there are no more solutions or we have already enumerated thresh + 1 solutions. In practice, a SAT solver is used to realize the NP oracle. To ensure polynomially many calls to the oracle, thresh is set to be polynomial in the input parameter ε. To determine the right number of cells, i.e., the value of m for $\mathcal{H}(n, m)$, a search procedure is invoked. Finally, the subroutine, called ApproxMCCore, computes the estimate as the number of solutions in the randomly chosen cell scaled by the number of cells (i.e, 2^m). To achieve probabilistic amplification of the confidence, multiple invocations of the underlying subroutine, ApproxMCCore, are performed with the final count computed as the median of estimates returned by ApproxMCCore.

Two key algorithmic improvements proposed in ApproxMC2 [10] are significant to practical performance: (1) the search for the right number of cells can be performed via galloping search, and (2) one can first perform linear search over a small enough interval (chosen to be of size 7) around the value of m found in the previous iteration of ApproxMCCore. The practical profiling of ApproxMC2 reveals that linear search is sufficient after the first invocation of ApproxMCCore. Note that the linear search seeks to identify a value of m such that $\mathsf{Cnt}_{\langle F, m-1\rangle} \geq$ thresh

and $\mathsf{Cnt}_{\langle F,m\rangle} < \mathsf{thresh}$ for an appropriately chosen thresh. ApproxMC is currently in its third generation: ApproxMC3.

3.2 Hashing-Based Sampling

Jerrum, Valiant, and Vazirani [14] showed that the approximate counting and almost-uniform counting are polynomially inter-reducible. Building on Jerrum et al.'s result, Bellare, Goldreich, and Petrank [2] proposed a probabilistic uniform generator that makes polynomially many calls to an NP oracle where each NP query is the input formula F conjuncted with constraints encoding a degree n polynomially representing n-wise independent hash functions where n is the number of variables in F. The practical implementation of Bellare et al.'s technique did not scale beyond few tens of variables. Chakraborty, Meel, and Vardi [7,9], sought to combine the inter-reducibility and the usage of independent hashing, and proposed a hashing-based framework, called UniGen, that employs 3-wise independent hashing and makes polynomially many calls to an NP oracle.

The core theoretical idea of the hashing-based sampling framework, proposed in UniGen, exploits the close relationship between counting and sampling. UniGen first invokes ApproxMC to compute an estimate of the number of solutions of the given formula F. It then uses the count to determine the number of cells that the solution space should be partitioned into using 3-wise independent hash functions. At this point, it is worth mentioning that the state of the art hashing-based sampling employ 3-wise independent hash functions. Fortunately, the family of hash functions, $\mathcal{H}_{xor}$, is also known to be 3-wise independent. There after, similar to ApproxMC, a linear search over a small enough interval (chosen to be of size 4) is invoked to find the *right* value of m where a randomly chosen cell's size is within the desired bounds. For such a cell, all its solutions are enumerated and one of the solutions is randomly chosen. Again, similar to ApproxMC2 (and ApproxMC3), the linear search seeks to identify a value of m such that $\mathsf{Cnt}_{\langle F,m-1\rangle} \geq \mathsf{thresh}$ and $\mathsf{Cnt}_{\langle F,m\rangle} < \mathsf{thresh}$ for an appropriately chosen thresh. UniGen is currently in its second generation: UniGen2 [6].

3.3 The Underlying SAT Solver

The underlying SAT solver is invoked through subroutine BoundedSAT, which is implemented using CryptoMiniSat. Formally, BoundedSAT takes as inputs a formula F, a threshold thresh, and a sampling set S, and returns a subset Y of $sol(F)_{\downarrow S}$, such that $|Y| = \min(\mathsf{thresh}, |sol(F)_{\downarrow S}|)$. The formula F consists of the original formula, which we want to count or sample, conjuncted with a set of XOR constraints defined through a hash function sampled from the family $\mathcal{H}_{xor}$. We henceforth denote such formulas as CNF-XOR formulas. Note that the efficient encoding of XOR constraints into CNF requires the introduction of new variables and hence the sampling set S usually does not contain all variables in F.

As is consistent with prior studies, profiling of ApproxMC3 and UniGen2 reveal that over 99% of the time is spent in the runtime of BoundedSAT. Therefore recent efforts have focused on improving BoundedSAT. Soos and Meel [19] sought to address the performance of the underlying SAT solver by proposing a new architecture, called BIRD, that allows the usage of in- and pre-processing techniques for a Gauss Jordan Elimination (GJE)-augmented SAT solver. ApproxMC2, integrated with BIRD, called ApproxMC3, gave up to three orders of magnitude runtime performance improvement. Such significant improvements are rare in the SAT community. Encouraged by Soos and Meel's observations, we seek to build on top of BIRD to achieve an even tighter integration of the underlying SAT solver and ApproxMC3/UniGen2.

BIRD: Blast, Inprocess, Recover, and Destroy. Pre- and inprocessing techniques are known to have a large impact on the runtime performance of SAT solvers. However, earlier Guassian elimination architectures were unable to perform these techniques. Motivated by this inability, Soos and Meel [19] proposed a new framework, called BIRD, that allows usage of inprocessing techniques for GJE-augmented CDCL solvers. The key idea of BIRD is to blast XOR clauses into CNF clauses so that any technique working solely on CNF clauses does not violate soundness of the solver. To perform Gauss-Jordan elimination, one needs efficient algorithms and data structures to extract XORs from CNF. The entire framework is presented as follows:

BIRD: Blast, In-process, Recover, and Destroy

Step 1 **Blast** XOR clauses into normal CNF clauses
Step 2 **Inprocess** (and pre-process) over CNF clauses
Step 3 **Recover** simplified XOR clauses
Step 4 Perform CDCL on CNF clauses with on-the-fly Gauss-Jordan Elimination (GJE) on XOR clauses until inprocessing is scheduled
Step 5 **Destroy** XOR clauses and goto **Step 2**

The above loop terminates as soon as a satisfying assignment is found or the formula is proven UNSAT. The BIRD architecture separates inprocessing from CDCL solving and therefore every sound inprocessing step can be employed.

4 Technical Contributions to CNF-XOR Solving

Inspired by the success of BIRD, we seek to further improve the underlying SAT solver's architecture based on the queries generated by the hashing-based techniques. To this end, we relied on extensive profiling of CryptoMiniSat augmented with BIRD to identify the key performance bottlenecks, and propose solutions to overcome some of the challenges.

4.1 Detaching XOR Clauses from Watch-Lists

Given a formula F in CNF, the recovery phase of BIRD attempts to construct a set of XORs, H such that $F \rightarrow H$. As detailed in [19], the core technique for recovery of an XOR of size k is to establish whether the required 2^{k-1} combinations of clauses are implied by the existing CNF clauses. For example, the XOR $x_1 \oplus x_2 \oplus x_3 = 0$ (where $k = 3$) can be recovered if the existing set of CNF clauses implies the following $4(= 2^{3-1})$ clauses: $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$. To this end, the first stage of the recovery phase of BIRD iterates over the CNF clauses and for a given clause, called base_cl of size k, searches whether the remaining $2^{k-1} - 1$ clauses are implied as well, in which case the resulting XOR is added. It is worth noting that a clause can imply multiple clauses over the the set of variables of base_cl; For example if the base_cl $= (x_1 \vee \neg x_2 \vee x_3)$, then the clause $(\neg x_1)$ would imply the two clauses $(\neg x_1 \vee \neg x_2 \vee \neg x_3)$ and $(\neg x_1 \vee x_2 \vee x_3)$. Note that given a base_cl, we are only interested in clauses over the variables in base_cl.

During blasting of XORs into CNF, XORs are first cut into smaller XORs by introducing auxiliary variables. Hence, the first stage of recovery phase must recover these smaller XORs and the second phase reconstructs the larger XORs by XOR-ing two XORs together if they differ only on one variable, referred to as a *clash variable*. For example, $x_1 \oplus x_2 \oplus x_3 = 0$ and $x_3 \oplus x_4 \oplus x_5 = 1$ can be XOR-ed together over clash variable x_3 to obtain $x_1 \oplus x_2 \oplus x_4 \oplus x_5 = 1$.

Since BIRD performs CDCL in tandem with Gauss-Jordan elimination, it is worth noting that the Gauss-Jordan elimination (GJE)-based decision procedure is sound and complete, i.e., all unit propagations and conflicts implied by the given set of XORs would be discovered by a GJE-based decision procedure. For the initial formula (in CNF) F and the recovered set of XORs, H, if a set of CNF clauses G is implied by H, then presence or absence of G does not affect soundness and completeness of GJE-augmented CDCL engine. Our extensive profiling of the BIRD framework integrated in CryptoMiniSat revealed a significant time spent in examination of clauses in G during unit propagation. To this end, we sought to ask how to design an efficient technique to find all the CNF clauses implied by the recovered XORs. These clauses could be detached from unit propagation without any negative effect on correctness of execution.

A straightforward approach would be to mark all the clauses during the blasting phase of XORs into CNF. However, the incompleteness of the recovery phase of BIRD does not guarantee that all such marked clauses are indeed implied by the recovered set of XORs. Another challenge in the search for detachable clauses arises due to construction of larger XORs by combining smaller XORs. For example, while $x_1 \oplus x_2 \oplus x_3 = 0$ and $x_3 \oplus x_4 \oplus x_5 = 1$ imply $(x_1 \vee x_2 \vee \neg x_3)$ and $(x_3 \vee x_4 \vee x_5)$, the combined XOR $x_1 \oplus x_2 \oplus x_4 \oplus x_5 = 1$ does not imply $(x_1 \vee x_2 \vee \neg x_3)$ and $(x_3 \vee x_4 \vee x_5)$.

Two core insights inform our design of the modification of the recovery phase and search for detachable clauses. Firstly, given a base clause base_cl, if a clause cl participates in the recovery of XORs over the variables in base_cl, then cl is

implied by the recovered XOR if the number of variables in cl is the same as that of base_cl. We call such a clause cl a *fully participating clause*. Secondly, let G_1 and G_2 be the set of CNF clauses implied by two XORs q_1 and q_2 that share exactly one variable, say x_i. Let $U = (\mathsf{Vars}(q_1) \cup \mathsf{Vars}(q_2)) \backslash x_i$. Let q_3 be the XOR obtained by XORing together q_1 and q_2, then, $sol(q_3)_{\downarrow U} \subseteq sol(G_1 \wedge G_2)_{\downarrow U}$ if x_i does not appear in the remaining clauses, i.e., $x_i \notin \mathsf{Var}[F \setminus (G_1 \cup G_2)]$.

The above two insights lead us to design a modified recovery and detachment phase as follows. During recovery, we add every *fully participating clause* to the set of detachable clauses D. Let $\mathcal{U} = S \cup (\mathsf{Vars}(D) \cap \mathsf{Vars}(F \backslash D))$. Then, the recovery of longer XORs is only performed over clash variables that do not belong to $\mathcal{U}$. We then detach the clauses in D from watch-lists during GJE-augmented CDCL phase, mark the clash variables as non-decision variables, perform CDCL, and only reattach the clauses and re-set the clash variables to be decision variables after the Destroy phase of BIRD.

If the formula is satisfiable, the design of the solver is such that the solution is always found during the GJE-augmented CDCL solving phase. Since clauses in D are detached and the clash variables are set to be not decided on during this phase, the clash variables are always left unassigned. As discussed below, however, we only need to extract solutions over the sampling set S, therefore the solution found is adequate as-is, without the clash variables, which are by definition not over S as they are only introduced for having short encodings of XORs into CNF.

Conceptually, this approach reconciles the overhead introduced by BIRD, i.e., that XOR constraints are also present as regular clauses, with the neatness of the original CryptoMiniSat that stored XOR and regular constraints in different data structures. This reconciliation takes the best of both worlds.

4.2 Fast Propagation/Conflict Detection and Reason Generation

We identified two key bottlenecks in the the current GJE component of BIRD framework integrated in CryptoMiniSat, which we sought to improve upon. To put our contributions in the context, we first describe the technical details of the core data structures and algorithms.

Han-Jiang's GJE. To perform Gaussian elimination on a set of XORs, the XORs are represented as a matrix where each row represents an XOR and each column represents a variable. The framework proposed by Soos et al. updates the matrix whenever a variable is assigned and removes the assigned variable from all XORs by zeroing out the corresponding column. However, using the matrix in such a way involves significant memory copying during backtracking due to having to revert the matrix to a previous version.

To avoid the overhead, Han and Jiang proposed a new framework [13] building on Simplex-like techniques that performs Gauss-Jordan elimination, i.e., using reduced row echelon form instead of row echelon form. The key data structure innovation was to employ a two-watched variable scheme for each row of the

matrix wherein the watched variables are called basic and non-basic variables. Essentially, the basic variables are the variables on the diagonal of a matrix in reduced row echelon form and hence every row has exactly one basic variable and the basic variable only occurs in one row. Similar to standard CDCL solving, when a matrix row's watch is assigned, the GJE component must determine whether the row (1) propagates, (2) needs to assign a new watch, (3) is satisfied, or (4) is conflicted. It is worth recalling that a row would propagate if all except one variable has been assigned and would conflict or be satisfied if all the variables in a row have been assigned. Furthermore, we need to find a new watch if a watched variable was assigned and there is more than one unassigned variable left. If a basic variable is replaced by a new watch then the two corresponding columns are swapped and the reduced row echelon form is recomputed. In practice swapping columns is avoided by keeping track of which column is a basic variable.

For propagation, checking for conflict, and conflict clause generation Han-Jiang proposed a sequential walk through a row that eagerly computes the reason clause and stops when it encounters a new watch variable or reaches until the end of the row. At that point, the system (1) knows whether the row is satisfied, propagating, or conflicted, and (2) if not satisfied, has eagerly computed the reason clause for the propagation or the conflict.

For general benchmarks where XOR constraints do not play an influential role in determining satisfiability of the underlying problem, the GJE component can be as small as 10% of the entire solving time. However, for formulas generated generated by hashing-based techniques, our profiling demonstrated several cases where the Gaussian elimination component could be very time consuming, taking up to 90% of solving time.

While the choice of GJE combined with clever data structure maintenance led to significant improvements of the runtime of Gaussian Elimination component, our profiling identified two processes as key bottlenecks: propagation checking and reason generation. We next discuss our proposed algorithmic improvements that achieve significant runtime improvement by addressing these bottlenecks.

Tinted Fast Unit Propagation. The core idea to achieve faster propagation is based on bit-level parallelism via the different native operations supported by modern CPUs. In particular, modern CPUs provide native support for basic bitwise operations on bit fields such as AND, INVERT, hamming weight computation (i.e., the number of non-zero entries), and *find first set* (i.e., finding the index of first non-zero bit). Given the widespread support of SIMD extensions, the above operations can be performed at the rate of 128...512 bits per instruction. Therefore, the core data structure represents every 0-1 vector as a bit field.

A set of XORs over n variables $x_1, \ldots, x_n$ is represented as $\boldsymbol{M}\boldsymbol{x} = \boldsymbol{b}$ for a 0-1 matrix $\boldsymbol{M}$ of size $m \times n$, 0-1 vector b of length m and $\boldsymbol{x} = (x_1, \ldots, x_n)^T$. Consider the $i-$th row of $\boldsymbol{M}$, denoted by $\boldsymbol{M}[i]$. Let $\boldsymbol{a}$ be a 0-1 vector of size n such that $\boldsymbol{a}[j]$=1 if the variable x_j is assigned True or False, and 0 in case

x_j is unassigned. Let $\boldsymbol{v}$ be a 0-1 vector of size n such that $\boldsymbol{v}[j] = 1$ if x_j is set to True and 0 otherwise. Let $\overline{\boldsymbol{z}}$ be the bitwise inverse of a 0-1 vector $\boldsymbol{z}$ and & be the bitwise AND operation. Let $W_{unass} = \mathsf{hamming\_weight}(\overline{\boldsymbol{a}}\&\boldsymbol{M}[i])$ the number of unassigned variables in the XOR represented by row i, and $W_{val} = \mathsf{hamming\_weight}(\boldsymbol{v}\&\boldsymbol{M}[i])$ the number of satisfied variables. We view the computation of W_{unass} and W_{val} as viewing the world of $\boldsymbol{M}$ through the tinted lens of $\boldsymbol{v}$ and $\overline{\boldsymbol{a}}$. Now, the following holds:

1. Row i is satisfied if and only if $W_{unass} = 0$ and $(W_{val} \mod 2) \oplus \boldsymbol{b}[i] = 0$.
2. Row i causes a conflict if and only if $W_{unass} = 0$ and $(W_{val} \mod 2) \oplus \boldsymbol{b}[i] = 1$.
3. Row i propagates if and only if $W_{unass} = 1$. Propagated variable is the one that corresponds to the column with the only bit set in $\overline{\boldsymbol{a}}\&M[i]$. The value propagated is $(W_{val} \mod 2) \oplus \boldsymbol{b}[i]$.
4. A new watch needs to be found for row i if and only if $W_{unass} \geq 2$. The new watch is any one of the variables corresponding to columns with the bits set to 1 in $\overline{\boldsymbol{a}}\&\boldsymbol{M}[i]$, except for the already existing watch variable.

Reason Generation. For propagation and conflict we generate the reason clauses for row i as follows. We forward-scan $\boldsymbol{M}[i]$ for all set bits and insert the corresponding variable into the reason clause as a literal that evaluates to false under the current assignment. In the case of propagation, the literal added for the propagated variable, say x_j, is added as literal $\neg x_j$ if $(W_{val} \mod 2) \oplus \boldsymbol{b}[i] = 0$ and x_j otherwise.

Example. For example, let $\boldsymbol{b}[i] = 1$ and $\boldsymbol{M}[i] = 10011$ corresponding to variables $x_1, x_2, \ldots x_5$ and assignments 1?11? respectively, where "?" indicates an unassigned variable. Then $\boldsymbol{a} = 10110, \overline{\boldsymbol{a}}\&\boldsymbol{M}[i] = 00001, W_{unass} = 1, \boldsymbol{v} = 10110, \boldsymbol{v}\&\boldsymbol{M}[i] = 10010, W_{val} = 2$ and $(W_{val} \mod 2) \oplus \boldsymbol{b}[i] = 1$. Therefore, this row propagates (case 3 above), and the reason generated is $(\neg x_1 \vee \neg x_4 \vee x_5)$. If the assignements were 11110, then $W_{unass} = 0$ and $(W_{val} \mod 2) \oplus \boldsymbol{b}[i] = 1$ so this row conflicts (case 2 above), with conflict clause $(\neg x_1 \vee \neg x_4 \vee x_5)$.

Performance. Notice that all cases only require bitwise and, inverse, hamming weight and find first set operations. To find a new watch in case 4 we first find the first bit that is set to 1 in $\bar{\boldsymbol{a}}\&\boldsymbol{M}$ by invoking find first set. In case the obtained index is the same as the existing watch variable, we remove the first 1-bit by left shifting and run find first set again to find the second 1-bit. Bitwise and and inverse are trivially single-assembly instructions. We use compiler intrinsics to execute find first set and hamming weight functions, which compile down to BSF and POPCNT in x86 assembly, respectively. It is worth pointing out that we keep the bit field representations of $\boldsymbol{a}$ and $\boldsymbol{v}$ synchronized when variables are assigned. During backtracking we reset these to zero and refill them as needed. For better cache efficiency, we use sequential set of bit-packed 64-bit integers to represent all bit-fields, rows, and matrices.

Although bit-packing is not a novel concept in the context of CNF-XOR solving, let us elaborate why we believe that our contribution is conceptually interesting. Soos et al. [20] used bit-packed pre- and post-evaluated matrices. Since

post-evaluated matrices lose information, they have to be saved and reloaded on backtracking. Han and Jiang's code [13] changed this to using pre-evaluated matrices only, which free the system from having to save and reload. But it was slow, because bit-by-bit evaluation had to happen on every matrix row read (thanks to the missing post-evaluation matrix). Our improved approach is essentially merging the best of both worlds: fast evaluation, without having to save and reload.

4.3 Lazy Reason Clause Generation

As discussed earlier, the current BIRD performs eager reason clause generation in a spirit similar to the original proposal by Han and Jiang. At the time of proposal of eager clause generation by Han and Jiang, the state of the art SAT solver at that time could solve problems with XOR clauses of sizes in few tens to few hundreds. The improved scalability, however, highlights the overhead due to eager reason clause generation. During our profiling, we observed that for several problems, the independent support of the underlying formula ranges in thousands, and therefore, leading to generation of reason clauses involving thousands of variables. The generation of such long reason clauses is time consuming and tedious. Furthermore, a significant fraction of reason clauses are never required during conflict analysis phase as we are, often, focused only on finding a 1UIP clause. Therefore, we seek to explore lazy reason clause generation.

Let the state of a clause c indicate whether c is satisfied, conflicted or undetermined (i.e., the clause is neither satisfied nor conflicted). The core design of our lazy generation technique is based on the following invariant satisfied by CDCL-based techniques: Once a (CNF/XOR) clause is satisfied or conflicted, the assignment to the variables in the clause does not change as long the state of the clause does not change. Observe that when a clause propagates, the propagated literal changes the state of the clause to satisfied. Furthermore, as long as all variables are assigned, the row will not participate in GJE because none of the contained variables can become a basic watch. Therefore, whenever an XOR clause propagates, we keep an index of the row and the propagating literal but do not compute the reason clause. Now, whenever a reason clause is requested, we compute the reason clause as detailed above and return a pointer to the computed reason clause, and index the computed clause by the corresponding row. To ensure correctness, whenever a row causes a propagation, we delete the existing reason clause but we do not eagerly compute the new corresponding reason clause. On the other hand, if a row is conflicting, the conflict analysis requires the reason clause immediately and as such the reason clause is eagerly computed.

Lazy reason clause generation allows us to skip the majority of reason clauses to be generated. Furthermore, given that a row cannot lead to more than one reason clause, it allows us to statically allocate memory for them. This is in stark contrast to the original implementation that not only eagerly computed all reason clauses, but also dynamically allocated memory for them, freeing the memory up during backtracking.

4.4 Skipping Solution Extension of Eliminated Variables

SAT solvers aim to present a clean and uncomplicated API interface with internal behavior typically hidden to enable fast pacing development of heuristics without necessitating change in the interface for the end users. While such a design philosophy allows easier integration, it may be an hindrance to achieving efficiency for the use cases that may not be seeking a simple off-the-shelf behavior. Given the surge of projected counting and sampling as the desired formulation, BoundedSAT is invoked with a sampling set and we are interested only in the assignment to variables in the sampling set. A naive solution would be to obtain a complete assignment over the entire set of variables and then extract an assignment over the desired sampling set. In this context, we wonder if we can terminate early after the variables in the sampling set are assigned. In modern SAT solvers, once the solver has determined that the formula is satisfied, the *solution extension* subroutine is invoked that extends the current partial assignment to a complete assignment. Upon profiling, we observed that, during solution extension, a significant time is spent in computing an assignment to the variables eliminated due to Bounded Variable Elimination (BVE) [12] during pre- and inprocessing. When a solution is found, the eliminated clauses must be re-examined in reverse, linear, order to make sure the eliminated variables in the model are correctly assigned. This examination process can be time-consuming on large instances with large portions of the CNF eliminated.

BVE is widely used in modern SAT solvers owing to its ability to eliminate a large subset of the input formula and thereby allowing compact data structures. While disabling BVE would eliminate the overhead during solution extension phase, it would also significantly degrade performance during solving phase. Since we are interested in solutions only over the sampling set, we disable the invocation of bounded variable elimination for variables in the sampling set. Therefore, whenever the SAT solver determines that the current partial assignment satisfies the formula, all the variables in the sampling set are assigned and we do not invoke solution extension. The disabling of solution extension can save significant (over 20%) time on certain instances.

4.5 Putting It All Together: BIRD2

We combine improvements proposed above into our new framework, called BIRD2, a namesake to capture the primary architecture of Blast, In-process, Recover, Detach, and Destroy. For completeness, we present the core skeleton of BIRD2 in Algorithm 1. BIRD2 terminates as soon as a satisfying assignment is found or the formula is proven UNSAT. Similar to BIRD, BIRD2 architecture separates inprocessing from CDCL solving and therefore every sound inprocessing step can be employed.

Algorithm 1. BIRD2(φ) ▷ *φ has a mix of CNF and XOR clauses*

1: **Blast** XOR clauses into normal CNF clauses
2: **In-process** (and pre-process) over CNF clauses
3: **Recover** XOR clauses
4: **Detach** CNF clauses implied by recovered XOR clauses
5: Perform CDCL on CNF clauses with on-the-fly *improved* GJE on XOR clauses until: (a) in-processing is scheduled, (b) a satisfying assignment is found, or (c) formula is found to be unsatisfiable
6: **Destroy** XOR clauses and reattach detached CNF clauses. Goto line 2 if conditions (b) or (c) above don't hold. Otherwise, return satisfying assignment or report unsatisfiable.

5 Technical Contribution to Counting and Sampling

In this section, we discuss our primary technical contribution to hashing-based sampling and counting techniques.

5.1 Reuse of Previously Found Solutions

The usage of a prefix-slicing ensures monotonicity of the random variable, $\mathsf{Cnt}_{\langle F,i\rangle}$, since from the definition of prefix-slicing, we have that for all i, $h^{(i+1)}(x) = \alpha^{(i+1)} \implies h^{(i)}(x) = \alpha^{(i)}$. Formally,

Proposition 1. *For all* $1 \leq i < m$, $\mathsf{Cell}_{\langle F,i+1\rangle} \subseteq \mathsf{Cell}_{\langle F,i\rangle}$

Furthermore as is evident from the analysis of ApproxMC3 [10], the pairwise independence of the family $\mathcal{H}_{xor}$ implies $\frac{\mathsf{E}[\mathsf{Cnt}_{\langle F,i\rangle}]}{\mathsf{E}[\mathsf{Cnt}_{\langle F,j\rangle}]} = 2^{j-i}$. Therefore, once we obtain the set of solutions from invocation of BoundedSAT for $F \wedge (h^i)^{-1}(\mathbf{0})$ (i.e., after putting i XORs), we can potentially reuse the returned solutions when we are interested in enumerating solutions for $F \wedge (h^j)^{-1}(\mathbf{0})$. In particular, note that if $i > j$, then Proposition 1 implies that all the solutions $F \wedge (h^i)^{-1}(\mathbf{0})$ are indeed solutions for $F \wedge (h^j)^{-1}(\mathbf{0})$ and we can invoke BoundedSAT with adjusted threshold. On the other hand, for $i < j$, we can check if the solutions of $F \wedge (h^i)^{-1}(\mathbf{0})$ also satisfy $F \wedge (h^{i+1})^{-1}(\mathbf{0})$.

On closer observation, we find that the latter case may not be always helpful when i and j differ by more than a small constant since the ratio of their expected number of solutions decreases exponentially with $j-i$. Interestingly, as discussed in Sect. 3, both ApproxMC3 and UniGen2 employ linear search over intervals of sizes 4 to 7. for the right values of m. In particular, for both ApproxMC3 and UniGen2, the linear search seeks to identify a value of m^* such that $\mathsf{Cnt}_{\langle F,m^*-1\rangle} \geq \mathsf{thresh}$ and $\mathsf{Cnt}_{\langle F,m^*\rangle} < \mathsf{thresh}$ for an appropriately chosen thresh. Therefore, when invoking BoundedSAT for $i = k$ after determining that for $i = k+1$, $\mathsf{Cnt}_{\langle F,k+1\rangle} < \mathsf{thresh}$, we can replace thresh with $\mathsf{thresh} - \mathsf{Cnt}_{\langle F,k+1\rangle}$. Similarly, when invoking BoundedSAT for $i = k$ after determining that for $i = k-1$, $\mathsf{Cnt}_{\langle F,k-1\rangle} \geq \mathsf{thresh}$, we first check how many solutions of $F \wedge (h^{k-1})^{-1}(\mathbf{0})$ satisfy $F \wedge (h^k)^{-1}(\mathbf{0})$. As noted above, in expectation, $\mathsf{thresh}/2$ out of thresh solutions of $F \wedge (h^{k-1})^{-1}(\mathbf{0})$ would satisfy $F \wedge (h^k)^{-1}(\mathbf{0})$.

5.2 ApproxMC4 and UniGen3

That said, we turn our focus to hashing-based sampling and counting techniques to showcase the impact of BIRD2. To this end, we integrate BIRD2 along with the proposed technique in Sect. 5.1 into the state of the art hashing-based counting and sampling tools: ApproxMC3 and UniGen2 respectively. We call our improved counting tool ApproxMC4 and our improved sampling tool UniGen3.

Assurance of Correctness. We believe it to be imperative to strongly verify correctness and quality of results provided by our tools, as it is not only possible but indeed easy to accidentally generate incorrect or low quality results, as demonstrated by Chakraborty and Meel [5]. To ensure the quality and correctness of our sampler and counter, we used three methods: (1) fuzzed the system as first demonstrated in SAT by Brummayer et al. [3], (2) compared the approximate counts returned by ApproxMC4 with the counts computed by a known good exact model counter as previously performed by Soos and Meel [19], and (3) compared the distribution of samples generated by UniGen4 on an example problem against that of a known good uniform sampler as previously performed by Chakraborty et al. [9]. We focus on (1), i.e. fuzzing, here and defer the discussion about (2) and (3) to the next section.

Fuzzing is a technique [17] used to find bugs in code by generating random inputs and observing crashes, invariant check fails, and other errors from the output of the system under test. CryptoMiniSat has such a built-in fuzzer generating random CNFs and verifying the output of the solver. To account for XOR constraints, we improved the built-in fuzzer of CryptoMiniSat by adding a counting- and sampling-specific XOR-CNF generator. This inserts randomly generated XORs that form distinct matrices inside the generated CNFs and adds a randomly generated sampling set over some of these matrices. We also added hundreds of lines of invariant checks to our improved Gauss-Jordan elimination algorithm, running throughout our fuzzing tests. Running this improved fuzzer for many hundreds of CPU hours has greatly helped debugging and gaining confidence in our implementation.

6 Evaluation

To evaluate the performance and quality of approximations and samples computed by ApproxMC4 and UniGen3, we conducted a comprehensive study involving 1896 benchmarks as released by Soos and Meel [16] comprising a wide range of application areas including probabilistic reasoning, plan recognition, DQMR networks, ISCAS89 combinatorial circuits, quantified information flow, program synthesis, functional synthesis, logistics, and the like.

In the context of counting, we focused on a comparison of the performance of ApproxMC4 vis-a-vis ApproxMC3. In the context of sampling, a simple methodology would have been a comparison of UniGen3 vis-a-vis the state of the art sampler, UniGen2. Such a comparison, in our view, would be unfair to UniGen2

as while ApproxMC3 builds on BIRD framework, such is not the case for UniGen2. It is worth noting that the BIRD framework, proposed by Soos and Meel [19], can work as a drop-in replacement for the SAT solver in UniGen2, as it only changes the underlying SAT solver. Therefore, we used UniGen2 augmented with BIRD, called UniGen2+BIRD henceforth, as baseline for performance comparisons in the rest of this paper, as it is significantly faster than UniGen2, and therefore, will lead to a fair comparison and showcase improvements solely due to BIRD2.

To keep in line with prior studies, we set $\varepsilon = 0.8$ and $\delta = 0.8$ for ApproxMC3 and ApproxMC4 respectively. Similarly, we set $\varepsilon = 16$ for both UniGen3 and UniGen2+BIRD respectively. The experiments were conducted on a high performance computer cluster, each node consisting of 2xE5-2690v3 CPUs with 2×12 real cores and 96 GB of RAM. We use a timeout of 5000 s for each experiment, which consisted of running a tool on a particular benchmark.

6.1 Performance

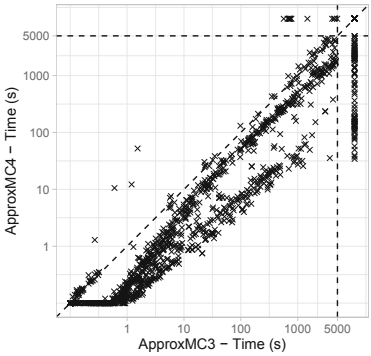


Fig. 1. Comparison of ApproxMC4 and ApproxMC3. ApproxMC4 is faster below the diagonal. Time outs are plotted behind the 5000 s mark.

ApproxMC4 vis-a-vis ApproxMC3. Figure 1 shows a scatter plot comparing ApproxMC4 and ApproxMC3. Although, there are some benchmarks that are solved faster with ApproxMC3 there is a clear trend demonstrating the speed up achieved through our improvements: ApproxMC4 can solve many benchmarks more than 10 times faster and in total solves 77 more instances than ApproxMC3. In particular, ApproxMC3 and ApproxMC4 solved 1148 and 1225 instances respectively, while achieving PAR-2 scores of 4146 and 3701 respectively.

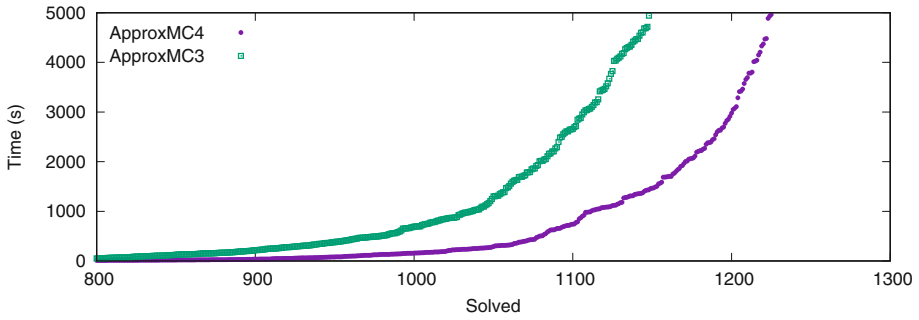


Fig. 2. Cactus plot showing behavior of ApproxMC4 and ApproxMC3

Figure 2 shows the cactus plot for ApproxMC3 and ApproxMC4. We present the number of benchmarks on the x-axis and the time taken on the y-axis. A point (x, y) implies that x benchmarks took less than or equal to y seconds to solve for the corresponding tool.

To present a detailed picture of performance gain achieved by ApproxMC4 over ApproxMC3, we present a runtime comparison of ApproxMC4 vis-a-vis ApproxMC3 in Table 1 on a subset of benchmarks. Column 1 of the table presents benchmarks names, while columns 2 and 3 list the number of variables and clauses. Column 4 and 5 list the runtime (in seconds) of ApproxMC4 and ApproxMC3, respectively.

While investigating the large improvements in performance, we observed that when both the sampling set and the number of solutions is large for a problem, the new system can be up to an order of magnitude faster. In these cases the Gauss-Jordan elimination (GJE) component of the SAT solver dominated the runtime of ApproxMC3 due to the large matrices involved in such problems. The improvements of BIRD2 has led to significant improvement in efficiency of GJE component and we observe that the runtime, in such instance, is now often dominated by the CDCL solver's propagation and conflict clause generation routines.

UniGen3 vis-a-vis UniGen2+BIRD. Similar to Fig. 2, Fig. 3 shows the cactus plot for UniGen3, UniGen2+BIRD, and UniGen2. We present the number of benchmarks on the x-axis and the time taken on the y-axis. UniGen3 and UniGen2+BIRD were able to solve 1012 and 1063 instances, respectively while achieving PAR-2 scores of 4574 and 4878, respectively. UniGen2 could solve only 360 benchmarks, thereby justifying our choice of implementing UniGen2+BIRD as a baseline for fair comparison to showcase strengths of BIRD2. We would like to highlight that the cactus plot shows that given a 2600 s timeout, UniGen can sample as many benchmarks as UniGen2+BIRD would do for a 5000 s timeout.

To present a clear picture of performance gain by UniGen3 over UniGen2+BIRD, we present runtime comparison for UniGen3 vis-a-vis UniGen2+BIRD in Table 1, where in addition to data on ApproxMC3 and

Table 1. Performance comparison of ApproxMC3 vis-a-vis ApproxMC4 and UniGen2+BIRD vis-a-vis UniGen3. TO indicates timeout after 5000 s or out of memory. Notice that on many problems that used to time out even for counting, we can now confidently sample.

Benchmark	Vars	Cls	ApproxMC3 time (s)	ApproxMC4 time (s)	UniGen2+BIRD 500 samples time (s)	UniGen3 500 samples time (s)
or-70-5-1-UC-20	140	350	6.03	2.07	14.21	6.08
prod-4	7497	37358	56.65	7.09	171.57	36.54
min-8	1545	4230	152.53	5.58	471.47	35.04
parity.sk_11_11	13116	47506	389.26	436.32	705.85	809
leader_sync4_11	205198	129149	346.4	20.55	1019.09	106.93
blasted_TR_b12_2	2426	8373	308.08	20.46	1218.01	546.62
hash-8-6	377545	1517574	462.28	266.59	1321.91	633.84
s15850a_15_7	10995	24836	1206.17	31.69	2782.96	230.17
ConcreteRole	395951	1520924	1694.19	309.07	3083.99	923.69
tire-3	577	2004	3059.19	233.28	3876.03	797.42
04B-2	19510	86961	1860.97	625.81	TO	2236.31
blasted_case138	849	2253	TO	3691.9	TO	TO
hash-11-4	518449	2082039	4602.95	4043.4	TO	TO
karatsuba.sk_7_41	19594	82417	3192.85	3410.36	TO	TO
log-3	1413	29487	TO	123.15	TO	408.25
modexp8-8-6	167793	633614	4439.21	TO	TO	TO
or-100-5-6-UC-20	200	500	TO	1689.47	TO	4898.43
prod-28	52233	261422	TO	235.02	TO	1053.9
s38417_15_7	25615	57946	TO	187.71	TO	TO
signedAvg	30335	91854	TO	114.15	TO	582.01

ApproxMC4, columns 5 and 6 lists the runtime for UniGen3 and UniGen2+BIRD respectively. Similar to the observation above, we note that UniGen3 is able to sample for instances that timed out even for ApproxMC3. It is worth to recall that UniGen3 (and UniGen2) first makes a call to an approximate counter during its parameter search phase.

Remark 1. Since the runtime improvements of ApproxMC4 and UniGen3 are primarily due to improvements in the underlying SAT solver, it is worth pointing out, to put our contribution in context, that the difference between average PAR-2 scores of the top two solvers in a SAT competition is usually less than 100.

6.2 Quality and Correctness

Quality of Counting. To evaluate the quality of approximation we follow the same approach as Soos and Meel [19] and compare the approximate counts returned by ApproxMC4 with the counts computed by an exact model counter, namely DSharp[4]. The approximate counts and the exact counts are used to compute the observed tolerance ε_{obs}, which is defined as $\max(\frac{|sol(F)_{\downarrow S}|}{\text{AprxCount}} - 1, \frac{\text{AprxCount}}{|sol(F)_{\downarrow S}|} - 1)$, where AprxCount is the estimate computed by ApproxMC4 for a formula F and a sampling set S, which are both given for each benchmark. Note that, using ε_{obs}, we can rewrite the theoretical (ε, δ)-guarantee to $\mathsf{Pr}[\varepsilon_{obs} \leq \varepsilon] \geq 1 - \delta$ and hence we expect that ε_{obs} is mostly below $\varepsilon = 0.8$. The observed tolerance ε_{obs} over all benchmarks is shown in Fig. 4. We observe a maximal value for ε_{obs} of 0.3333 and the the arithmetic mean of ε_{obs} across all benchmarks is 0.0411. Hence, the approximate counts are much closer to the exact counts than is theoretically guaranteed.

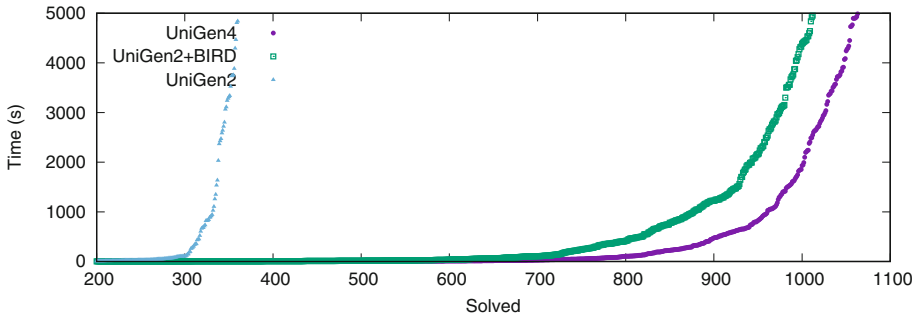


Fig. 3. Sampling performance of UniGen2 and UniGen2+BIRD versus UniGen3.

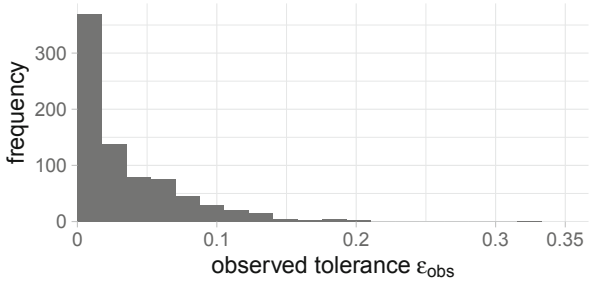


Fig. 4. The histogram of the observed tolerance ε_{obs} shows that the approximate counts are very close to the exact counts.

[4] DSharp is used because of its ability to handle sampling sets.

Quality of Sampling. To evaluate the quality of sampling, we employed the uniformity tester, Barbarik, proposed by Chakraborty and Meel [5]. To this end, we selected 35 benchmarks from the pool of benchmarks employed by Chakraborty and Meel in their work and we tested UniGen3 for all the 35 benchmarks. We observed that Barbarik accepts UniGen3 for all the 35 instances, thereby providing a certificate for uniformity. We refer the reader to [5] for detailed discussion of the guarantees provided by Barbarik. Keeping in line with past work on sampling that tries to demonstrate the quality of sampling on a representative benchmark where exact uniform sampling is feasible via enumeration-based techniques, we chose the CNF instance *blasted_case110* (287 variables and 16384 solutions), which has been chosen in the previous studies as well. To this end, we implemented a simple ideal uniform sampler, denoted by US henceforth, by enumerating all the solutions and then picking a solution uniformly at random. We then generate $4,039,266$ samples from both UniGen3 and US. In each case, the number of times various witnesses were generated was recorded, yielding a distribution of the counts. Fig. 5 shows the distributions of counts generated versus # of solutions. The x-axis represents counts and the y-axis represents the number of witnesses appearing the specified number of times. Thus, the point (230,212) represents the fact that each of 212 distinct witnesses were generated 230 times among the $4,039,266$ samples. While UniGen3 provides guarantees of almost-uniformity only, the two distributions are statistically indistinguishable. In particular, the KL divergence [15] of the distribution by UniGen from that of US is 0.003989.

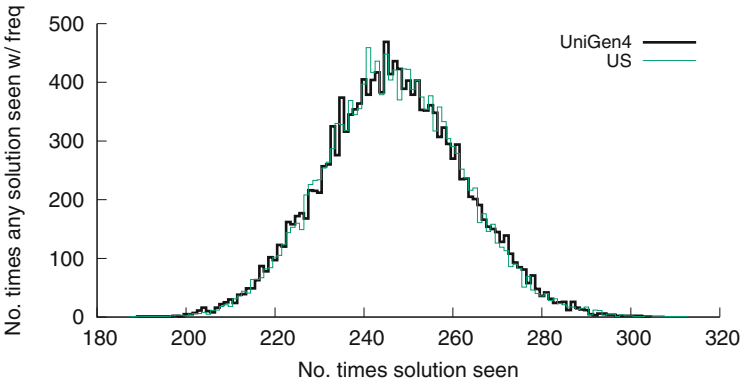


Fig. 5. Distribution of solution recurrence as generated by UniGen3 and US for the CNF `blasted_case110.cnf`.

7 Conclusions

We investigated the bottlenecks of CNF-XOR solving in the context of hashing-based approximate model counting and almost uniform sampling as implemented

in ApproxMC3 and UniGen2 respectively. In this paper, we proposed five technical improvements, as follows: (1) detaching the clausal representation of XOR constraints from unit propagation, (2) lazy reason generation for XOR constraints, (3) bit-level parallelism for XOR constraint propagation, (4) partial solution extraction only covering the sampling set and (5) solution reuse. These improvements were incorporated into the new framework BIRD2, which led to the construction of improved approximate model counter ApproxMC4 and almost uniform sampler UniGen3. Experiments over a large set of benchmarks from various domains clearly show an improvement in running time and 77 more problems could be solved for counting and 51 more for sampling.

Acknowledgments. We are grateful to Yash Pote for the early discussions of solution reuse. Work done in part while the second author visited NUS.

This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme[NRF-NRFFAI1-2019-0004] and AI Singapore Programme [AISG-RP-2018-005], and NUS ODPRT Grant [R-252-000-685-13]. The second author was funded by the Swedish Research Council (VR) grant 2016-00782. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore https://www.nscc.sg.

References

1. Baluta, T., Shen, S., Shinde, S., Meel, K.S., Saxena, P.: Quantitative verification of neural networks and its security applications. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, pp. 1249–1264 (2019)
2. Bellare, M., Goldreich, O., Petrank, E.: Uniform generation of NP-witnesses using an NP-oracle. Inform. Comput. **163**(2), 510–526 (2000)
3. Brummayer, R., Lonsing, F., Biere, A.: Automated testing and debugging of SAT and QBF solvers. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 44–57. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14186-7_6
4. Carter, J.L., Wegman, M.N.: Universal classes of hash functions. In: ACM Symposium on Theory of Computing, pp. 106–112. ACM (1977)
5. Chakraborty, S., Meel, K.S.: On testing of uniform samplers. In: Proceedings of AAAI Conference on Artificial Intelligence (AAAI), January 2019
6. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: On parallel scalable uniform sat witness generation. In: Proceedings of TACAS, pp. 304–319 (2015)
7. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable and nearly uniform generator of SAT witnesses. In: Proceedings of CAV, pp. 608–623 (2013)
8. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable approximate model counter. In: Proceedings of CP, pp. 200–216 (2013)
9. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Balancing scalability and uniformity in SAT witness generator. In: Proceedings of DAC, pp. 1–6 (2014)
10. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Algorithmic improvements in approximate counting for probabilistic inference: from linear to logarithmic SAT calls. In: Proceedings of IJCAI (2016)

11. Duenas-Osorio, L., Meel, K.S., Paredes, R., Vardi, M.Y.: Counting-based reliability estimation for power-transmission grids. In: Proceedings of AAAI (2017)
12. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) Proceedings of SAT, pp. 61–75 (2005)
13. Han, C.-S., Jiang, J.-H.R.: When boolean satisfiability meets Gaussian elimination in a simplex way. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 410–426. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_31
14. Jerrum, M.R., Valiant, L.G., Vazirani, V.V.: Random generation of combinatorial structures from a uniform distribution. Theor. Comput. Sci. **43**(2–3), 169–188 (1986)
15. Kullback, S., Leibler, R.A.: On information and sufficiency. Ann. Math. Stat. **22**(1), 79–86 (1951)
16. Meel, K.S.: Model counting and uniform sampling instances (May 2020). https://doi.org/10.5281/zenodo.3793090
17. Miller, B.P., et al.: Fuzz revisited: a re-examination of the reliability of UNIX utilities and services. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences (1995)
18. Roth, D.: On the hardness of approximate reasoning. Artif. Intell. **82**(1), 273–302 (1996)
19. Soos, M., Meel, K.S.: BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In: AAAI Conference on Artificial Intelligence, AAAI, pp. 1592–1599. AAAI Press (2019)
20. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Proceedings of SAT (2009)
21. Stockmeyer, L.: The complexity of approximate counting. In: Proceedings of STOC, pp. 118–126 (1983)
22. Toda, S.: On the computational power of PP and (+)P. In: Proceedings of FOCS, pp. 514–519. IEEE (1989)
23. Vadhan, S.P., et al.: Pseudorandomness. Found. Trends Theor. Comput. Sci. **7**(1–3), 1–336 (2012)
24. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM J. Comput. **8**(3), 410–421 (1979)

Automated and Scalable Verification of Integer Multipliers

Mertcan Temel[1,2(✉)], Anna Slobodova[2], and Warren A. Hunt Jr.[1]

[1] University of Texas at Austin, Austin, TX, USA
{mert,hunt}@cs.utexas.edu
[2] Centaur Technology, Inc., Austin, TX, USA
anna@centtech.com

Abstract. The automatic formal verification of multiplier designs has been pursued since the introduction of BDDs. We present a new rewriter-based method for efficient and automatic verification of signed and unsigned integer multiplier designs. We have proved the soundness of this method using the ACL2 theorem prover, and we can verify integer multiplier designs with various architectures automatically, including Wallace, Dadda, and 4-to-2 compressor trees, designed with Booth encoding and various types of final stage adders. Our experiments have shown that our approach scales well in terms of time and memory. With our method, we can confirm the correctness of 1024×1024-bit multiplier designs within minutes.

Keywords: Multipliers · Hardware verification · Formal methods · ACL2

1 Introduction

Arithmetic circuit designs may contain bugs that may not be detected through random testing. Since the Pentium FDIV bug [29], formal verification has become more prominent for validating the correctness of arithmetic circuits. Despite being a crucial part of all processors, verifying the correctness of arithmetic circuits, specifically multipliers, is still an ongoing challenge.

There have been numerous efforts to find a scalable and automated method to formally verify integer multipliers. Early methods that were based on attempts to represent hardware and its specification in various canonical forms - BDDs [6] and derivatives, have an exponential space complexity. Therefore, they were applicable only for small circuits. Similarly, SAT-based methods did not prove to be scalable [28].

There are several approaches for the verification of hardware multipliers used in the industry. One is based on writing a simple RTL multiplier design without optimizations and comparing it to the candidate multiplier design through equivalence checking [14,35]. This approach works only when the reference design is structurally close to the original under verification and relies on the correctness

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 485–507, 2020.
https://doi.org/10.1007/978-3-030-53288-8_23

of the reference design and proof maintenance whenever designers make structural changes. Another approach is to find a suitable decomposition of a design into parts that can be verified automatically and compose those results into a top-level theorem [13,15,30]. The drawback of this method is that it requires manual intervention by the verification engineer who decides about the boundaries of the decomposition. A third approach involves guiding a mechanized proof checker manually [27].

In recent years, the search for more automatic procedures resulted in methods based on symbolic computational algebra [7,16,22,23,40] . This approach makes it possible for certain types of multipliers to be verified automatically for larger designs. However, they have limitations as to what type of multipliers they can check (see experiments in Sect. 6). They are implemented as unverified programs and, as far as we are aware, only one of them [16] produces certificates.

We have developed an automatic rewriter-based method for verification of hardware integer multipliers that is

- widely applicable,
- provably correct, and
- scalable

We implemented and verified our method with the ACL2 theorem proving system, which is a subset of the LISP programming language. Our method is not ACL2 specific and can be adapted to other platforms with suitable adjustments. In this paper, we also provide proof of its termination. Even though we have not proved the completeness of this method, our tool can verify various multiplier designs. We test our method on designs implemented with (System) Verilog where design hierarchy is maintained. We can verify various types of multipliers in a favorable time; for example, we tested our method with 8 different types of 1024×1024 multipliers and verified each of them in less than 10 min, while the other state-of-the-art tools ran for more than 3 h.

The paper is structured as follows. In Sect. 2, we present some concepts that might be necessary to understand our approach. These include the basic notion of term rewriting and the ACL2 system (Sect. 2.1), the semantics for hardware modeling (Sect. 2.2), and some basic multiplier architectures (Sect. 2.3). Preliminaries are followed by our specification and top-level correctness theorem for multiplier designs (Sect. 3). We explain our methodology to prove this top-level correctness theorem with term rewriting in Sect. 4. Section 5 describes the termination of our rewriting algorithm. Experiments with various benchmarks are given and discussed in Sect. 6.

2 Preliminaries

In this section, we describe the concepts and tools required to understand the method proposed in this paper. We review the ACL2 theorem prover and term rewriting, how Verilog designs are translated and used in proofs, and various integer multiplier architectures.

2.1 ACL2 and Term Rewriting

ACL2 is a LISP-based interactive theorem prover that can be used to model computer systems and prove properties about such models using both its internal procedures as well as appealing to external tools such as SAT and SMT solvers. ACL2 is used by the industry for both software and hardware verification [12]. Our methodology to prove multipliers correct uses ACL2-based term rewriting.

ACL2 can store proved lemmas as *rewrite rules*, and later use them when attempting to confirm other conjectures. ACL2 terms are prefix expressions and rewriting is attempted on terms such as `(fnc arg1 arg2 ...)`. Left-hand side of a rewrite rule is unified with terms; in case of a successful unification, the matched term is replaced by a properly instantiated right-hand side if all hypotheses are satisfied. Example 1 shows two rewrite rules, the second of which can be proved using the first as a lemma. When users submit a `defthm` event, ACL2 attempts to confirm the conjecture by rewriting it in an inside-out manner. For the conjecture given in `x-x_y-y`, the rewriter replaces `(+ x (- x))` and `(+ y (- y))` with `0` using `a-a` as a lemma. Then the resulting term `(+ 0 0)` is replaced with `0` using the executable counterpart of the function `+`.

Example 1. A simple rewrite rule `a-a`, and a theorem `x-x_y-y` proved subsequently using `a-a` as a lemma.

```
(defthm a-a
  (implies (integerp a)
           (equal (+ a (- a)) 0)))
(defthm x-x_y-y
  (implies (and (integerp x) (integerp y))
           (equal (+ (+ x (- x)) (+ y (- y)))
                  0)))
```

The rewriting mechanism in ACL2 is much more complex and intricate than we indicate here [18]. Throughout the rest of this paper, we omit ACL2 specific implementation details whenever possible. Understanding the basics of term rewriting is sufficient to follow our methodology.

2.2 Semantics for Hardware Designs

We convert (System) Verilog designs to *SVL* netlists in ACL2 and use SVL functions for semantics and simulation of circuit designs [33]. SVL netlists preserve hierarchical information about hardware designs and they are based on the *SV* [31] and *VL* [32] tools that are also included in the ACL2 libraries. These tools have been used by several companies to confirm the correctness of various circuit designs [12]. In this section, we describe the format of SVL netlists, and how they are simulated hierarchically.

An SVL netlist is an association list where each key is a module name, and its corresponding value is the definition of the module. An SVL module is composed of input and output signals, and a list of occurrences. An occurrence can be an

assignment or an instantiation of another module. Example 2 shows a simplified SVL netlist containing a half and a full-adder.

Example 2. An *SVL* netlist for half and full-adder.

```
(("ha" (inputs  x y)
       (outputs s c)
       (occs ((occ1 :assign s (bitxor x y))
              (occ2 :assign c (bitand x y)))))
 ("fa" (inputs  x y z)
       (outputs s c)
       (occs ((occ1 :module "ha" (ins x y)  (outs t1 t2))
              (occ2 :module "ha" (ins t1 z) (outs s t3))
              (occ3 :assign c    (bitor t2 t3))))))
```

The semantics of an SVL netlist is given by a recursively defined ACL2 function, `svl-run`. This function traverses occurrences of a module and simulates them in order by evaluating the assignments and making a recursive call for the submodules. After each occurrence, the values of wires/signals are stored in an association list, and when finished, `svl-run` retrieves and returns the values of output signals from this association list. These values can be concrete (**`svl-run`** is executed), or symbolic (the rewriter processes a call of **`svl-run`** with variables for inputs), which can create ACL2 expressions representing the functionality of the design for each output. For example, we can generate expressions for the outputs of the full-adder (**`"fa"`**) in Example 2: `(⊕ x y z)` and `(∨ (∧ x y) (∧ (⊕ x y) z))`. Alternatively, since the design retains hierarchy, submodules can be replaced by their specification. For example, assume that we have specification functions `s-ha` and `c-ha` for each output of the half-adder (**`"ha"`**), and we proved a rewrite rule to replace calls of **`svl-run`** of **`"ha"`** with these functions. If we rewrite the instantiations of **`"ha"`** with this rule while expanding the definition of **`"fa"`**, we can instead get `(s-ha (s-ha x y) z)` and `(∨ (c-ha x y) (c-ha (s-ha x y) z))` for each output of **`"fa"`**.

2.3 Multiplier Architectures

In this section, we discuss the most commonly used algorithms to implement integer multipliers. We summarize partial-product *generation* algorithms, such as Booth encoding, and partial-product *summation* algorithms, such as Wallace-tree. Even though the applicability of our verification method is not confined to a specific set of algorithms, reviewing them is beneficial for understanding the verification problem.

We can divide multiplier designs into two main components: partial product generation and summation. Figure 1a shows these two steps on multiplication of two 3-bit two's-complement signed integers. We perform sign-extension (for signed numbers) or zero-extension (for unsigned numbers) on inputs, generate partial products, and then add them together to obtain the multiplication result

in a fashion similar to grade-school multiplication. The integer multipliers we have verified implement various partial-product generation and summation algorithms for the same functionality with optimizations for better gate-delay and/or area.

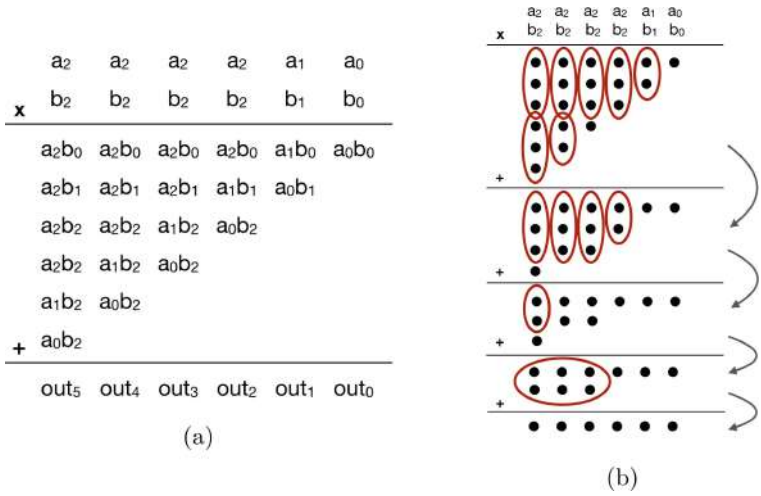


Fig. 1. (a) Grade-school-like multiplication for two 3-bit two's-complement integers, and (b) a Wallace-tree-like multiplier performing bit-level additions on the partial products

Baugh-Wooley [1] and Booth [2] are commonly used algorithms to generate partial products. Baugh-Wooley is used for signed multiplication, and it generates partial products as shown in Fig. 1a, but with a sign-extension algorithm to prevent the repetition of generated partial product bits. A more commonly used alternative is Booth encoding, which can be used for both signed and unsigned multiplication. Instead of simply multiplying all the single bits of the two inputs with each other, Booth encoding uses more than one bit at a time from one of the operands, and it derives a more complex form for partial products. This helps reduce the number of rows for partial products, thus helping shrink the summation circuitry and allowing more parallelism. Booth encoding can be implemented with different radices, which determine the number of multiplier bits used at a time to create partial products (e.g., Booth radix-4 [21] uses 3 bits at a time). The higher the radix, the fewer the partial products; however, higher radices yield a more complex design. Booth encoding can be combined with sign-extension algorithms [38] to prevent repetition in generated partial products.

A rudimentary way to sum partial products is by using a shift-and-add algorithm. One may use an accumulator and a vector adder such as a ripple-carry adder to shift and add partial products. An array multiplier is a variation of this algorithm and it is implemented using a very similar principle with some additional optimizations. Due to their regular structure, verifying the correctness of

these multipliers has not been a challenging problem [5]. However, these circuits often have very large gate delays, and Wallace-tree like multipliers are preferred over these algorithms in industrial applications.

A family of partial product summation algorithms, which are often called Wallace-tree like multipliers [36], use parallelism to obtain multiplication results with less gate-delay but produce a very irregular and complex design structure. Figure 1b shows an example of a Wallace-tree algorithm. In the first summation layer, we see the generated partial products corresponding to the ones in Fig. 1a. The Wallace-tree algorithm selects groups of bits from these partial products and passes them to full and half-adders. After these parallel bit-level additions, resulting carry and sum output bits are replaced on another layer whose summation will also yield the multiplication result. At each stage, layers are compressed, and the number of rows decreases. We repeat this process until we reach a state where we have only two rows. Then, instead of using full and half adders to finish additions, a vector adder (final stage adder), such as carry-lookahead and parallel prefix adders, is used. This method may provide a significant delay reduction over array multipliers. There exist numerous variations of Wallace-tree multipliers such as Dadda-tree [8] and 4-to-2 compressor trees [11]. Due to their highly irregular structure, reasoning about Wallace-tree like multipliers is a difficult problem, especially when combined with complex partial product generation algorithms such as Booth encoding. There is a lot of room for circuit designers to deviate from text-book algorithm definitions when creating multipliers, which increases the importance of having an automated method to verify these circuits with minimal assumptions about the structure.

3 Specification

We aim to prove the functional correctness of signed and unsigned multiplier designs. We do that by proving an ACL2 theorem demonstrating the equivalence of semantics of a multiplier circuit design to the built-in ACL2 multiplication function (*) with appropriate sign extensions and truncations.

We work with integer multiplier circuits that are designed to multiply two numbers (signed or unsigned) stored in bit-vectors and cut (truncate) the resulting number to return it as a bit-vector. If we are multiplying m-bit and n-bit numbers, then the first $m+n$ bits of the result is sufficient to represent all output values. For example, assume that we are multiplying signed numbers –4 and 3, represented with 4-bit vector `1100` and 3-bit vector `011`, respectively. Then, a correct multiplier would return the 7-bit vector `1111100`, which represents -12.

Listing 1.1 shows the final ACL2 theorem we prove for signed integer multipliers, where a and b are variables and $*m*$ and $*n*$ are concrete values[1]. This theorem states that for all integers a and b, simulating an m-by-n signed multiplier circuit returns a value that is equivalent to multiplication of sign-extended a and b, truncated at $m + n$ bits. On the left-hand side, `*signed_mxn_mult*`

[1] By convention, "$*$" characters surrounding variables, such as `*m*`, signify constants in ACL2.

is an ACL2 constant that contains the multiplier design in SVL format which is translated from (System) Verilog, and `svl-run` is the function to simulate this module with inputs a and b. On the right-hand side, $*$ is the built-in integer multiplication function, `truncate` returns first $m+n$ bits of the result, and `signext` returns a number that represents the sign-extended value of a bit-vector. Multiplier designs are implemented with fixed values of m and n; therefore, we prove such theorems for constants m and n and variables a and b. The ACL2 theorem for unsigned multiplication has the same form but in the place of `signext`, we use the `truncate` function, which performs zero-extension. The actual statement of the theorem contains more components than shown, including function calls to extract outputs and parameters for state-holding elements; we only give the essentials for brevity.

Listing 1.1. The Final Correctness Theorem for Signed Multipliers

```
(defthm multiplier_is_correct
  (implies (and (integerp a)
                (integerp b))
           (equal (svl_run (list a b) *signed_mxn_mult*)
                  (truncate (+ *m* *n*)
                            (* (signext *m* a)
                               (signext *n* b))))))
```

4 Methodology

The correctness theorem given in Listing 1.1 is proved by rewriting both sides of the equality to two syntactically equivalent terms. In this section, we describe our methodology to rewrite both sides to a specific form through an automated rewriting mechanism.

We have a targeted final expression for each output bit of a multiplier design, the mathematical formula of which is given in Definition 2. The variables a and b are the inputs/operands of multiplication with a certain size (e.g., 64 bits for 64×64 multiplication); and in this formula, they are sign-extended for two's complement signed multiplication or zero-extended for unsigned multiplication.

Definition 1. *We define functions s and c as follows.*

$$\forall x \in \mathbb{Z}\ s(x) = mod_2(x)$$
$$\forall x \in \mathbb{Z}\ c(x) = \left\lfloor \frac{x}{2} \right\rfloor$$

Definition 2. *The targeted form for each output bit (out_j) is defined as follows.*

$$w_j = \begin{cases} (\sum_{i=0}^{j} a_i b_{j-i}) + c(w_{j-1}) & \text{if } j \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$
$$out_j = s(w_j)$$

where $a_i b_{j-i}$ is logical AND of the ith and $(j-i)$th bits of operands a and b.

Table 1 shows an example of this targeted final form for the first four output bits of 3×3 two's complement signed multiplication (see Fig. 1a). Each output bit is represented with expressions composed of the s, c, and $+$ functions. In this representation, the outermost function of each expression is s, carry bits from previous columns are calculated with a single c per column, and the terms in summations are sorted lexicographically. Two's complement signed or unsigned integer multiplication implemented by our candidate designs (See Sect. 6) can be represented by an expression of this form.

Table 1. Expressions for the final form of the first four output bits from Fig. 1a

out_3	out_2	out_1	out_0
$s(a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$ $+c(a_0b_2 + a_1b_1 + a_2b_0$ $+c(a_1b_0 + a_0b_1$ $+c(a_0b_0)))$	$s(a_0b_2 + a_1b_1 + a_2b_0$ $+c(a_1b_0 + a_0b_1$ $+c(a_0b_0)))$	$s(a_1b_0 + a_0b_1$ $+c(a_0b_0))$	$s(a_0b_0)$

A summary of our rewriting approach to verify multiplier designs is given in Fig. 2. Our method works with design semantics such as SVL where circuit hierarchy can be maintained and we reason about adder modules and the main multiplier module at different stages. As the first step, we work only with adder modules (e.g., half/full-adders and final stage adders) instantiated as submodules by the candidate multiplier design. We state a conjecture similar to Listing 1.1 for each adder module. We simplify their gate-level circuit description and prove them equivalent to their specification. We save these proofs as rewrite rules where lhs is `svl-run` of adder module and rhs is its specification. Having created these rewrite rules for all the adder modules, we start working on the correctness proof of the multiplier design as stated in Listing 1.1. On the LHS, as we derive ACL2 expressions from the definition of multiplier designs (see Sect. 2.2), we replace instantiated adder modules with their specification, and we apply two other sets of rewrite rules to simplify summation tree and partial product logic. On the RHS, we rewrite the multiplier specification into the targeted final form of multiplication, and we syntactically compare the two resulting terms to conclude our multiplier design proofs.

We simplify adder and multiplier modules by stating a set of lemmas in the form of equality $lhs = rhs$. These lemmas are used to create a term rewriting mechanism where expressions from circuit definitions are unified with lhs and replaced with their corresponding rhs. We aim to provide a set of lemmas so that such an automated system of rewriting can reduce a wide range of multiplier circuit designs to the final form as given in Table 1. In pursuit of this goal, we devised and experimented with various rewriting strategies; and we came up with a well-performing heuristic. In the subsections below, we describe these

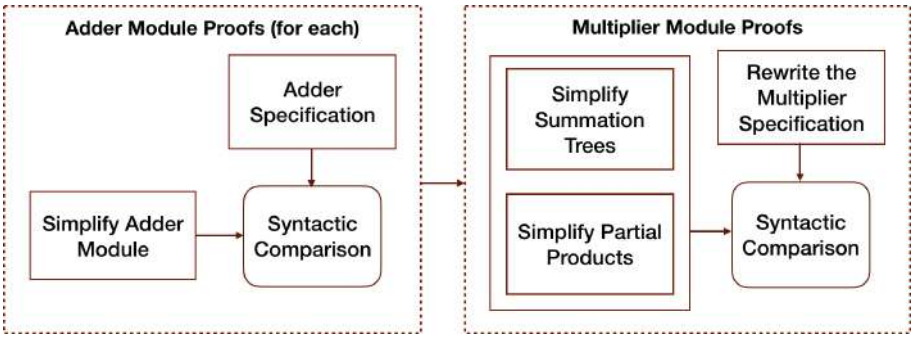


Fig. 2. Summary of the overall method

lemmas separated into two main sets for adder and multiplier modules, and the general mechanism to prove them equivalent to their specification. The lemmas we introduce are proved using ACL2, and we omit the proofs for brevity.

4.1 Adder Module Proofs

The first step of our rewriting strategy is to represent the outputs of adder modules in terms of the s, c, and $+$ functions. We first determine the modules that serve as adder components in multiplier designs, such as half-adders, full-adders, 4-to-2-compressors, and final stage adders. Then we state a conjecture similar to Listing 1.1 where *lhs* is `svl-run` of the adder module and *rhs* is its specification. We prove this conjecture with a library of rewrite rules, derived from the lemmas given in this section, which can simplify various types of adder modules and prove them equivalent to their specification.

For vector adders, specifications have a fixed format as shown in Table 2; however, for single-bit adders, such as full-adders and 4-to-2 compressors, specifications may vary. The format of these specifications can be of any form as long as they are composed of only the s, c, and $+$ functions as given in Table 2. For adders that are not given in this table (e.g., 4:2 compressors), users may derive their specifications by simplifying them with the lemmas introduced below.

We expect adder modules to be composed of logical AND ($\wedge$), OR ($\vee$), XOR ($\oplus$), and NOT ($\neg$) gates in certain patterns. We get expressions for these circuits' functionality in terms of these functions through SVL semantics. We rewrite these expressions with the lemmas given below to simplify them to the same form as their specification. We define the operators $\wedge$ (and), $\vee$ (or), $\oplus$ (exclusive or), and $\neg$ (negation) to work with integer-valued bits (e.g., $1 \wedge 0 = 0$, $1 \vee 1 = 1$, or $0 \oplus 1 = 1$).

Lemma 1. $\forall x, y \in \{0,1\}\ x \oplus y = s(x + y)$

Lemma 2. $\forall x, y \in \{0,1\}\ x \wedge y = c(x + y)$

Lemma 3. $\forall x, y, h, g \in \{0,1\}\ c(x + y + h) \vee (s(x + y) \wedge g) = c(x + y + (h \vee g))$

Table 2. Rewritten outputs of some adders

Adder	out_3	out_2	out_1	out_0
Half-adder	–	–	$c(a_0 + a_1)$	$s(a_0 + a_1)$
Full-adder	–	–	$c(a_0 + a_1 + a_2)$	$s(a_0 + a_1 + a_2)$
Vector adders	$s(a_3 + b_3$ $+c(a_2 + b_2$ $+c(a_1 + b_1$ $+c(a_0 + b_0)))$	$s(a_2 + b_2$ $+c(a_1 + b_1$ $+c(a_0 + b_0)))$	$s(a_1 + b_1$ $+c(a_0 + b_0))$	$s(a_0 + b_0)$

We implement these lemmas as well as some corollaries as rewrite rules so that terms that can be unified with the *lhs* of equations are replaced by their respective *rhs*. An example corollary is $\forall x, y, g \in \{0, 1\}\ (x \wedge y) \vee (s(x + y) \wedge g) = c(x\ +\ y\ +\ g)$ that can be derived from Lemmas 2 and 3. Similarly, $\forall x, y, h \in \{0, 1\}\ c(x+y+h) \vee s(x+y) = c(x+y+1)$ can be derived from Lemma 3. These extra lemmas help expand our coverage to match more term patterns that may occur.

We add other rewrite rules using elementary properties of $\vee$, $\wedge$ and $+$ that help facilitate simplification. Lemma 3, and some corollaries rewrite terms with repeated variables. In such cases, in order for the rewriter to match the *lhs* with an applicable term, it is necessary to flatten the terms with associativity (e.g., $((a + b) + c) = (a + b + c)$) and lexicographically sort them using commutativity (e.g., $(b\ +\ a)\ =\ (a\ +\ b)$) for every $+$, $\vee$ and $\wedge$ instance. Other examples of rewrite rules we have in our system implement identity and inverse properties of addition. Finally, we have a lemma that rewrites the definition of $\oplus$, which is $(\neg ab \vee a\neg b)$, in terms of s as given in Lemma 1.

Note that we put a restriction on the use of the rewrite rule for Lemma 2 such that it is used only when x and y are input wires of the adder module. The function c is a specification for carry, and not all AND gates may calculate carry by themselves. We have observed that only the logical AND of input signals should be rewritten to c. Rewriting the other instances of $\wedge$ in terms of c prevents application of Lemma 3 and complicates our rewriting approach. We enforce this restriction in ACL2 through a syntactic check.

Our experiments given in Sect. 6 demonstrate that the method we described in this section can automatically simplify vector adders including ripple-carry, carry-lookahead [26] and parallel-prefix adders such as Brent-Kung [4], Ladner-Fischer [20], Kogge-Stone [19], Han-Carlson [9] and others.

Reasoning about adder modules before the candidate multiplier module is a crucial step in our rewriting mechanism. The functionality of all the adder modules should be represented with the s, c, and $+$ functions when expanding the definition of the multiplier module. Then, and only then, the multiplier design can be simplified and proved correct with the lemmas described in the subsequent section.

4.2 Multiplier Module Proofs

After creating rewrite rules for adder modules, we start working with the correctness proof of our candidate multiplier design as given in Listing 1.1. Similarly, we convert multiplier modules into ACL2 expressions, replace instantiated adder modules with their specifications, and perform simplification with a rewriting mechanism derived from the lemmas introduced in this section. We first describe how we simplify complex expressions that originate from summation tree algorithms such as Wallace-tree. Secondly, we add more lemmas to simplify partial product logic that may be generated with Booth encoding. After rewriting with these lemmas, we expect to have simplified multiplier designs to our targeted final form as given in Table 1. We rewrite the multiplication specification into our final form as well and conclude verification with a syntactic equivalence check.

Simplify Summation Trees. In some integer multiplier designs, summation of partial products may be implemented with a very irregular structure, as is the case with Wallace-tree like multipliers (see Sect. 2.3), and it can be challenging to simplify them to a regular and more easily interpretable form. We describe a set of lemmas, solving this problem by providing an efficient and automated mechanism for such complex structures. Below, we discuss the simplification method for multiplier designs implemented with simple partial products.

Having rewritten the adder components in terms of the s, c, and $+$ functions, Example 3 shows the term representing the 4th LSB of a Wallace-tree multiplier output. Our goal is to reduce such terms to our final form as given in Table 1.

Example 3. The 4th LSB of the Wallace-tree multiplier output from Fig. 1b after adder submodules are rewritten in terms of the s, c and $+$ functions:

$$\begin{aligned} &s(\ s(\ s(a_3b_0 + a_2b_1 + a_1b_2) \\ &\quad\ +a_0b_3 \\ &\quad\ +c(a_2b_0 + a_1b_1 + a_0b_2)) \\ &\ \ +c(s(a_2b_0 + a_1b_1 + a_0b_2) + c(a_1b_0 + a_0b_1))) \end{aligned}$$

In such summation trees, we observe many nested calls for s. These can be simplified easily by the following rule.

Lemma 4. $\forall x, y \in \mathbb{Z}\ s(s(x) + y) = s(x + y)$

Example 4. Example 3 simplified with Lemma 4:

$$\begin{aligned} &s(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\ &\ \ +c(a_2b_0 + a_1b_1 + a_0b_2) \\ &\ \ +c(s(a_2b_0 + a_1b_1 + a_0b_2) + c(a_1b_0 + a_0b_1))) \end{aligned}$$

Terms derived from summation trees may include many instances for addition of two or more calls of c. Since such instances are not present in the final form, we try to remove them. That can be done by merging such calls of c through a temporary conversion to d as implemented with the lemmas given below.

Definition 3. *We define function d as follows.*

$$\forall x \in \mathbb{Z}\ d(x) = \frac{x}{2}$$

Lemma 5. $\forall x, y \in \mathbb{Z}\ c(x) + c(y) = d(x + y - s(x) - s(y))$

Lemma 6. $\forall x, y \in \mathbb{Z}\ c(x) + d(y) = d(x + y - s(x))$

Lemma 7. $\forall x, y \in \mathbb{Z}\ d(x) + d(y) = d(x + y)$

Lemma 8. $\forall x \in \mathbb{Z}\ d(-s(x) + x) = c(x)$

Applying Lemmas 5, 6, 7, and 8 repeatedly to the term in Example 4, we obtain the term given in Example 5. Since $\forall a, b \in \{0, 1\}\ c(a \wedge b) = 0$, we have a term that matches the 4th bit of the final form for multiplication as given in Table 1. It is not required to convert certain instances of d back to c with Lemma 8; however, we can achieve better proof-time performance by shrinking terms with this rewrite.

Example 5. Example 4 simplified with Lemma 5, 6, 7, 8:

$$\begin{aligned} &s(a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 \\ &\quad + c(a_2b_0 + a_1b_1 + a_0b_2 \\ &\qquad + c(a_1b_0 + a_0b_1))) \end{aligned}$$

Rewriting with Lemmas 5 and 6 creates new instances of s, which may not seem preferable at first glance because terms become less similar to the final form. However, we have found that for correct designs, these extra subterms cancel out and vanish during the rewriting process. We have seen this to be the case even for very large and much more complex terms that may have millions of nodes.

We implement these lemmas as rewrite rules as well as some elementary algebraic properties in order to flatten and sort terms lexicographically in summations. Our rewrite rules do not subsume each other, and they may be applied with an arbitrary order until none of the rules are applicable.

Simplify Partial Products. Unlike the simple partial product generation method, multipliers with Booth encoding implement a more advanced algorithm to generate partial products. That results in terms that are more complex (see Example 6) than those we have addressed so far. We expand our rewriting mechanism for simplification of summation trees and add more rewrite rules for automated simplification of partial products such as the ones generated with Booth encoding and sign extension tricks.

Example 6. Below is a term for the second LSB of a multiplier output, implemented with Booth radix-4 encoding and before any simplification for partial products took place:

$$s([\neg b_1 b_0 a_1 \vee b_1 \neg b_0 \neg a_0 \vee b_1 b_0 \neg a1] \\ +c([b_1 b_0 \vee b_1 \neg b_0] \\ +[b_1 \neg b_0 \vee \neg b_1 b_0 a_0 \vee b_1 b_0 \neg a_0]))$$

Similar to other multiplier verification methods [25], we perform algebraic rewriting on the $\oplus$, $\vee$ and $\neg$ functions with the following lemmas.

Lemma 9. $\forall x \in \{0,1\}\ \neg x = 1 - x$

Lemma 10. $\forall x, y \in \{0,1\}\ x \vee y = x + y - xy$

Lemma 11. $\forall x, y \in \{0,1\}\ x \oplus y = x + y - xy\ \ - xy$

Example 7. Example 6 rewritten with Lemma 9, 10, and 11 as well as elementary algebraic properties.

$$s(b_1 + b_0 a_1 - b_1 a_0 + b_1 b_0 a_0 - b_1 b_0 a_1 - b_1 b_0 a_1 \\ +c(b_1 + b_1 + b_0 a_0 - b_1 b_0 a_0 - b_1 b_0 a_0))$$

We would like such expressions to be simplified to our final form. When deriving our rewrite rules, we concentrate on the terms with negative and/or duplicate arguments and realize that applying the following set of lemmas is sufficient to simplify such complex expressions.

Lemma 12. $\forall x, y \in \mathbb{Z}\ s((-x) + y) = s(x + y)$

Lemma 13. $\forall x, y \in \mathbb{Z}\ c((-x) + y) = (-x) + c(x + y)$

Lemma 14. $\forall x, y \in \mathbb{Z}\ d((-x) + y) = (-x) + d(x + y)$

Lemma 15. $\forall x, y \in \mathbb{Z}\ s(x + x + y) = s(y)$

Lemma 16. $\forall x, y \in \mathbb{Z}\ c(x + x + y) = x + c(y)$

Lemma 17. $\forall x, y \in \mathbb{Z}\ d(x + x + y) = x + d(y)$

Example 8. Below is the resulting term after Example 7 is simplified using Lemma 12–17 and elementary algebraic properties. We obtain a term matching the final form in Table 1.

$$s(b_0 a_1 + b_1 a_0 + c(b_0 a_0))$$

We implement these lemmas as rewrite rules along with the rules for simplification of summation trees. All of these lemmas automatically work together without any user intervention.

Algebraic rewriting of logical gates can be very expensive in terms of time and memory. For this reason, we limit the application of these rules to the partial

product logic only. For example, if applied indiscriminately, Lemmas 10 and 11 can cause terms to grow exponentially. Even though partial product generation logic may allocate a large area in multipliers, rewriting the adders to the s, c, and $+$ functions isolates partial products from each other and segregates them into small chunks. We expect that expressions representing partial products are composed of the $\vee$, $\wedge$, $\oplus$, and $\neg$ functions only. Therefore, we restrict Lemmas 9–11 to apply to terms that are composed of these functions only; and we restrict Lemmas 12–17 to apply to terms that are composed of minterms, and the $-$ and $+$ functions only. For instance, in Lemma 13, if we are unifying x with a term that contains an instance of s, c or d, then we prevent rewriting with a syntactic check. This heuristic helps contain this potentially expensive approach to only local and smaller terms.

Rewrite the Multiplier Specification. In our proposed rewriting scheme, we have a targeted representation for each output bit of multiplication as given in Definition 2. The rewriter cannot derive this form directly from the built-in ACL2 multiplication ($*$) function. Thus, we provide a recursively defined function *multbycol* that follows the formula in Definition 2. We prove *multbycol* to be equivalent to the $*$ function. When the rewriter works on the conjecture stating the correctness of a multiplier design as shown in Listing 1.1, `(truncate size (* a b))` is rewritten to `(multbycol a b size)`. The rewriter can then efficiently convert the specification into the targeted final form.

Using the rewriting mechanism described in this section, we can verify multipliers with Baugh-Wooley, sign/unsigned Booth radix-4, and simple partial product generation algorithms with various summation tree algorithms such as Wallace and Dadda tree. Note that Lemmas 9–17 work together with Lemmas 4–8 but contradict Lemmas 1–3. This is the reason why our method relies on semantics where the design hierarchy is maintained so that we can simplify the logic in adder modules with Lemmas 1–3 and simplify the remainder of a multiplier design with Lemmas 4–17 at a different time. When this separation is possible, multiplier designs are verified fully automatically without requiring users to designate the type of algorithm used. The complete process of proving the equivalence of semantics of a multiplier design to its specification is verified using ACL2.

5 Termination

Our rewriter does not enforce proof of termination for rewrite rules. The program terminates either when there are not any applicable rules or when a certain number of steps are taken, which may happen if that number is too small for the current conjecture, there is a loop between rules, or some rules grow some terms indefinitely. Even though it is not required by the rewriter, it is important to show that our rewriting algorithm requires a limited number of steps and does not run indefinitely.

Terms from conjectures change every time a rewrite rule is applied. Therefore, for each of our rewriting algorithms (adder and multiplier module simplification), we define a measure calculated on the term and show that it decreases every time we rewrite with one of our lemmas. We first define the measure for simplifying adder modules (Lemmas 1–3). Since carried out separately, we define another measure for the summation tree and partial product simplification algorithms (Lemmas 4–17). For brevity, we omit the discussion for termination with other lemmas pertaining to elementary algebraic properties such as commutativity and associativity.

5.1 Measure for Adder Module Simplification

The first part of our multiplier verification algorithm is simplifying the logic in adder components and rewriting them in terms of the s, c, and $+$ functions. Below, we define auxiliary functions and a measure that guarantees termination of this part of the algorithm that rewrites terms with Lemmas 1–3.

Definition 4 (f_1). *Function f_1 counts the number of symbols (constants, functions and variables) in a term.*

Definition 5 (f_2). *Function f_2 counts the occurrences of $\wedge$ and $\oplus$ in a term.*

For example, computing f_1 and f_2 on the term $s(x \oplus y + x \wedge z + c(x \oplus y))$ yields 13 and 3, respectively.

Definition 6 *We define a measure m_1 as follows, where the resulting ordered pairs are compared lexicographically.*

$$m_1(term) =< f_2(term), f_1(term) >$$

The pairs produced by m_1 are ordered lexicographically: thus, the value of m_1 decreases if f_2 decreases (no matter the value of f_1), or f_2 stays the same and f_1 decreases. Rewriting with Lemmas 1, 2, and 3 decreases f_2. Rewriting with some corollaries does not change the value of f_2 but decreases f_1 . For example, rewriting with the corollary $\forall x, y, h \in \{0, 1\}$ $c(x + y + h) \vee s(x + y) = c(x + y + 1)$ does not change f_2 but decreases f_1. In short, every step taken with these lemmas decreases the value of m_1 calculated on the resulting term. Therefore, the rewriting algorithm for adder modules terminates.

5.2 Measure for Multiplier Module Simplification

Rewriting for summation tree and partial product generation algorithms are performed together with a rewriting algorithm derived with Lemmas 4–17, excluding Lemmas 1–3. Therefore, we define a single measure to describe the termination of this part of the rewriting mechanism. Below we give definitions for some auxiliary functions and our measure.

Definition 7 (f_3). *Function f_3 sums the occurrence-depth of negative minterms, where the occurrence-depth is calculated with respect to the overall term.*

For example, computing f_3 on the term $s(x_0x_1 + c(-x_2y_0 + c(-x_3y_1)))$ yields 5 because its negative minterms $-x_2y_0$ and $-x_3y_1$ occur at depth 2 and 3, respectively. These values can be calculated by counting the unclosed parentheses from the beginning up to the occurrence of these terms.

Definition 8 (f_4). *Function f_4 computes the number of unique occurrences of functions $\{c,\ d,\ \neg,\ \oplus,\ \vee\}$.*

For example, computing f_4 for the term $c(x_0) + s(x_1 + c(x_0) + c(x_1))$ yields 2 because even though there are three instances of c, the second occurrence of $c(x_0)$ is not counted.

Definition 9. *We define measure m_2 to return ordered triples as follows, to be compared lexicographically.*

$$m_2(term) =< f_4(term), f_3(term), f_1(term) >$$

The value of m_2 decreases if f_4 decreases, or f_4 stays the same and f_3 decreases, or f_4 and f_3 stay the same and f_1 decreases. Below we discuss how rewriting with Lemmas 4–17 satisfy this measure for termination.

Rewriting with Lemmas 4 and 8 does not change the value of f_4. For both lemmas, if x is unified with a term that contains a negative minterm, then the value of f_3 decreases, otherwise, f_3 remains the same. By removing an instance of s, rewriting with both lemmas decreases f_1 and consequently m_2.

Rewriting with Lemmas 5, 6, 7, 9, 10, and 11 decreases f_4, and therefore m_2, by removing an instance of d, c, $\neg$, $\vee$ or $\oplus$. Even though rewriting with some of these lemmas creates copies of terms, the value of f_4 decreases because it does not count the same term more than once.

Rewriting with Lemmas 12–17 does not affect the value of f_4 since they are restricted to rewrite terms that contain only the $+$ and $-$ functions, and minterms. For Lemmas 12, 13, and 14, x can only be unified with a positive minterm. Therefore, rewriting with these lemmas does not change f_3. For Lemmas 15, 16, and 17, if x is unified with a negative minterm, then f_3 decreases. Otherwise, f_3 remains the same and f_1 decreases.

In short, rewriting with Lemmas 1–3 decreases the measure m_1 and rewriting with Lemmas 4-17 decreases the measure m_2. Therefore, our proposed rewriting mechanism terminates.

6 Experiments

In this section, we present our experimental results and compare them to the other state-of-the-art tools for the automated verification of multiplier designs. We have gathered a large set of multipliers from 3 different generators, and run

all the experiments for other verification tools and ours on the same computer (A 2014 model iMac Intel(R) Core(TM) i7-4790K CPU @ 4.00 GHz with 32 GB system memory) for comparison. The instructions and a ready-to-run VM image to run our tool and reproduce these experimental results can be found online at http://mtemel.com/mult.html.

For benchmarking, we used 3 different generators. The tool from Homma et al. [10] generates Booth encoded sign and unsigned multipliers (input size up to 64 bits) with various summation tree and final stage adders. Designs from Homma et. al. have multiple copies of half/full-adder modules as well as some other adder modules. Since our method requires reasoning about each adder module, we wrote a function that scans the modules and automatically simplifies them as described in Sect. 4.1. Secondly, we used *SCA-genmul* [24] to generate simple unsigned and Baugh-Wooley based signed (also referred to as simple signed) multipliers. This tool does not generate Booth-encoded multipliers. Finally, we used another multiplier generator [34] that can generate large Booth-encoded multipliers.

We have measured the complete proof time for each benchmark, when available, and compared our results to the work of D. Kaufmann et al. [16] and A. Mahzoon et al. [23]. These methods are based on computer algebra, and they are the best performing tools at the time this paper is rewritten. Since we verified the correctness of our tool using ACL2, we do not generate certificates. D. Kaufmann et al. implement their method in a stand-alone C program but they generate certificates to check their proofs. We measured the total time to verify/certify and check certificates. A. Mahzoon et al. also test their method with a stand-alone C program but it does not produce any certificates. Even though it is not a complete comparison, we still include the results of their tool for the same benchmarks.

When we run our tool on these benchmarks, we only need to identify the names of the adder modules, their I/O size; multiplier I/O size, and whether they perform signed or unsigned multiplication in order to determine their specification. The proofs finish automatically, and users can see the specification explicitly to validate what is proved. The other tools are not interactive and use some heuristics to decide on the specification internally based on the design.

D. Kaufmann et al. [16] and A. Mahzoon et al. [23] both use AIGs as inputs, and we use SVL [33], all of which are translated from (System) Verilog using external tools. For the other tools, we used Yosys [39] and ABC [3] to create AIGs, without any optimization. For our tool, we created SVL netlists as described in Sect. 2.2. Since we compare the performance of different verification methods, we do not include the translation time in any of these results.

Table 3 shows the result of experiments run with a collection of circuits. The benchmarks are described with the generator, partial product generation algorithm, summation tree algorithm, and final stage adder. Generators are *tem* [34], *sca* [24], and *hom* [10]. Partial product generation algorithms are *sp* (simple unsigned/signed or Baugh-Wooley-based), and *bp* (unsigned and signed Booth radix-4 encoded). Summation tree algorithms are *dt* (Dadda tree), *wt* (Wal-

Table 3. Proof-time results in seconds for various multiplier designs

Size	Benchmark	AM [23][a]	DK [16]		Our tool	
		Unsigned	Unsigned	Signed	Unsigned	Signed
64 × 64	sca sp-dt-bk	39	6	6	1	1
	sca sp-wt-lf	33	6	6	1	1
	sca sp-cwt-ks	TO	65	58	1	1
	sca sp-ar-rc	23	5	5	1	1
	tem sp-dt-ks	173	7	7	1	1
	tem sp-wt-lf	33	6	6	1	1
	tem bp-dt-hc	TO	44	49	1	1
	tem bp-wt-rp	TO	45	49	2	2
	hom bp-dt-ks	288	8	TE	2	2
	hom bp-bdt-hc	TO	7	7	2	2
	hom bp-os-bk	71	6	TO	3	3
	hom bp-wt-cla	108	24	21	13	12
	hom bp-4:2-lf	TE	7	7	3	3
128 × 128	sca sp-dt-bk	643	33	36	2	3
	sca sp-wt-lf	633	34	38	2	2
	sca sp-cwt-ks	TO	TO	TO	3	3
	sca sp-ar-rc	384	27	27	18	18
	tem sp-dt-ks	TO	47	49	2	3
	tem sp-wt-lf	650	40	40	2	2
	tem bp-dt-hc	TO	877	1037	7	7
	tem bp-wt-rp	TO	918	1067	12	13
256 × 256	sca sp-dt-bk	TO	213	209	9	11
	sca sp-wt-lf	15351	226	223	11	13
	sca sp-cwt-ks	TO	TO	TO	13	15
	tem sp-dt-ks	TO	234	232	10	12
	tem sp-wt-lf	15552	220	221	10	12
	tem bp-dt-hc	TO	11555	14043	41	47
	tem bp-wt-rp	TO	11975	14264	54	58
512 × 512	sca sp-dt-bk	TO	1562	1562	53	64
	sca sp-wt-lf	TO	1588	1577	61	76
	tem sp-dt-ks	TO	1655	1655	68	75
	tem sp-wt-lf	TO	1604	1609	65	82
	tem bp-dt-hc	TO	TO	TO	246	281
	tem bp-wt-rp	TO	TO	TO	371	380
1024 × 1024	sca sp-dt-bk	TO	13746	13247	339	397
	sca sp-wt-lf	TO	13560	14005	322	345
	tem sp-dt-ks	TO	14125	15198	324	392
	tem sp-wt-lf	TO	13664	13708	327	393

[a] Does not produce certificates.
TE: Terminated with an error. **TO**: Time-out. 5400 s. (90 min) for 64 × 64 and 128 × 128, 16200 s (270 min) for the rest.

lace tree), *cwt* (counter-based Wallace tree), *ar* (array), *os* (overturned-stairs tree), *bdt* (balanced delay tree), and *4:2* (4-to-2 compressor tree). Finally, the final stage adders are *bk* (Brent-Kung), *lf* (Ladner-Fischer), *rc* (Ripple-carry), *ks* (Kogge Stone), *csk* (Carry-skip), *hc* (Han-Carlson), and *cla* (Carry-lookahead). The selection of benchmarks was arbitrary but we have concentrated on Wallace-tree-like multipliers with complex final stage adders as they have a more widespread industrial application. For experiments with 64 × 64 and 128 × 128 multipliers, we set the time limit to 1.5 h, and for larger designs, we set the limit to 4.5 h. The results are given in seconds rounded to the nearest integer.

For all the benchmarks we have tested, our tool out-performed the other tools in all cases. Our method is shown to verify benchmarks the others cannot and produce a more homogeneous timing performance across different designs. A. Mahzoon et al. [23] work only on unsigned multipliers. Both A. Mahzoon et al. and D. Kaufmann et al. [16] give fluctuating results for multipliers with different architectures and/or different generators. For some benchmarks, the other tools terminated with an error such as segmentation fault (marked with TE). Our work is more resilient to differences in designs and it scales much better (proof times increase by 4.5–6 times when circuit size grows 4 times). For Wallace-tree like multipliers with simple partial products, about 40% of the time on average is spent on simplification with the lemmas given in Sect. 4, and the rest is spent by conversion of SVL semantics to ACL2 expressions. For multipliers with Booth-encoding, over 70% of the time is spent on partial product simplification. Array multipliers are the only type of circuit for which our tool struggles to scale. We believe that is because the minimal parallelism this circuit implements causes our rewriting engine to do much more work as compared to other multiplier structures. Even though memory use is not reported here, it scales the same way as timings, and it grows as big as 30 GB for the largest (1024 × 1024) circuits we have tested.

Additionally, since integer multipliers are used to implement floating-point operations, we tested our method in a correctness proof for an implementation of a floating-point multiply-add instruction for Centaur Technology, and we got similar results.

7 Related Work and Conclusion

Having described our method, we now compare it with the related work. Well-known methods to verify multipliers include generic reasoning methods such as BDDs and SAT solvers. However, these tools do not scale well with large multipliers. For the last few years, efforts to verify large integer multipliers have explored the symbolic computer algebra approach based on Gröbner basis [7,16,22,23,28,37]. As far as we are aware, all these tools are stand-alone, unverified C programs and none of them except D. Kaufmann et al. [16] produces certificates. The soundness and completeness of this approach is shown only in theory [17]. We compared our method to the studies with the best timing performance [16,23]. The tools implementing these methods identify adder

components in designs automatically and perform some rewriting. Their rewriting strategy is different than ours; their method does not rely on maintained design hierarchy and separate reasoning of adder and multiplier modules. Even though they provide a more automatic system, their application appears to be limited to some known patterns. Additionally, our tool is implemented on an interactive tool, which can enable users to carry out more complicated proofs such as the correctness of floating-point circuits. The limitation of our method is that it relies on maintaining circuit hierarchy. Should this pose a problem for some designs, it might be possible for our method to be adapted in the future to work with flattened modules and identify adder components similarly to the related work.

When a proof fails for a multiplier design, our tool does not output a user-friendly message. We will work to improve our tool to process the resulting terms from failed verification attempts and generate counterexamples for incorrect designs.

In this paper, we have presented an efficient method with a proven tool to verify large and complex integer multipliers. With maintained circuit hierarchy, we can automatically verify very irregular multiplier designs; for example, various 1024×1024 Wallace-tree like multipliers can be verified in less than 10 min. We believe that our tool can find broader applications because it can be extended to verify circuits, such as floating-point multipliers, that include an integer multiplier as a submodule.

Acknowledgments. We would like to thank the reviewers for their feedback, and Matt Kaufmann for his helpful directives when implementing this method in ACL2. This material is based upon work supported in part by DARPA under Contract No. FA8650- 17-1-7704. A part of this work was completed while M. Temel was working at Centaur Technology.

References

1. Baugh, C.R., Wooley, B.A.: A two's complement parallel array multiplication algorithm. IEEE Trans. Comput. **C–22**, 1045–1047 (1973). https://doi.org/10.1109/t-c.1973.223648
2. Booth, A.D.: A signed binary multiplication technique. Q. J. Mech. Appl. Math. **4**, 236–240 (1951). https://doi.org/10.1093/qjmam/4.2.236. Oxford University Press (OUP)
3. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5
4. Brent, K.: A regular layout for parallel adders. IEEE Trans. Comput. **C–31**(3), 260–264 (1982). https://doi.org/10.1109/tc.1982.1675982
5. Bryant, R.E., Chen, Y.A.: Verification of arithmetic functions with binary moment diagrams. In: DAC 1994 (1994). https://doi.org/10.21236/ada281028
6. Burch, J.R.: Using BDDs to verify multipliers. In: Proceedings of the 28th ACM/IEEE Design Automation Conference, DAC 1991, pp. 408–412. Association for Computing Machinery, New York (1991). https://doi.org/10.1145/127601.127703

7. Ciesielski, M., Su, T., Yasin, A., Yu, C.: Understanding algebraic rewriting for arithmetic circuit verification: a bit-flow model. IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. (2019). https://doi.org/10.1109/tcad.2019.2912944
8. Dadda, L.: Some Schemes for Parallel Multipliers (1965)
9. Han, T., Carlson, D.A.: Fast area-efficient VLSI adders. In: 1987 IEEE 8th Symposium on Computer Arithmetic, pp. 49–56 (1987). https://doi.org/10.1109/arith.1987.6158699
10. Homma, N., Watanabe, Y., Aoki, T., Higuchi, T.: Formal design of arithmetic circuits based on arithmetic description language. IEICE Trans. **89–A**, 3500–3509 (2006). https://doi.org/10.1109/ispacs.2006.364918. https://www.ecsis.riec.tohoku.ac.jp/topics/amg/
11. Hsiao, S.F., Jiang, M.R., Yeh, J.S.: Design of high-speed low-power 3–2 counter and 4–2 compressor for fast multipliers. Electron. Lett. **34**, 341–343 (1998). https://doi.org/10.1049/el:19980306. Institution of Engineering and Technology (IET)
12. Hunt, W.A., Kaufmann, M., Moore, J.S., Slobodova, A.: Industrial hardware and software verification with ACL2. Philos. Trans. Roy. Soc. A Math. Phys. Eng. Sci. **375**(2104), 20150399 (2017). https://doi.org/10.1098/rsta.2015.0399
13. Hunt, W.A., Swords, S., Davis, J., Slobodova, A.: Use of formal verification at centaur technology. In: Hardin, D. (ed.) Design and Verification of Microprocessor Systems for High Assurance Applications, pp. 65–88. Springer, Heidelberg (2010). https://doi.org/10.1007/978-1-4419-1539-9_3
14. Jacobi, C., Weber, K., Paruthi, V., Baumgartner, J.: Automatic formal verification of fused-multiply-add FPUS. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2005, vol. 2, pp. 1298–1303. IEEE Computer Society, USA (2005). https://doi.org/10.1109/DATE.2005.75
15. Kaivola, R., Narasimhan, N.: Formal verification of the pentium ® 4 floating-point multiplier. In: 2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002), Paris, France, 4–8 March 2002, pp. 20–27 (2002). https://doi.org/10.1109/DATE.2002.998245
16. Kaufmann, D., Biere, A., Kauers, M.: Verifying large multipliers by combining SAT and computer algebra. In: 2019 Formal Methods in Computer Aided Design (FMCAD), pp. 28–36 (2019). https://doi.org/10.23919/FMCAD.2019.8894250
17. Kaufmann, D., Biere, A., Kauers, M.: Incremental column-wise verification of arithmetic circuits using computer algebra. Formal Methods Syst. Des. (2019). https://doi.org/10.1007/s10703-018-00329-2
18. Kaufmann, M., Moore, J.S.: ACL2 rule classes documentation (2019). http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/ACL2____RULE-CLASSES
19. Kogge, P.M., Stone, H.S.: A parallel algorithm for the efficient solution of a general class of recurrence equations. IEEE Trans. Comput. **C–22**(8), 786–793 (1973). https://doi.org/10.1109/tc.1973.5009159
20. Ladner, R.E., Fischer, M.J.: Parallel prefix computation. J. ACM (JACM) **27**(4), 831–838 (1980). https://doi.org/10.1145/322217.322232
21. MacSorley, O.L.: High-speed arithmetic in binary computers. Proc. IRE **49**, 67–91 (1961). https://doi.org/10.1109/jrproc.1961.287779
22. Mahzoon, A., Große, D., Drechsler, R.: PolyCleaner: clean your polynomials before backward rewriting to verify million-gate multipliers. In: 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 1–8 (2018). https://doi.org/10.1145/3240765.3240837

23. Mahzoon, A., Große, D., Drechsler, R.: RevSCA: using reverse engineering to bring light into backward rewriting for big and dirty multipliers. In: Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, pp. 185:1–185:6. ACM, New York (2019). https://doi.org/10.1145/3316781.3317898
24. Mahzoon, A., Große, D., Drechsler, R.: SCA multiplier generator GenMul (2019). http://www.sca-verification.org
25. Ritirc, D., Biere, A., Kauers, M.: A practical polynomial calculus for arithmetic circuit verification. In: Bigatti, A.M., Brain, M. (eds.) 3rd International Workshop on Satisfiability Checking and Symbolic Computation (SC2 2018), pp. 61–76. CEUR-WS (2018)
26. Rosenberger, G.B.: Simultaneous Carry Adder (1960)
27. Russinoff, D.M.: Formal Verification of Floating-Point Hardware Design: A Mathematical Approach. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-319-95513-1
28. Sayed-Ahmed, A., Große, D., Kühne, U., Soeken, M., Drechsler, R.: Formal verification of integer multipliers by combining Gröbner basis with logic reduction. In: Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1048–1053. Research Publishing Services (2016). https://doi.org/10.3850/9783981537079_0248
29. Sharangpani, H., Barton, M.L.: Statistical Analysis of Floating Point Flaw in the Pentium Processor (1994)
30. Slobodova, A., Davis, J., Swords, S., Hunt, W.A.: A flexible formal verification framework for industrial scale validation. In: Proceedings of the 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 89–97. IEEE/ACM, Cambridge (2011). https://doi.org/10.1109/memcod.2011.5970515
31. Swords, S.: ACL2 SV Documentation (2015). http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____SV
32. Swords, S.: ACL2 VL Documentation (2015). http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____VL
33. Temel, M.: ACL2 SVL Documentation (2019). http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____SVL
34. Temel, M.: Fast Multiplier Generator (2019). https://github.com/temelmertcan/multgen
35. Vasudevan, S., Viswanath, V., Sumners, R.W., Abraham, J.A.: Automatic verification of arithmetic circuits in RTL using stepwise refinement of term rewriting systems. IEEE Trans. Comput. **56**(10), 1401–1414 (2007). https://doi.org/10.1109/tc.2007.1073
36. Wallace, C.S.: A suggestion for a fast multiplier. IEEE Trans. Electron. Comput. **13**, 14–17 (1964). https://doi.org/10.1109/pgec.1964.263830
37. Watanabe, Y., Homma, N., Aoki, T., Higuchi, T.: Application of symbolic computer algebra to arithmetic circuit verification (2007). https://doi.org/10.1109/iccd.2007.4601876
38. Weste, N., Harris, D.M.: Principles of CMOS VLSI Design: a systems perspective. STIA **85**, 547–555 (1993)
39. Wolf, C., Glaser, J., Kepler, J.: Yosys-A Free Verilog Synthesis Suite (2013)
40. Yu, C., Ciesielski, M., Mishchenko, A.: Fast algebraic rewriting based on and-inverter graphs. IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. **37**(9), 1907–1911 (2018). https://doi.org/10.1109/tcad.2017.2772854

Interpolation-Based Semantic Gate Extraction and Its Applications to QBF Preprocessing

Friedrich Slivovsky(✉)

TU Wien, Vienna, Austria
fs@ac.tuwien.ac.at

Abstract. We present a new semantic gate extraction technique for propositional formulas based on interpolation. While known gate detection methods are incomplete and rely on pattern matching or simple semantic conditions, this approach can detect any definition entailed by an input formula.

As an application, we consider the problem of computing unique strategy functions from Quantified Boolean Formulas (QBFs) and Dependency Quantified Boolean Formulas (DQBFs). Experiments with a prototype implementation demonstrate that functions can be efficiently extracted from formulas in standard benchmark sets, and that many of these definitions remain undetected by syntactic gate detection.

We turn this into a preprocessing technique by substituting unique strategy functions for input variables and test solver performance on the resulting instances. Compared to syntactic gate detection, we see a significant increase in the number of solved QBF instances, as well as a modest increase for DQBF instances.

1 Introduction

Due to the effectiveness of modern satisfiability (SAT) solvers [20], propositional logic has become the language of choice for encoding hard combinatorial problems arising in areas such as electronic design automation [50] and AI planning. Since many of these problems are hard for levels of the polynomial hierarchy beyond NP, their propositional encodings can be exponentially larger than their original descriptions. This imposes a limit on the problem instances that can be feasibly solved even with extremely efficient SAT solvers, and has prompted research on decision procedures for more succinct logical formalisms such as Quantified Boolean Formulas (QBFs).

Quantified Boolean Formulas (QBFs) are propositional formulas combined with universal and existential quantification over truth values and offer much more succinct encodings of problems from domains such as planning and synthesis [12]. At the same time, QBF evaluation is PSPACE-complete, and in spite

This research was supported by the Vienna Science and Technology Fund (WWTF) under grant number ICT19-060.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 508–528, 2020.
https://doi.org/10.1007/978-3-030-53288-8_24

of substantial progress in solver technology, many practically relevant instances remain hard to solve.

In part, this hardness appears to be a matter of encoding. The most commonly used format for QBFs is Prenex Conjunctive Normal Form (PCNF). A PCNF formula consists of a quantifier prefix and a matrix in conjunctive normal form. As in the case of propositional logic, any QBF can be converted to PCNF with linear overhead but this transformation is known to adversely affect solver performance [1]. This appears to be due to two issues: First, conversion to CNF causes a bias towards reasoning about unsatisfiability while making it difficult to reason about solutions, violating the inherent duality of QBF solving. Second, prenexing introduces spurious variable dependencies that needlessly constrain solvers [5,40]. In light of these issues, researchers have introduced two new formats for representing non-CNF (and even non-prenex) QBFs in the QCIR [30] and QAIGER standards, and solvers supporting these standards have been developed. When only a PCNF encoding is available, *gate extraction* techniques can be used to (re)construct a non-CNF QBF [21]. *Syntactic* gate extraction relies on the detection of patterns of clauses and auxiliary variables introduced when converting a propositional formula to CNF [16]. The corresponding algorithms are fast but incomplete and can only detect definitions from a pre-defined library of gates.

In this paper, we introduce a new semantic gate extraction technique based on SAT solving and interpolation. In contrast to known approaches, this method is complete: a definition ψ of a variable x can be extracted from a propositional formula φ whenever the equivalence $x \equiv \psi$ is entailed by φ. We obtain this result as a generalization of recent work that leverages definability for propositional model counting [25,33]. Owing to a result known as Padoa's Theorem, determining whether a variable x is definable in terms of X is in coNP and can be decided by a SAT call [33]. We show that a definition ψ of x in terms of X can be obtained as an *interpolant* of the formula passed to the SAT solver (Theorem 2). For SAT solvers that use a proof system with feasible interpolation—in particular, CDCL solvers that generate resolution proofs [32]—this means a definition can be efficiently extracted from a proof of definability.

We apply this new gate extraction technique to identify unique strategy functions of QBFs and Dependency QBFs. In a controller synthesis setting, a variable with a unique strategy function corresponds to a control signal with a unique (as a Boolean function) implementation. We can add such an implementation to the specification without affecting the remaining control signals.

Experiments with a prototype show that definitions can be efficiently computed for formulas from standard QBF benchmark sets, and that for many instances a large fraction of variables have unique strategy functions that cannot be identified by syntactic gate detection. We further test the performance of solvers on instances obtained by replacing input variables with their definitions. For 2QBF formulas and PCNF formulas, this significantly increases the number of instances solved by some systems compared to purely syntactic gate extraction. Our experiments further show that semantic gate detection is orthogonal to techniques implemented in state-of-the-art preprocessors.

Semantic gate detection is efficient and conceptually simple. By definition, it preserves logical equivalence and is compatible with strategy extraction. As such, we believe it is an essential addition to the state of the art in preprocessing (D)QBF.

2 Preliminaries

We assume a countably infinite set V of propositional *variables* and consider *propositional formulas* constructed from V using the connectives $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), and $\leftrightarrow$ (the biconditional). For a propositional formula φ, we write $var(\varphi)$ to denote the set of variables occurring in φ. A *literal* is a variable v or a negated variable $\neg v$. A *clause* is a finite disjunction of literals. A clause is *tautological* if it contains both v and $\neg v$ for some variable v. A propositional formula is in *conjunctive normal form (CNF)* if it is a finite conjunction of non-tautological clauses. An *assignment* of a subset $X \subseteq V$ of variables is a function that maps X to the set $\{0,1\}$ of truth values. For a set X of variables we let $[X]$ denote the set of assignments of X. Two assignments $\sigma : X \to \{0,1\}$ and $\tau : Y \to \{0,1\}$ *agree* on a subset $W \subseteq X \cap Y$ of their common domain if $\sigma(w) = \tau(w)$ for each $w \in W$. For two assignments $\sigma : X \to \{0,1\}$ and $\tau : Y \to \{0,1\}$ that agree on the entire intersection of their domains we define the combined assignment $\sigma \cup \tau : X \cup Y \to \{0,1\}$ as $(\sigma \cup \tau)(v) = \sigma(v)$ if $v \in X$ and $(\sigma \cup \tau)(v) = \tau(v)$ otherwise.

For a propositional formula φ and an assignment $\tau : X \to \{0,1\}$ with $var(\varphi) \subseteq X$, we let $\varphi[\tau]$ denote the truth value obtained by evaluating φ under τ. The formula φ is *satisfied* by τ if $\varphi[\tau] = 1$. In this case we call τ a *satisfying assignment* of φ. Otherwise, if $\varphi[\tau] = 0$, formula φ is *falsified* by τ. A formula is *satisfiable* if it has a satisfiable assignment, otherwise it is *unsatisfiable*. A formula φ *implies* a formula ψ if $\varphi \wedge \neg\psi$ is unsatisfiable.

We consider Quantified Boolean Formulas (QBFs) in *Prenex Normal Form (PNF)*. A QBF $\Phi = \mathcal{Q}.\varphi$ in PNF consists of a *quantifier prefix* $\mathcal{Q}$ and a propositional formula φ, called the *matrix* of Φ. The quantifier prefix is a sequence $Q_1 x_1 \dots Q_n x_n$ where $Q_i \in \{\forall, \exists\}$ and the x_i are pairwise distinct variables for $1 \leq i \leq n$. The quantifier prefix defines an ordering $<_\Phi$ on its variables as $x_i <_\Phi x_j$ for $1 \leq i < j \leq n$. We assume that QBFs do not contain free variables and every variable in the quantifier prefix appears in the matrix, formally $\{x_1, \dots, x_n\} = var(\varphi)$. Accordingly, we write $var(\Phi) = var(\varphi)$ for the set of variables appearing in the QBF Φ. We further assume that every variable of Φ occurs exactly once in its quantified prefix. The set of *existential* variables of Φ is $var_\exists(\Phi) = \{\, x_i \mid 1 \leq i \leq n, Q_i = \exists \,\}$, and the set of *universal* variables of Φ is $var_\forall(\Phi) = \{\, x_i \mid 1 \leq i \leq n, Q_i = \forall \,\}$. For a variable $x \in var(\Phi)$, we let $type_\Phi(x) = Q$ if $x \in var_Q(\Phi)$, for $Q \in \{\forall, \exists\}$, omitting Φ from the subscript if the QBF is understood.

Let Φ a QBF and let $x \in var(\Phi)$ be one of its variables with $type(x) = Q$. A *strategy function* for x is a function $f : [var(\Phi) \setminus var_Q(\Phi)] \to \{0,1\}$ such that $f(\tau) = f(\tau')$ for any two assignments τ and τ' that agree on variables in

$\{ v \in var(\Phi) \setminus var_Q(\Phi) \mid v <_\Phi x \}$.[1] Given an indexed family $F = \{f_x\}_{x \in X}$ of strategy functions such that $X \subseteq var_Q(\Phi)$ for $Q \in \{\forall, \exists\}$, the *response* of F to an assignment $\tau : (var(\Phi) \setminus var_Q(\Phi)) \to \{0,1\}$ is the assignment $F(\tau) : X \to \{0,1\}$ given by $F(\tau)(x) = f_x(\tau)$. An *existential winning strategy* (for Φ) is a family $F = \{f_u\}_{u \in var_\exists(\Phi)}$ of strategy functions such that, for any universal assignment $\tau : var_\forall(\Phi) \to \{0,1\}$, the assignment $\tau \cup F(\tau)$ satisfies the matrix of Φ. Dually, a *universal winning strategy* (for Φ) is a family $F = \{f_u\}_{u \in var_\forall(\Phi)}$ of strategy functions such that, for any existential assignment $\sigma : var_\exists(\Phi) \to \{0,1\}$, the assignment $\sigma \cup F(\sigma)$ falsifies the matrix. A QBF Φ is *true* if there is an existential winning strategy for Φ, and *false* if there exists a universal winning strategy for Φ.

3 Semantic Gate Extraction by Interpolation

This work builds on an application of propositional *definability* to the model counting problem [33]. We begin by recalling two basic concepts.

Definition 1. *Let φ be a formula, let X be a subset of its variables, and let x be a variable. Variable x is* defined *in terms of X in φ if $\sigma(x) = \tau(x)$ for any two satisfying assignments σ and τ of φ that agree on X. A* definition *of x by X in φ is a formula ψ with $var(\psi) \subseteq X$ such that $\sigma(x) = \psi[\sigma]$ for any satisfying assignment σ of φ.*

It is readily verified that there is a definition for every variable that is defined. Lagniez et al. [33] observe that the following result can be used to determine whether a variable is defined [34,39].

Theorem 1 (Padoa's Theorem). *Let φ be a formula and let $X \subseteq var(\varphi)$ be a subset of its variables. Let φ' be the propositional formula obtained by replacing every variable $y \in var(\varphi) \setminus X$ by a new variable y'. Let $x \in var(\varphi)$ be a variable. If $x \notin X$, then x is defined in φ by X if, and only if, the formula $\varphi \wedge x \wedge \varphi' \wedge \neg x'$ is unsatisfiable.*

For the purposes of preprocessing in model counting, it is sufficient to know *that* a variable x is defined by X in φ, and the above result shows that this can be decided by a SAT solver. It is not necessary to compute the corresponding definition, whose size is not polynomially bounded in the size of φ under common assumptions in computational complexity [33].

While finding definitions is harder than deciding definability in theory, the difference virtually disappears in practice. Our main theoretical contribution, stated as Theorem 2 below, says that a definition can be obtained as an *interpolant* of the formula constructed in the statement of Padoa's Theorem. Since interpolants can be efficiently (in linear time) generated from resolution proofs [22,32], the distinction between detecting definability and computing definitions

[1] We sometimes refer to existential strategy functions as *Skolem* functions and universal strategy functions as *Herbrand* functions.

becomes moot when a CDCL SAT solver is used to decide (un)satisfiability: once it determines that the formula is unsatisfiable it has already (implicitly or explicitly) produced a proof from which a definition can be extracted at a small overhead.[2]

Before proving Theorem 2, we recall the definition of an interpolant following McMillan [36].

Definition 2 (Interpolant). *Let ψ and χ be an formulas such that $\psi \wedge \chi$ is unsatisfiable. An* interpolant *for ψ and χ is a formula I such that*

(1) ψ implies I,
(2) $I \wedge \chi$ is unsatisfiable, and
(3) I only refers to variables common to ψ and χ.

Craig's Interpolation Theorem [9] states that every pair of jointly unsatisfiable propositional formulas have an interpolant.[3] It remains to show that an interpolant for a formula witnessing definability in fact yields a definition.

Lemma 1. *Let φ be a formula and let $X \subseteq var(\varphi)$ be a subset of its variables. Let φ' be the formula obtained by replacing every variable $y \in var(\varphi) \setminus X$ by a new variable y'. For any variable $x \in var(\varphi) \setminus X$, an interpolant for $\varphi \wedge x$ and $\varphi' \wedge \neg x'$ is a definition of x by X in φ.*

Proof. Let I be an interpolant for $\varphi \wedge x$ and $\varphi' \wedge \neg x'$. By property (3) of Definition 2, I only refers to the common variables $var(\varphi \wedge x) \cap var(\varphi' \wedge \neg x') = X$ of these formulas. To see that I defines x in φ, consider a satisfying assignment $\sigma : var(\varphi) \to \{0, 1\}$ of φ. If $\sigma(x) = 1$ then $\varphi \wedge x$ is satisfied by σ. The formula $\varphi \wedge x$ implies I by property (1), so $I[\sigma] = 1$ as well. Otherwise, $\sigma(x) = 0$ and we can construct a satisfying assignment σ' of $\varphi' \wedge \neg x'$ by setting $\sigma'(v) = \sigma(v)$ for $v \in X$ along with $\sigma'(v') = \sigma(v)$ for $v \in var(\varphi) \setminus X$. By property (2), $I \wedge \varphi' \wedge \neg x'$ is unsatisfiable, so we must have $I[\sigma'] = I[\sigma] = 0$.

Theorem 2. *Let φ be a formula and let $X \subseteq var(\varphi)$ be a subset of its variables. Let φ' be the formula obtained by replacing every variable $y \in var(\varphi) \setminus X$ by a new variable y'. A variable $x \in var(\varphi) \setminus X$ is defined in terms of X in φ if, and only if, the formula $\varphi \wedge x \wedge \varphi' \wedge \neg x'$ is unsatisfiable, and a definition of x in terms of X can be obtained as an interpolant for $\varphi \wedge x$ and $\varphi' \wedge \neg x'$.*

Proof. By Theorem 1 variable $x \in var(\varphi) \setminus X$ is defined in terms of X in φ if, and only if, the formula $\varphi \wedge x \wedge \varphi' \wedge \neg x'$ is unsatisfiable. Craig's Interpolation Theorem tells us that in this case there is an interpolant for $\varphi \wedge x$ and $\varphi' \wedge \neg x'$, which defines x in terms of X by Lemma 1.

[2] Assuming the SAT solver does not use the full power of the DRAT proof system [51].
[3] In fact, the result holds even for first order logic, but we will confine ourselves to the propositional case.

4 Extracting Unique QBF Strategy Functions

In this section, we show how Theorem 2 can be used to extract unique strategy functions of QBFs. We say that the Skolem (Herbrand) function of an existential (universal) variable x in a QBF is *unique* if it is the same in every existential (universal) winning strategy. In particular, if x is existentially (universally) quantified and the formula is false (true), then the strategy function of x is trivially unique (there is none). In other words, the strategy function of a variable x is unique if there is *at most one* such function for x that is part of a winning strategy. The following result states that propositional definability is a sufficient condition for uniqueness of a strategy function.

Proposition 1. *Let $\Phi = Q_1 x_1 \dots Q_n x_n.\varphi$ be a QBF. If an existential (universal) variable x_i is defined in terms of variables $X \subseteq \{ x_j \mid 1 \leq j < i, Q_j \neq Q_i \}$ in φ ($\neg\varphi$) its Skolem (Herbrand) function is unique.*

Proof. We only consider the case where x_i is an existential variable of Φ (the case where x_i is a universal variable is symmetric). Let $F = \{f_{x_j}\}_{x_j \in var_\exists(\Phi)}$ and $G = \{g_{x_j}\}_{x_j \in var_\exists(\Phi)}$ be existential winning strategies and $\tau : var_\forall(\Phi) \to \{0,1\}$ an assignment to the universal variables. Since F and G are existential winning strategies both $\sigma_F = \tau \cup F(\tau)$ and $\sigma_G = \tau \cup G(\tau)$ must be satisfying assignments of φ. The assignments σ_F and σ_G agree on $X \subseteq var_\forall(\Phi)$, so we must have $f_{x_i}(\tau) = \sigma_F(x_i) = \sigma_G(x_i) = g_{x_i}(\tau)$ because x_i is defined in terms of X. Since τ was chosen arbitrarily, this identity holds for every universal assignment, so the functions f_{x_i} and g_{x_i} coincide.

To see that definability is not a *necessary* condition for a strategy function to be unique, consider the following example.

Example 1. Let $\Phi = \forall x \exists y \forall z.(x \leftrightarrow y) \vee z$. The formula $\psi = x$ represents the unique existential winning strategy (set y to the same value as x). However, variable y is not defined in terms of x: the assignments $\{x, y, z\}$ and $\{x, \neg y, z\}$ both satisfy the matrix and agree on x, but differ on y. Intuitively, the reason why the existential strategy function for y is unique in spite of y not being defined is that the universal player would never assign z true as required by one of the assignments witnessing non-definability.

4.1 An Algorithm for Computing Unique Strategy Functions

We now describe an algorithm for computing unique strategy functions of a QBF based on Proposition 1. By using an interpolating SAT solver (ItpSatSolver) that supports both incremental solving and assumptions [22], we can extract definitions for variables of a given quantifier type (universal or existential) using a single solver instance. Pseudocode is shown as Algorithm 1 below.

Let $\Phi = Q_1 x_1 \dots Q_n x_n.\varphi$ be a QBF and let $Q \in \{\forall, \exists\}$ be a quantifier type. Algorithm 1 first determines the leftmost variable x_i in the prefix of Φ that has quantifier type Q (line 3). The strategy function of any variable to the

right of x_i in the prefix (including x_i itself) may use the variables to its left (*shared*), so we can begin by looking for definitions of x_i in terms of *shared*. Towards constructing the formula for the corresponding unsatisfiability check according to Theorem 2, $\textsc{copy}(\varphi, X)$ returns a copy φ' of the matrix φ where each variable $x \in var(\varphi) \setminus shared$ has been replaced by a fresh variable x'. Next (lines 9–14), we consider each variable x_j with quantifier type Q—these are the variables we want to find definitions of—and introduce two fresh "selector" variables s_i and s'_i, while adding clauses $(\neg s_j \vee x_j)$ and $(\neg s'_j \vee \neg x'_j)$ to φ and φ', respectively. These clauses allow us to represent $\varphi \wedge x_j \wedge \varphi' \wedge \neg x'_j$ by assuming literals s_j and s'_j.[4]

After initializing the SAT solver, we consider the variables $x_1, \ldots, x_n$ in the order of the quantifier prefix (lines 18–29). If variable x_j has quantifier type Q, we want to check whether x_j is defined in φ in terms of oppositely quantified variables X_j that precede it in the prefix (Proposition 1 tells us that in this case the strategy function of x_j is unique). For the first such variable x_j, it is clear that the set of variables common to φ and φ' is precisely X. Unsatisfiability of $\varphi \wedge x_j \wedge \varphi' \wedge \neg x'_j$ is decided by calling the SAT solver under assumptions $\{s_j, s'_j\}$: the assumptions ensure that x_j and $\neg x'_j$ are set to true by propagation, and all remaining selector variables can be set to false so as to satisfy the clauses they occur in without interfering with the remaining clauses. If the solver determines unsatisfiability, an interpolant I_j is computed (line 22), which by Theorem 2 corresponds to a definition of x_j, and adds the pair (x_j, I_j) to a list of definitions. Otherwise, if x_j has the quantifier type opposite to Q, the strategy function of any variable with quantifier type Q considered later may use x_j. Accordingly (lines 26–27), we add clauses $(x_j \vee \neg x'_j)$ and $(\neg x_j \vee x'_j)$ to φ' through the incremental interface of the SAT solver. This has two effects: first, it enforces equivalence of x_j and x'_j, and second, x_j is added to the common vocabulary of φ and φ', so that it can appear in interpolants computed in later iterations.[5]

Soundness of Algorithm 1 as stated in the following proposition can be proved by a straightforward induction on the quantifier prefix using Theorem 2 and Proposition 1.

Proposition 2. *Given a quantified Boolean formula Φ and a quantifier type $Q \in \{\forall, \exists\}$, Algorithm 1 terminates with a (possibly empty) set $\{(x_1, I_1) \ldots (x_k, I_k)\}$ of pairs (x_i, I_i) such that I_i represents the unique strategy function of x_i in Φ and $var(x_i) \in var_Q(\Phi)$ for $1 \leq i \leq k$.*

Example 2. Consider the QBF $\Psi = \forall x_1 \exists y_1 \forall x_2 \exists y_2 . \varphi$, where

$$\varphi = (x_1 \vee y_1) \wedge (\neg x_1 \vee \neg y_1) \wedge (x_2 \vee y_2) \wedge (\neg x_2 \vee \neg y_2).$$

[4] Two distinct selector variables are required to ensure that they do not belong to the common variables of φ and φ'.

[5] One could also add these clauses to φ, in which case x'_j would become part of the shared vocabulary. This has the slight disadvantage that subsequently computed definitions may use a mixture of variables from φ and φ', rather than just φ.

Algorithm 1. Extracting Unique Strategy Functions by Interpolation

```
1: procedure getDefinitionsQBF(Φ, Q ∈ {∀, ∃})
2:     Q_1 x_1 ... Q_n x_n.φ ← Φ
3:     i = min{ 1 ≤ i ≤ n | Q_i = Q }
4:     shared ← {x_1, ..., x_{i−1}}
5:     if Q = ∀ then
6:         φ ← ¬φ                          ▷ ∀-strategies aim to falsify the matrix.
7:     end if
8:     φ' ← copy(φ, shared)
9:     sametype ← { j | 1 ≤ j ≤ n and Q_j = Q }
10:    for j ∈ sametype do
11:        s_j, s'_j ← fresh variables
12:        φ ← φ ∧ (¬s_j ∨ x_j)
13:        φ' ← φ' ∧ (¬s'_j ∨ ¬x'_j)
14:    end for
15:    solver ← ItpSatSolver(φ, φ')
16:    defined ← ∅
17:    k ← max{ i ≤ k ≤ n | Q_k = Q }
18:    for j = i, ..., k do
19:        if Q_j = Q then
20:            result ← solver.solve({s_j, s'_j})
21:            if result = UNSAT then
22:                I_j ← solver.getInterpolant()
23:                defined ← defined ∪ {(x_j, I_j)}
24:            end if
25:        else                                ▷ Q_j ≠ Q
26:            solver.addClause(φ', x_j ∨ ¬x'_j)
27:            solver.addClause(φ', ¬x_j ∨ x'_j)
28:        end if
29:    end for
30:    return defined
31: end procedure
```

We illustrate a run of Algorithm 1 on Ψ with $Q = \exists$. Since y_1 is the leftmost existential variable, we create a copy φ' of φ with every variable except x_1 renamed, that is,

$$\varphi' = (x_1 \vee y'_1) \wedge (\neg x_1 \vee \neg y'_1) \wedge (x'_2 \vee y'_2) \wedge (\neg x'_2 \vee \neg y'_2).$$

We also add the clauses $(\neg s_1 \vee y_1)$ and $(\neg s_2 \vee y_2)$ to φ and the clauses $(\neg s'_1 \vee \neg y'_1)$ and $(\neg s'_2 \vee \neg y'_2)$ to φ'. In the main loop, Algorithm 1 first checks whether $\varphi \wedge \varphi'$ is unsatisfiable under the assumptions $\{s_1, s'_1\}$. Unit propagation simplifies φ to (omitting unused selector variables and clauses)

$$(\neg x_1) \wedge (x_2 \vee y_2) \wedge (\neg x_2 \vee \neg y_2),$$

and φ' simplifies to

$$(x_1) \wedge (\neg x'_2 \vee y'_2) \wedge (\neg x'_2 \vee \neg y'_2).$$

By resolving $(\neg x_1)$ with (x_1) we obtain the empty clause, and $\neg x_1$ is the corresponding interpolant,[6] so $(y_1, \neg x_1)$ is added to the set of definitions. Next, we consider the universally quantified variable x_2 and add the clauses $(x_2 \vee \neg x_2')$ and $(\neg x_2 \vee x_2')$ to φ'. Finally, we check whether y_2 is definable by calling the SAT solver under the assumptions $\{s_2, s_2'\}$. Now, the formula φ simplifies to

$$(x_1 \vee y_1) \wedge (\neg x_1 \vee \neg y_1) \wedge (\neg x_2),$$

and φ' simplifies to

$$(x_1 \vee y_1') \wedge (\neg x_1 \vee \neg y_1') \wedge$$
$$(x_2') \wedge (x_2 \vee \neg x_2') \wedge (\neg x_2 \vee x_2').$$

Unit propagation derives the clause (x_2) from the clauses in the second line, which can be resolved with the clause $(\neg x_2)$ from φ to obtain a resolution refutation of the formula $\varphi \wedge \varphi'$, with $\neg x_2$ as an interpolant. Accordingly, $(y_2, \neg x_2)$ is added to the set of definitions. Algorithm 1 terminates with the definitions $\{(y_1, \neg x_1), (y_2, \neg x_2)\}$, and it is readily verified that $y_1 \equiv \neg x_1$, $y_2 \equiv \neg x_2$ is indeed the unique existential winning strategy of Ψ.

4.2 Improvements and Generalization to Dependency QBF

Consider a QBF $\Phi = \forall x_1, x_2 \exists y_1, y_2.(x_1 \leftrightarrow x_2) \leftrightarrow (y_1 \leftrightarrow y_2)$. It is easy to verify that Φ is true and that y_1 and y_2 do not have unique Skolem functions: for every assignment to the universal variables there are two ways of setting y_1 and y_2 so as to satisfy the matrix, so neither existential variable is defined by the universal variables alone. However, each variable is defined by all remaining variables. For instance, variable y_2 is defined by x_1, x_2, and y_1.

More generally, increasing the set of defining variables allows us to detect more definitions: if x is defined in terms of X then it is also defined in terms of any enclosing set $X' \supset X$. To exploit this, we modified Algorithm 1 so as to assume a total ordering of variables and check for definitions of a variable x in terms of *all* variables X which precede it in the quantifier prefix. This can be implemented by simply adding clauses encoding equivalence of x_j and x_j' (lines 26–27) regardless of quantifier type.

Technically, this leads to an alternative definition of a "winning strategy" for a QBF where each strategy function takes an assignment to all preceding variables as input. Both definitions are ultimately equivalent in the sense that a winning strategy according to one definition can be transformed into a winning strategy according to the other definition without changing its responses (cf. the work on quantifier elimination by functional composition and self-substitution [8,14,28,29]). One can prove an analogue of Proposition 1 stating that the strategy function—according to the alternative definition—of a variable x is unique whenever x is defined in terms of the variables preceding x in the quantifier prefix.

[6] As mentioned above, interpolants can be efficiently extracted from resolution refutations [32,36,46].

Dependency Quantified Boolean Formulas (DQBFs) generalize QBFs by allowing a non-linear quantifier prefix. More specifically, each existential variable is annotated with a set of universal variables its Skolem function may depend on. A DQBF is true if there is an existential winning strategy such that each Skolem function satisfies these restrictions [2]. Although evaluating DQBF is NEXPTIME-complete and thus believed to be much harder than evaluating QBF, the fact that problems can be concisely encoded in DQBF [12,18] has prompted the development of dedicated DQBF solvers [13,15,17,48].

Algorithm 1 can easily be extended to compute unique Skolem functions of DQBF. The standard DQDIMACS format [15] allows for the combination of a linear quantifier prefix with variables for which the dependency sets are explicitly stated. The linear quantifier prefix can be handled as before. For each existential variable x with explicit dependency set D_x we simply check whether x is defined by D_x. If multiple variables $x_1, \ldots, x_k$ have the same dependency set D_x (which is frequently the case in benchmark formulas) we check whether x_i is defined by $D_x \cup \{x_1, \ldots, x_{i-1}\}$ for each $1 \leq i \leq k$. Again, this technically requires a non-standard definition of Skolem functions for DQBF but can easily be proven sound.

5 Implementation

We implemented the algorithm described in the previous section in a prototype named Unique. As a back end SAT solver we use ItpMiniSat, a modified version of MiniSat [11] bundled with the ExtAvy model checker that efficiently generates interpolants in memory and supports both assumptions and incremental solving [22,49]. Unique can read PCNF formulas (QDIMACS), prenex non-CNF QBFs (QCIR), as well as DQBFs with CNF matrices (DQDIMACS).

Interpolants obtained from ItpMiniSat are represented as And-Inverter graphs (AIGs) and accessed through the AIG library of ABC [7]. To make use of the structural sharing capabilities of AIGs, we maintain a single AIG representing the interpolants computed in the main loop (lines 18–29) of Algorithm 1. Whenever a new interpolant is obtained, the corresponding AIG returned by ItpMiniSat is merged into the existing AIG. If the number of AIG nodes exceeds a (geometrically increasing) threshold, we use the ABC macro *compress2* to reduce the size of the combined AIG. Upon termination, and assuming the AIG is not too large, this is followed up by a round of *FRAIGing* [37] and a final application of *compress2*.

While running Unique on QBFs with multiple quantifier alternations we noticed that ItpMiniSat got stuck attempting to solve some of the definability queries. Further testing revealed that the corresponding instances were hard for most state-of-the-art solvers. Increasing the overall timeout would allow us to solve these instances in some cases, but naturally the corresponding interpolants (for unsatisfiable instances) were very large (and difficult to compress with ABC). This clearly defeats the purpose of detecting unique strategy functions quickly. We thus decided to impose a limit on the number of conflicts

for each call of ItpMiniSAT (currently set to 1000 conflicts). This significantly reduces the overall running time of Unique for many instances and ensures that individual interpolants are small, but only marginally decreases the total number of definitions found.

Since the individual definability queries are independent of each other, it is not necessary to determine for each input variable whether it is defined. Accordingly, we implemented Unique as an *anytime* algorithm: upon termination, it returns the set of variables with unique strategy functions identified up to that point, along with the AIG representing the corresponding functions.

6 Experiments

For the experiments described below we used a cluster with Intel Xeon E5649 processors at 2.53 GHz running 64-bit Linux.

6.1 Gate Extraction

We first ran Unique to compute unique strategy functions for the instances in the 2QBF (402 instances) benchmark set from the 2018 QBF Evaluation, as well as the PCNF (558), QCIR (341), and DQBF (333) benchmark sets from the 2019 QBF Evaluation.[7] For each job we imposed a time limit of 600 s and a memory limit of 1.8 GB.

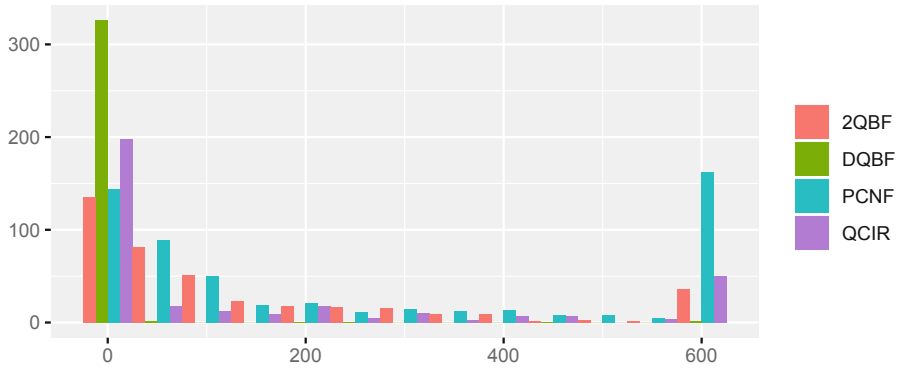


Fig. 1. Running time (s) of Unique by benchmark set. For each 50-s interval within the time limit (x-axis), the number of instances (y-axis) processed by Unique with a running time in that interval is shown.

Figure 1 shows a histogram for the running time of Unique on different benchmark sets. While most instances are processed quickly, Unique runs into

[7] http://www.qbflib.org.

the time limit for a significant number of PCNF instances. Generally, the running time increases with the size of the matrix and the number of variables. This explains why almost all DQBF formulas are processed quickly, as these tend to be much smaller compared to formulas from the other benchmark sets.

Figure 2 shows a histogram for the fraction of existential variables with unique strategy functions in 2QBF and PCNF instances (turquoise bars). We clearly see a bimodal distribution here: there is a large number of instances where the strategy functions of most variables are unique, but also a significant number of instances where few existential strategy functions are unique. To determine how many of the corresponding definitions cannot be found by syntactic gate detection, we used the QCIR-conv script provided by GhostQ [31] to convert 2QBF and PCNF instances to QCIR, and ran Unique again on the resulting circuits. To do this, the circuit is translated (back) to CNF, but auxiliary variables representing gates are ignored by the definability check. Testing showed that a one-sided CNF encoding [42] works better than standard Tseitin conversion.

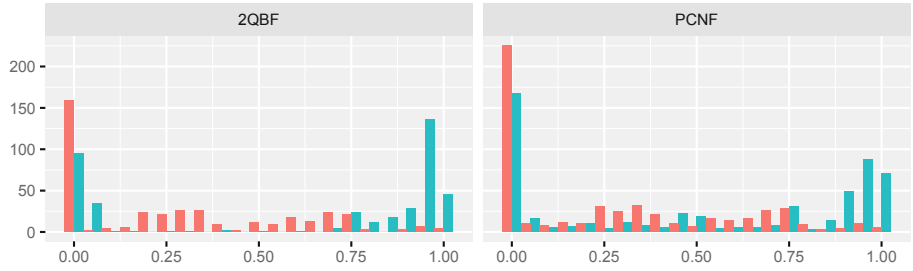


Fig. 2. Fraction of existential variables with unique strategy functions in 2QBF (left) and PCNF (right) instances before (turquoise) and after (red) syntactic gate detection. For each fraction (x-axis) we see the number of instances (y-axis) with the corresponding fraction of unique existential strategy functions. (Color figure online)

Table 1 (left) shows quartiles for the distributions of unique existential strategy functions detected by Unique in each benchmark set.[8] We only show the distribution for existential variables in Table 1 and Fig. 2 since very few universal variables were found to have unique strategy functions. In fact, only 51 instances from the QCIR benchmark set encoding bounded synthesis for Petri games contained such universal variables.

The fraction of variables with unique strategy functions was smallest for QCIR instances. This is expected, since they can represent circuit structure directly and do not require auxiliary variables to encode gate definitions. By

[8] For instance, the left side of the first row of Table 1 says that for 75% of 2QBF instances, Unique was able to identify 3% of Skolem functions as unique; for half of the instances, at least 90% of existential variables were identified as having unique Skolem functions; and for 25% of instances, at least 96%.

Table 1. Distribution (quartiles) of the fraction of unique Skolem functions identified by UNIQUE before (left) and after (right) preprocessing with HQSPRE. Rows marked by a star (*) show the distribution after syntactic gate detection.

	Original			Preprocessed		
	1st	Median	3rd	1st	Median	3rd
2QBF	0.03	0.9	0.96	0	0	0
2QBF*	0	0.22	0.54	0	0	0
PCNF	0	0.53	0.94	0	0	0.03
PCNF*	0	0.21	0.53	0	0	0.02
QCIR	0	0	0.13	–	–	–
DQBF	0.57	0.88	0.94	0	0.22	0.45

contrast, 2QBF and DQBF instances contain many variables with unique strategy functions. For about half of the instances, between roughly 90% and 95% of the existential strategy functions are unique.

On the right of Table 1 we show the distribution of unique existential strategy functions after preprocessing with HQSPRE [52]. Clearly, only very few unique Skolem functions are detected by UNIQUE. This may be in part due to the fact that preprocessing detects and removes gate definitions [27]. Another possibility is that definitions are simply lost: some of the most powerful preprocessing techniques for QBF currently used only preserve the truth value and not the set of strategies [23]. We will return to this topic at the end of the next subsection.

6.2 Solving Formulas Augmented with Definitions

Unique strategy functions of a (D)QBF can be substituted for their variables without changing the set of winning strategies. This can be used in preprocessing to reduce the number of quantified variables, typically at the cost of increasing the size of the matrix. In the following experiments, we substituted definitions found by UNIQUE for the defined variables and ran QBF and DQBF solvers on the resulting instances.

First, we considered the 2QBF benchmark set. We picked the QCIR solvers QUABS [47], QFUN [26], and GHOSTQ [31], along with the dedicated 2QBF (PCNF) solver CADET [43]. For the QCIR solvers, the performance on instances constructed by syntactic gate detection with QCIR-CONV serves as a baseline. We compare it with performance on instances obtained by UNIQUE and—since QCIR-CONV also performs circuit-level simplifications that go beyond gate extraction—with a combination of both where QCIR-CONV and UNIQUE are run in sequence.

For CADET, we compare performance on the original 2QBF instances with performance on QDIMACS instances augmented with CNF encodings of definitions extracted by UNIQUE. For each configuration, we report the number of

instances solved within a time limit of 15 min. To isolate the effect of adding definitions, the time required by UNIQUE (and QCIR-CONV) is not counted towards the time limit.[9] The results are shown in Fig. 3 (left).

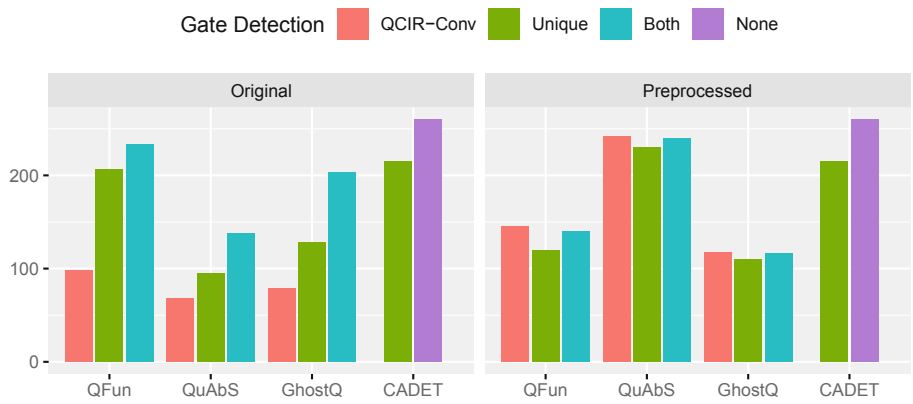


Fig. 3. Number of 2QBF instances solved (y-axis) by solvers (x-axis) using different gate detection methods before (left) and after (right) preprocessing with HQSPRE.

QFUN, QUABS, and GHOSTQ benefit considerably from semantic gate extraction, in particular when applied on top of syntactic gate extraction. By contrast, CADET solves *fewer* instances augmented with gate definitions than original instances. We found this surprising, since variable definitions should be detected by CADET's heuristic for identifying unique Skolem functions. Perhaps most definitions found by UNIQUE are already covered in this way, so that the additional clauses simply slow down propagation. We believe that explicitly telling CADET which variables have already been identified as determined should result in a speedup overall.

Figure 4 takes a closer look at solving times for individual instances (for this plot, memory outs are treated as timeouts). CADET is slower on instances augmented by UNIQUE but fairly consistent, while the effect on the other solvers is more erratic. We conjecture that this is because the set of existential strategies is preserved and the instances thus "look similar" to CADET.

Next, we tested with PCNF instances and considered the QDIMACS solvers DEPQBF [5] and CAQE [44], as well as the QCIR solvers QUABS [47], QFUN [26], and QUTE [40]. Again, we compare the number of instances solved in 15 min with different options for gate detection. Results are shown in Fig. 5 (left). Again all QCIR solvers benefit from gate detection with UNIQUE when performed on top of syntactic gate detection with QCIR-CONV, while performance

[9] The results are qualitatively the same when the running time of UNIQUE is counted towards the time limit: the largest decrease in the number of solved instances across all benchmark sets and configurations is 7.

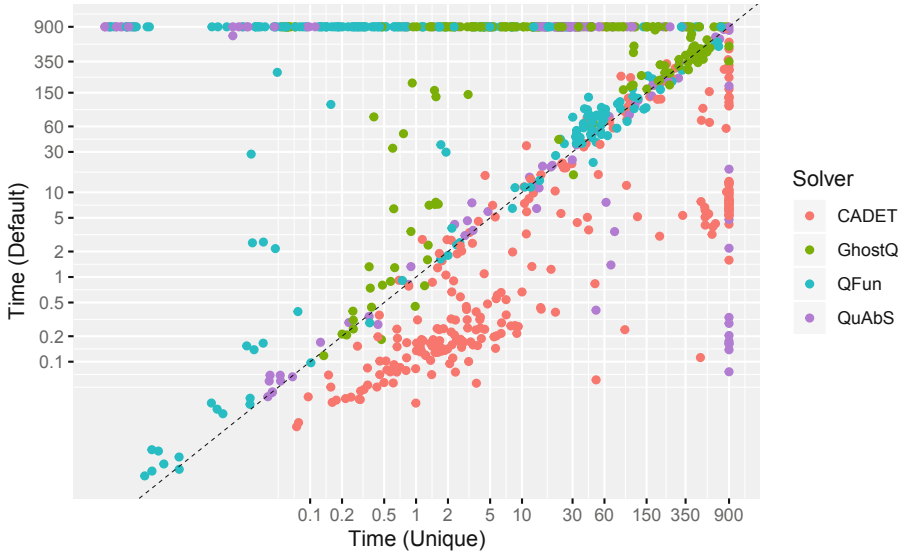


Fig. 4. Solving time (s) for 2QBF instances with (x-axis) and without UNIQUE (y-axis).

decreases for both QDIMACS solvers. The additional clauses and variables introduced by UNIQUE apparently do not help these solvers and simply result in a slowdown.

Finally, we tested the impact of UNIQUE on DQBF (DQDIMACS) instances solved by HQS [19] and DCAQE [48] within 15 min. Since DQBF solvers currently do not (yet) support non-CNF input, we translate definitions to CNF and add them to the original formulas. Note that whenever an existential variable x is defined by (a subset of) its dependency set, we can safely let x depend on additional variables. This is sound since the response of variable x is already determined by the variables in the original dependency set and cannot change depending on other inputs. In particular, we can collect all defined variables (and auxiliary variables) in an "innermost" existential quantifier block that depends on all universal variables. Since many existential variables have uniquely determined strategy functions (see Table 1), this allows us to push many variables into the innermost quantifier block and get closer to a linear quantifier prefix. For HQS, this translates into a small increase in the number of solved instances (208 vs. 189), whereas DCAQE basically solves the same number of instances (133 vs. 135).

Interaction with Preprocessing. QBF solvers for PCNF are typically paired with preprocessors such as BLOQQER [6] or HQSPRE [52]. These are highly engineered tools that batter instances with a barrage of techniques and can often solve formulas completely on their own. Most solvers benefit greatly from

Fig. 5. PCNF instances solved (y-axis) by solver (x-axis) using different methods for gate detection before (left) and after (right) preprocessing with HQSPre.

preprocessing. This is evident in Fig. 5 (right), which shows the number of solved PCNF instances with different forms of gate detection after preprocessing with HQSPre (within a timeout of 600 s). Here, the number of solved instances increases significantly for almost all systems.

At the same time, preprocessing appears to obscure or destroy definitions. Unique hardly finds any definitions in preprocessed instances (cf. Table 1) and accordingly has little impact on performance. For QFun, which benefitted most from gate detection in our experiments, this translates to a substantial reduction in the number of solved instances. On the 2QBF benchmark set (Fig. 3), both QFun and GhostQ solve significantly fewer instances with HQSPre compared to the combination of Unique and QCIR-Conv, whereas the number of solved instances almost doubles for QuAbS. Understanding which preprocessing techniques obscure gate definitions and why certain solvers benefit more from gate detection than others are important questions for future work.[10]

7 Related Work

Our semantic gate detection technique is closely related to a method for *determinizing* Boolean relations by Jiang et al. [29], a problem that essentially corresponds to solving 2QBF. The authors show that, for a (total) relation $R(X, y)$ with a single output variable y, a functional implementation of y can be obtained as an interpolant for $\neg R(X, 0) \wedge \neg R(X, 1)$. This can be used to determinize

[10] We also ran experiments with QCIR-conv and Unique applied *before* preprocessing. The results were significantly worse, so we do not report them in detail. Standard preprocessing requires PCNF input, so that definitions have to be encoded using additional clauses and Tseitin variables. Just like the PCNF solvers in the other experiments, HQSPre appears to be unable to do anything useful with these extra clauses and variables.

relations $R(X, Y)$ with a set of output variables $Y = \{y_1, \ldots, y_n\}$. First, an implementation f_n for y_n can be computed by treating R as a relation with inputs $X \cup \{y_1, \ldots, y_{n-1}\}$ and single output y_n. Subsequently, the implementation f_n can be substituted for y_n to obtain a relation $R'(X, Y \setminus \{y_n\})$. By repeating this process, a functional implementation f_1 of y_1 can eventually be obtained. Substituting f_i into f_{i+1} for $1 \leq i < n$ results in functional implementations that only depend on the original input variables X. This approach does not require for any of the output variables to be defined by X, but an implementation of y_i solely in terms of the input variables X is only available at the very end of this process. For *deterministic* relations $R(X, Y)$ (where every y is defined in terms of X), the authors show that a functional implementation of $y \in Y$ can be obtained as the interpolant of a formula that corresponds to the formula in the statement of Padoa's theorem. Our result stated as Theorem 2 is more general in that it holds for multi-output relations that are not necessarily deterministic.

Hofferek et al. use interpolation to synthesize multiple functional implementations from a single proof and thus avoid the increase in formula size incurred by repeated substitution [24]. This has an analogue in *strategy extraction* for QBF, which allows for implementations of all (existential or universal) variables to be obtained from a proof [3]. However, strategy extraction requires the input QBF has been solved, whereas our main interest is in preprocessing QBF.

There is a series of works on recovering gate definitions from CNF formulas. Li integrated rules for detecting equivalent literals in a Davis-Putnam style algorithm [35]. Ostrowski et al. represent formulas as graphs to detect patterns corresponding to and-gates, or-gates, and equivalences [38]. Roy et al. use CNF signatures to detect a richer set of gates [45]. Fu and Malik extend this to arbitrary (user-specified) gate libraries and ensure that a maximum acyclic circuit is constructed [16].

In the context of QBF, Bacchus and Goultiaeva showed that circuit reconstruction can speed up solvers by providing them with a better set of initial cubes [21]. They also extended the scope of these techniques to CNF formulas obtained from circuits by the Plaisted-Greenbaum encoding [42]. Scholl and Pigorsch developed a QBF solver that manipulates an AIG representation of the matrix to perform quantifier elimination and relies on circuit reconstruction to simplify the initial AIG [41].

Balabanov et al. proposed a SAT-based semantic gate extraction technique [4]. Their approach has the disadvantage that a subset of clauses inducing a definition has to be guessed. As a more efficient heuristic, they suggest to identify *pseudo* definitions instead. A set of clauses $(A_1 \vee x), \ldots, (A_k \vee x), (B_1 \vee \neg x), \ldots, (B_l \vee \neg x)$ is a pseudo definition of x if the formula $A_1 \wedge \cdots \wedge A_k \wedge B_1 \wedge \cdots \wedge B_l$ is unsatisfiable. Rabe and Seshia use a similar criterion in their *incremental determinization* algorithm to identify variables that are *(locally) deterministic* [43]. Checking for pseudo definitions is typically efficient but limits the range of definitions that can be detected.

8 Conclusion

Syntactic gate detection has been shown to benefit SAT solvers [10,16] and QBF solvers [21]. The underlying algorithms are fast but limited to a predefined library of gates. By contrast, our semantic gate extraction method can detect any definition entailed by an input formula but requires an interpolating SAT solver. In the context of SAT, this overhead likely outweighs any potential benefits. However—as demonstrated by our experiments—there is significant potential for application to harder problems such as QBF and DQBF evaluation. Here, preprocessing is just a first step.

At the same time, our results show that substituting unique strategy functions can slow down solvers. In some sense, this is counter-intuitive: ideally, providing solvers with unique strategy functions should give them a head start, or at least not hurt their performance. By analogy, if we give a SAT solver part of a backbone assignment, it can simply instantiate accordingly and need not consider the corresponding variables for the remainder of its run. With the exception of CADET, QBF solvers currently cannot "instantiate" variables with strategy functions in this way, since they are only equipped to reason about assignments. We believe that designing techniques for reasoning about *strategies* is a key challenge in developing the next generation of QBF solvers.

Acknowledgements. The author would like to thank Adrian Rebola-Pardo, Matthias Schlaipfer, and Georg Weissenbacher for helpful discussions.

References

1. Ansótegui, C., Gomes, C.P., Selman, B.: The Achilles' heel of QBF. In: Veloso, M.M., Kambhampati, S. (eds.) AAAI 2005, pp. 275–281. AAAI Press/The MIT Press (2005)
2. Balabanov, V., Chiang, H.J.K., Jiang, J.R.: Henkin quantifiers and Boolean formulae: a certification perspective of DQBF. Theor. Comput. Sci. **523**, 86–100 (2014)
3. Balabanov, V., Jiang, J.R.: Unified QBF certification and its applications. Formal Methods Syst. Des. **41**(1), 45–65 (2012)
4. Balabanov, V., Jiang, J.R., Mishchenko, A., Scholl, C.: Clauses versus gates in CEGAR-based 2QBF solving. In: Darwiche, A. (ed.) Beyond NP, Papers from the 2016 AAAI Workshop, AAAI Workshops, vol. WS-16-05. AAAI Press (2016)
5. Lonsing, F., Biere, A.: Integrating dependency schemes in search-based QBF solvers. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 158–171. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14186-7_14
6. Biere, A., Lonsing, F., Seidl, M.: Blocked clause elimination for QBF. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 101–115. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_10
7. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5

8. Bubeck, U., Kleine Büning, H.: Nested boolean functions as models for quantified boolean formulas. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 267–275. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39071-5_20
9. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symb. Log. **22**(3), 269–285 (1957)
10. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005). https://doi.org/10.1007/11499107_5
11. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
12. Faymonville, P., Finkbeiner, B., Rabe, M.N., Tentrup, L.: Encodings of bounded synthesis. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 354–370. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_20
13. Finkbeiner, B., Tentrup, L.: Fast DQBF refutation. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 243–251. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_19
14. Fried, D., Tabajara, L.M., Vardi, M.Y.: BDD-based boolean functional synthesis. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 402–421. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_22
15. Fröhlich, A., Kovásznai, G., Biere, A., Veith, H.: iDQ: Instantiation-based DQBF solving. In: Pragmatics of SAT 2014 (2014)
16. Fu, Z., Malik, S.: Extracting logic circuit structure from conjunctive normal form descriptions. In: VLSI Design 2007, (ICES 2007), pp. 37–42. IEEE Computer Society (2007)
17. Ge-Ernst, A., Scholl, C., Wimmer, R.: Localizing quantifiers for DQBF. In: Barrett, C.W., Yang, J. (eds.) 2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, 22–25 October 2019, pp. 184–192. IEEE (2019)
18. Gitina, K., Reimer, S., Sauer, M., Wimmer, R., Scholl, C., Becker, B.: Equivalence checking of partial designs using dependency quantified Boolean formulae. In: 31st International Conference on Computer Design, ICCD 2013, pp. 396–403. IEEE (2013)
19. Gitina, K., Wimmer, R., Reimer, S., Sauer, M., Scholl, C., Becker, B.: Solving DQBF through quantifier elimination. In: Nebel, W., Atienza, D. (eds.) DATE 2015, pp. 1617–1622. ACM (2015)
20. Gomes, C.P., Kautz, H., Sabharwal, A., Selman, B.: Satisfiability solvers. In: Handbook of Knowledge Representation, Foundations of Artificial Intelligence, vol. 3, pp. 89–134. Elsevier (2008)
21. Goultiaeva, A., Bacchus, F.: Recovering and utilizing partial duality in QBF. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 83–99. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39071-5_8
22. Gurfinkel, A., Vizel, Y.: Druping for interpolates. In: FMCAD 2014, pp. 99–106. IEEE (2014)
23. Heule, M., Järvisalo, M., Lonsing, F., Seidl, M., Biere, A.: Clause elimination for SAT and QSAT. J. Artif. Intell. Res. **53**, 127–168 (2015)
24. Hofferek, G., Gupta, A., Könighofer, B., Jiang, J.R., Bloem, R.: Synthesizing multiple Boolean functions using interpolation on a single proof. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 77–84. IEEE (2013)

25. Ivrii, A., Malik, S., Meel, K.S., Vardi, M.Y.: On computing minimal independent support and its applications to sampling and counting. Constraints **21**(1), 41–58 (2016)
26. Janota, M.: Towards generalization in QBF solving via machine learning. In: McIlraith, S.A., Weinberger, K.Q. (eds.) AAAI-18, pp. 6607–6614. AAAI Press (2018)
27. Järvisalo, M., Biere, A., Heule, M.: Simulating circuit-level simplifications on CNF. J. Autom. Reasoning **49**(4), 583–619 (2012)
28. Jiang, J.-H.R.: Quantifier elimination via functional composition. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 383–397. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_30
29. Jiang, J.R., Lin, H., Hung, W.: Interpolating functions from large Boolean relations. In: Roychowdhury, J.S. (ed.) ICCAD 2009, pp. 779–784. ACM (2009)
30. Jordan, C., Klieber, W., Seidl, M.: Non-CNF QBF solving with QCIR. In: Darwiche, A. (ed.) Beyond NP, Papers from the 2016 AAAI Workshop. AAAI Workshops, vol. WS-16-05. AAAI Press (2016)
31. Klieber, W., Sapra, S., Gao, S., Clarke, E.: A non-prenex, non-clausal QBF solver with game-state learning. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 128–142. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14186-7_12
32. Krajícek, J.: Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic. J. Symb. Log. **62**(2), 457–486 (1997)
33. Lagniez, J., Lonca, E., Marquis, P.: Improving model counting by leveraging definability. In: Kambhampati, S. (ed.) IJCAI 2016, pp. 751–757. IJCAI/AAAI Press (2016)
34. Lang, J., Marquis, P.: On propositional definability. Artif. Intell. **172**(8–9), 991–1017 (2008)
35. Li, C.M.: Integrating equivalency reasoning into Davis-Putnam procedure. In: Kautz, H.A., Porter, B.W. (eds.) Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA, pp. 291–296. AAAI Press/The MIT Press (2000)
36. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_1
37. Mishchenko, A., Chatterjee, S., Brayton, R.: FRAIGs: A unifying representation for logic synthesis and verification. Technical report, Berkeley (2005)
38. Ostrowski, R., Grégoire, É., Mazure, B., Saïs, L.: Recovering and exploiting structural knowledge from CNF formulas. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 185–199. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46135-3_13
39. Padoa, A.: Essai d'une théorie algébrique des nombres entiers, précédé d'une Introduction logique à une théorie déductive quelconque. Bibliothèque du Congrès International de Philosophie (1903)
40. Peitl, T., Slivovsky, F., Szeider, S.: Dependency learning for QBF. J. Artif. Intell. Res. **65**, 180–208 (2019)
41. Pigorsch, F., Scholl, C.: Exploiting structure in an AIG based QBF solver. In: Benini, L., Micheli, G.D., Al-Hashimi, B.M., Müller, W. (eds.) DATE 2009, pp. 1596–1601. IEEE (2009)
42. Plaisted, D.A., Greenbaum, S.: A structure-preserving clause form translation. J. Symb. Comput. **2**(3), 293–304 (1986)

43. Rabe, M.N., Seshia, S.A.: Incremental determinization. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 375–392. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_23
44. Rabe, M.N., Tentrup, L.: CAQE: A certifying QBF solver. In: Kaivola, R., Wahl, T. (eds.) FMCAD 2015, pp. 136–143. IEEE Computer Soc. (2015)
45. Roy, J., Markov, I., Bertacco, V.: Restoring circuit structure from SAT instances. In: Proceedings of International Workshop on Logic and Synthesis, pp. 663–678 (2004)
46. Schlaipfer, M., Weissenbacher, G.: Labelled interpolation systems for hyper-resolution, clausal, and local proofs. J. Autom. Reasoning **57**(1), 3–36 (2016)
47. Tentrup, L.: Non-prenex QBF solving using abstraction. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 393–401. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_24
48. Tentrup, L., Rabe, M.N.: Clausal abstraction for DQBF. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 388–405. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_27
49. Vizel, Y., Gurfinkel, A., Malik, S.: Fast interpolating BMC. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 641–657. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_43
50. Vizel, Y., Weissenbacher, G., Malik, S.: Boolean satisfiability solvers and their applications in model checking. Proc. IEEE **103**(11), 2021–2035 (2015)
51. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_31
52. Wimmer, R., Reimer, S., Marin, P., Becker, B.: HQSpre – an effective preprocessor for QBF and DQBF. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 373–390. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_21

TarTar: A Timed Automata Repair Tool

Martin Kölbl[1(✉)], Stefan Leue[1(✉)], and Thomas Wies[2(✉)]

[1] University of Konstanz, Konstanz, Germany
Martin.Koelbl@uni-konstanz.de, Stefan.Leue@uni-konstanz.de
[2] New York University, New York, USA
wies@cs.nyu.edu

Abstract. We present TarTar, an automatic repair analysis tool that, given a timed diagnostic trace (TDT) obtained during the model checking of a timed automaton model, suggests possible syntactic repairs of the analyzed model. The suggested repairs include modified values for clock bounds in location invariants and transition guards, adding or removing clock resets, etc. The proposed repairs guarantee that the given TDT is no longer feasible in the repaired model, while preserving the overall functional behavior of the system. We give insights into the design and architecture of TarTar, and show that it can successfully repair 69% of the seeded errors in system models taken from a diverse suite of case studies.

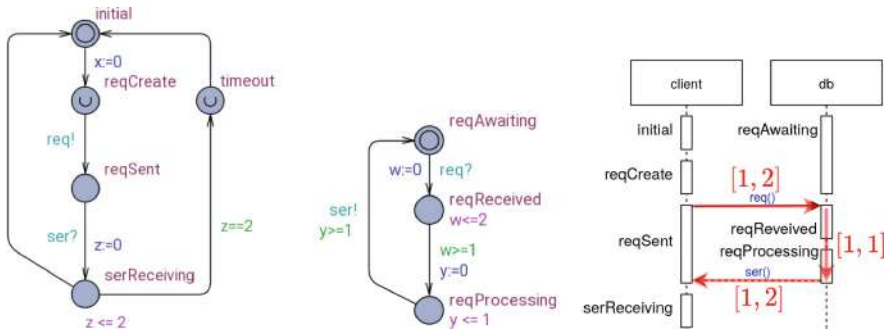
1 Introduction

A reactive system with requirements pertaining to its timing behavior is often modeled as a network of timed automata (NTA) [BY03]. Whether a timing requirement holds in an NTA can be analyzed by timed model checkers such as Uppaal [BLL+95] or opaal [DHJ+11]. In case of a requirement violation, a model checker returns a timed counterexample, also called a timed diagnostic trace (TDT). Until now, developers must manually identify and correct such violations by analyzing the generated TDTs. It is therefore desirable to support this process by an automated tool set that not only determines whether timing requirements are met, but also proposes syntactic repairs of the NTA in case they are not.

In [KLW19] we presented an automated repair analysis that analyzes a TDT obtained from the violation of a timed safety property and returns syntactic repair suggestions that avoid the concrete executions of the TDT violating the property. The analysis performs an additional admissibility check ensuring that the repaired model is functionally equivalent with the original NTA, which means that no action traces are added or omitted by the repair.

To illustrate the repair analysis consider the NTA in Figs. 1(a) and (b). It describes a *client* that sends a request *req* to a database *db* and expects to receive a response *ser* within 4 time units after sending the request. The client contains a

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 529–540, 2020.
https://doi.org/10.1007/978-3-030-53288-8_25



(a) Timed Automata `client` (b) Timed Automata `db` (c) TDT `tdt`

Fig. 1. Network of timed automata - running example

clock x that measures the time delay between the request creation and the receiving of a response in location *serReceiving*. The NTA allows to execute a TDT that violates the property, illustrated as a sequence diagram with time intervals in Fig. 1(c). A time interval in the sequence diagram denotes the minimal and maximal time delay for the message transmission and processing times in *db*, respectively. The repair computation analyzes the TDT and produces several syntactic repairs to the NTA that avoid the property violation. In [KLW19], the computed repairs aim at the modification of clock bounds in location invariants and transition guards. An example of such a repair is to reduce the bound in the time constraint $w \leq 2$ from 2 to 1. The modified bound constrains the maximal transmit time of the *req* message so that the resulting NTA receives all responses within the expected time. This repair eliminates the problematic executions of the TDT in the original NTA without changing the functional behavior of the system, which is confirmed by an admissibility test defined in [KLW19]. However, in general, it may not be possible to repair the model using only clock bound alterations.

Contributions. We present TarTar [tar20], which extends the initial prototype implementation of the clock bound repair analysis presented in [KLW19] to a more comprehensive NTA repair tool. Specifically, the extended tool implements new analyses that can suggest a whole range of repairs in addition to clock bound variation, such as modifying comparison operators in constraints, clock references, clock resets, and location urgency. Examples of new repairs computed for the model in Fig. 1 are:

- Exchanging the comparison operator in the constraint $w \geq 1$ to $w < 1$ ensures that the time to send a request is below 1 time unit.
- An exchange of clock z in $z \leq 2$ with clock y restricts the time of processing and receiving the response to at most 2 time units.
- To reset the clock y on the previous transition instead ensures that the time for sending and processing the request is below 1 time unit.

– Making the location *serReceiving* urgent reduces the time to receive a response to 0.

We call a repair admissible if the repaired system is functionally equivalent to the unrepaired system. The repair analysis implemented in TarTar returns the complete set of admissible repairs.

The repair analysis combines concepts and algorithms from model checking, constraint solving, and automata theory. A real-time model checker is used to generate TDTs for a given NTA that violate a given timed safety property. TarTar translates the TDT into a linear real arithmetic constraint system. An SMT solver is used to compute a repair for the generated constraint system by solving a MaxSMT problem. An automata-based language equivalence test checks whether the repair is admissible in the NTA model. The collaboration between these subcomponents yields a complex tool architecture. We provide insights into the design and implementation of this architecture and the underlying infrastructure of supporting tools. We evaluate the new repair analyses by applying TarTar to a number of NTA models. We systematically inject different modifications in these correct models and compute repairs for the obtained faulty models, which results in at least one admissible repair for 69% of the TDTs.

Related Work. Other tools exist that compute repairs. The tool BugAssist [JM11] analyzes C-code by solving a MaxSMT problem. The tool ReAssert [DDG+11] checks a set of possible modification to repair broken unit tests. Angelix [MYR16], S3 [LCL+17] and SemFix [NQRC13] compute repairs by symbolic execution and constraint solving. SketchFix [HZWK18] is based on lazy candidate generation. All tools are not repairing broken time constraints. We are not aware of related work on tools for the repair of timed automata models. A more comprehensive overview of related work on automated repair is given in [LPR19]. A discussion of work related to the foundations of our repair analysis can be found in [KLW19].

2 New Types of Repair Analyses

The repair analysis presented in [KLW19] and implemented in the prototype version of TarTar encodes a TDT as a constraint system in linear real arithmetic. It computes syntactic correct modifications of the underlying NTA by introducing bound variation variables v. For example, possible bound modifications for a clock bound $x \leq 2$ are expressed by a modified clock bound $x \leq 2+v$. The repairs are computed by solving a partial SMT problem on the TDT constraint system, involving soft-assert constraints on the bound variation variables. No repair is computed whenever the soft assertion $v = 0$ holds, otherwise the computed value of v characterizes the repair. In the following we sketch the new types of repairs implemented in TarTar. For a more comprehensive description, which space limitations do not allow us to provide here, we refer to [KLW20].

Operator Variation Repair Analysis. This analysis is motivated by the assumption that a wrong comparison operator in a location invariant or transition guard may cause a property violation. We assume for the repair encoding that the operators $\sim$ are indexed according to their order in the sequence $\langle <, \leq, =, \geq, > \rangle$. The possible repairs are encoded by a fresh variation variable v_i^{ov} where the value of v_i^{ov} is the index of the corresponding comparison operator. If $x < 4$ is computed as a repair, then $v_i^{ov} = 1$. Using this repair analysis, TarTar finds two admissible repairs for the example in Figs. 1(a) and (b) that replace the comparison operator in the clock constraint $w >= 1$ by $<$ or $<=$, respectively.

Clock Reference Repair Analysis. This analysis aims to repair property violations resulting from errors that stem from the unintended use of a wrong clock variable. We enumerate all the positions of clock variables in clock bound constraints using index i and all clock variables using index k. We then introduce for every position i, a fresh variation variable v_i^{cv} whose value k indicates the clock c_k to be used at that position in the repaired model. For example, if $y \leq 2$ is a repaired constraint, where the position of y in the constraint has index 3 and clock y has index 1, then $v_3^{cv} = 1$. Applying this repair analysis to the examples in Figs. 1(a) and (b), TarTar finds 13 admissible clock reference modification repairs, each involving two modifications. Nine repairs exchange y in the constraints $y \leq 1$ and $y \geq 1$ by a selection from the set of clocks z, x and w. Four repairs exchange y in the constraint $y \leq 1$ by w or x, and w in the constraint $w \geq 1$ by y or z.

Reset Clock Repair Analysis. This analysis aims to repair a property violation by adding or removing clock resets. We introduce a variation variable $v_{i,j}^{rv}$ for each clock c_i and the transition leaving location λ_j in the TDT. The reset status in the extended constraint system is inverted when $v_{i,j}^{rv} \neq 0$: if c_i was not reset before, it will now be reset, and vice versa. Applying the reset repair analysis to the examples in Figs. 1(a) and (b), TarTar finds four admissible repairs. One repair removes the reset of clock y, another removes the reset of clock z and two repairs add a reset of clock x either on the transitions towards the state *reqProcessing* or the transition towards the state *serReceiving*.

Urgent Location Repair Analysis. This analysis aims to repair cases where a faulty usage of urgent locations, which are always left with zero delay after entering, causes a property violation. Urgency of a location is modeled in the TDT constraint system by setting the location delay δ_j to 0. We define a fresh variation variable v_i^{uv} for a location λ_j. For $v_i^{uv} \neq 0$, the urgency for a location λ_j is inverted. Applying the urgency location repair analysis to the examples in Figs. 1(a) and (b), TarTar finds two inadmissible repairs. The first one makes the state *reqAwaiting* urgent, and another repair makes the state *serReceiving* urgent.

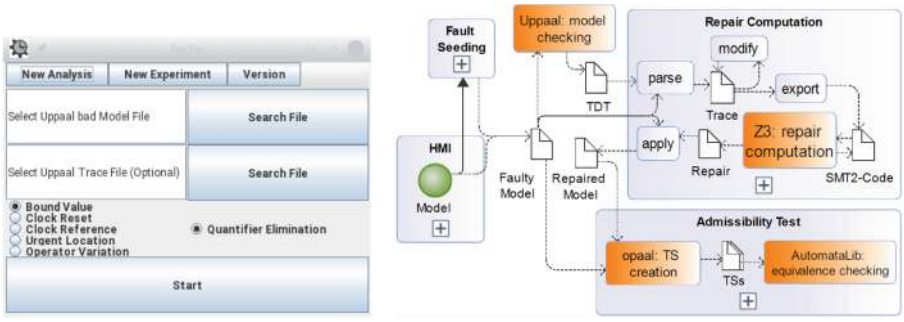
3 Usage of TarTar

We have implemented all repair analyses described in [KLW19] and in this paper in a tool named TarTar. It provides a graphical user interface, a command-

line interface and a web-interface which enables the execution of this resource intensive software on compute servers. A user selects one of these interfaces via arguments provided when invoking the Java library implementing TarTar. For real-time model checking, TarTar relies on Uppaal.

- The argument *–web* launches the web server and corresponding interface.
- Any other arguments launches the command-line mode. When using the argument *–help*, the command-line console prints some help information.
- When no arguments are given, the graphical user interface depicted in Fig. 2(a) is launched. The interface offers three tabs. *New Analysis* starts a repair analysis, *New Experiment* starts fault seeding which is described later in Sect. 5, and *Version* shows the current version number of TarTar.

All tool interfaces expect the same types of inputs in order to start a TarTar analysis run. The user specifies a file containing the Uppaal model as input and selects the kind of repair to compute. Optionally, a file with a TDT of the given Uppaal model can be specified. When no TDT is provided, TarTar automatically calls Uppaal to compute a TDT. The result of an analysis is one repaired model file for every computed repair, as well as a text file that summarizes which repairs are admissible.



(a) TarTar GUI (b) TarTar Architecture

Fig. 2. TarTar tool

4 Software Architecture and Implementation of TarTar

The software architecture of TarTar is depicted in Fig. 2(b). The orange rectangles in the figure represent external tools that TarTar calls in the course of the repair analysis. Uppaal is a state-of-the-art and closed-source model checking tool, which TarTar uses to compute a TDT for a given model and property. The SMT solver Z3 [dMB08] is used to solve the generated partial MaxSMT problems. To check the admissibility of a repair, TarTar uses opaal and the AutomataLib component of LearnLib [IHS15] since they conveniently provide functionality used during admissibility checking.

Data Flow Architecture. TarTar consists of many computation steps. For example, a TDT is parsed internally and stored as a Trace. This Trace is then modified and exported as SMT-LIB2 [BFT17] code. We define a computation step of TarTar as the computation transforming input into result artifacts. This focus on artifacts ensures a highly cohesive architecture and clear interfaces between any two computation steps. Computation steps with identical objectives are grouped into a project. This results in four projects depicted by blue rectangles in Fig. 2(b).

- *HMI* denotes the user interfaces of TarTar. The user inputs a timed model. TarTar then calls the project *Repair Computation* using a faulty timed model as a parameter. In case that the model is correct, TarTar calls the project *Fault Seeding*.
- *Fault Seeding* seeds faults into a correct model and then repairs the faulty model by computing repairs using *Repair Computation*. We use this analysis in Sect. 5 in order to benchmark the *Repair Computation* analyses.
- *Repair Computation* computes candidate repairs for a faulty timed model, applies these repairs to the model and finally automatically calls the *Admissibility Test*.
- *Admissibility Test* checks for every repaired model whether the computed repair is also admissible.

Control Flow Architecture. TarTar computes iteratively a set of repairs for a given faulty Uppaal model and a given property Π using the following steps:

0. *Counterexample Creation.* TarTar calls Uppaal to verify the model against Π. In case Π is violated, it stores a shortest symbolic TDT witnessing the violation in XML format.
1. *Diagnostic Trace Creation.* TarTar parses the model and the TDT into a data structure *Trace*. To add potential repairs, TarTar copies the trace and replaces the constraints that will potentially be subject to a repair by their modified variants. The modified trace is then translated to a logic constraint system, represented in SMT-LIB2 code.
2. *Repair Computation.* Z3 [dMB08] then solves a MaxSMT problem on the modified trace constraint system, computing a repair in which the number of unmodified constraints on the variation variables of type $v = 0$ is maximized. Since Z3 can solve a MaxSMT problem only for quantifier-free linear real arithmetic, TarTar first runs a quantifier elimination on the constraint system. It then solves the MaxSMT problem with soft constraints requiring $v = 0$ for all variation variables. For a more comprehensive presentation of this construction we refer the reader to [KLW20]. In case no solution is found, TarTar terminates. Otherwise, TarTar applies the repair to the faulty model and returns a repaired model.
3. *Admissibility Check.* TarTar checks the admissibility of a repair and compares the untimed languages of the faulty and repaired models. TarTar calls the model checker opaal in order to compute the timed transition systems (TTS) of the original and the repaired Uppaal model. We modified the

opaal model checker in such a way that it returns the TTS for a model. TarTar then checks whether the two TTS have equivalent untimed languages, in which case the repair is admissible. This check is implemented using the library AutomataLib. In case the two TTS are not equivalent, the admissibility test returns a trace as a witness for the difference.

4. *Iteration.* TarTar enumerates all repairs, i.e., all combinations of constraint modifications that correct the TDT. The repairs are iteratively enumerated starting with the ones that require the smallest number of modifications to the model. After a repair is computed, the combination of modified variables that has been found is prevented from being reconsidered for future repairs by setting these modification variables to 0 using hard asserts. TarTar then proceeds with attempting to compute further, previously unconsidered repairs.

Component Architecture. We implemented TarTar with the general infrastructure depicted in Fig. 3. The interface *Job* provides a general abstraction for an algorithm and specifies the necessary input and result values of the algorithm by the class *Description*. TarTar contains a *Job* for the projects *Fault Seeding*, *Repair Computations* and *Admissibility Test*. The class *Session* executes a *Job* and derivations of *Session* provide the different interfaces to the user. With this infrastructure, the analysis implementation in TarTar is independent from the implementation of the user interfaces, thus reducing coupling and improving modifiability of the code.

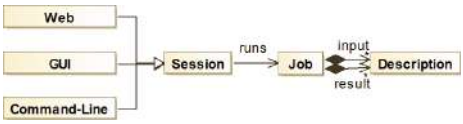


Fig. 3. TarTar component architecture

Implementation Details. We implemented the different projects that constitute TarTar in Java and use the build-management tool maven [Mav19] to manage the dependencies between the projects. TarTar interacts differently with the external tools that are needed for different purposes. It calls Uppaal via the command-line interface in order to generate a TDT and calls Z3 via its API to compute a repair. For the admissibility check, it calls opaal using a command-line script and the AutomataLib as an included Java library. For the implementation of the TarTar analyses the following two details are essential.

We modify constraints in an Uppaal model in order to apply a repair or to seed a fault. Since neither clock constraints nor transitions possess explicit unique identifiers in an Uppaal model, it is not obvious which constraint to change. We therefore uniquely identify a constraint by traversing the constraints in the sequence stored in the Uppaal model file and use the constraint index in this sequence as its identifier.

The complexity of the algorithms for solving quantifier elimination and the MaxSMT problem increase exponentially with the number of variables in the SMT model [KLW19]. We therefore reduce the number of variables by exploiting implied equality constraints. For example, a variable c_j is created for every

clock c in every step j of the TDT. We eliminate c_j explicitly before quantifier elimination by replacing it with the term $\sum_{i \in r..j} d_i$, where d_i is the time delay at step i in the trace and r is the last step before j where c was reset.

5 Evaluation

Evaluation Strategy. In order to evaluate the repair analyses both qualitatively and quantitatively, we need to synthesize a set of faulty timed automata. To the best of our knowledge, no benchmark suite for faulty timed automata exists. We therefore create faulty models by using the fault seeding strategy from [KLW19] which is motivated by ideas from mutation testing [JH11]. Mutation testing evaluates the quality of a test suite for a given program by systematically corrupting program code and determining the ratio of corruptions that the test suite is able to detect. We apply the same principle to evaluate the quality of our repair technique. As proposed in [KLW19], fault seeding modifies a single clock constraint so that the result is a set of models that violate a given property. During the seeding, the bound of a single clock constraint is modified by an amount of $\{-10, -1, +1, +0.1M, +M\}$, where M is the maximal clock bound occurring in a given model. Our observation was that making either small modifications that are close to the bound value or modifications in the order of the maximal bound value M often introduce actual errors in the model. We have extended fault seeding to the new types of repairs. In particular, fault seeding additionally exchanges the comparison operator in a clock constraint by $\{<, \leq, =, \geq, >\}$, swap a referenced clock with all other clocks occurring in the given model, modify the reset clocks of any transition, and switch for any location whether it is urgent. TarTar checks automatically whether a modified TA violates a given property. If this is the case, it performs all of the above defined repair analyses.

Results. We applied fault seeding to the models in [KLW19] and analyzed the obtained TDTs using the above described repair analyses implemented in TarTar. All analyses were performed on a computer with an i7-6700K CPU (4.00 GHz), 60 GB of RAM and a 64 bit Linux operating system. We summarize the results of the experiment per considered model (Table 1) and per type of considered repair (Table 2). Column *Sd* contains the count of seeded faults that result in a number $\#T$ of faulty models. T_{UP} is the maximal time that Uppaal needs to create a TDT for the faulty models, and the longest TDT has a length of *Ln*. TarTar computed for the TDTs overall a number $\#R$ repairs of which $\#A$ are admissible. An admissible repair is found for $\#S$ of the TDTs. The computation effort for a repair analysis is given by the time T_{QE} for successful quantifier elimination, the number of timeouts $\#O$ of quantifier eliminations after 10 min, the average time T_R to compute a repair and the memory consumption M_R. The constraint system that Z3 solves has the count $\#Vr$ of variables and $\#Cn$ of constraints. The effort for the admissibility check is given in time T_{Adm} and memory M_A. All times are given in seconds and memory consumption in MB. Notice that we omit the columns pertaining to the fault seeding and TDT computation in Table 2 as they are irrelevant here.

Table 1. Experimental results according to model.

Repair	#Sd	#T	T_{UP}	Ln	$\#R$	$\#A$	$\#S$	T_{QE}	#O	T_R	M_R	$\#Vr$	$\#Cn$	T_{Adm}	M_A
db rep.	110	13	0.016	4	229	138	9	89.346	2	0.911	14.53	30	91	2.080	45
csma	191	10	0.012	2	70	26	8	0.049	0	0.023	0.58	16	72	1.825	75
elevator	88	5	0.011	1	7	5	4	0.049	0	0.020	0.53	6	28	1.665	17
viking	310	9	0.015	18	9	7	5	86.539	21	1.436	20.07	120	180	1.952	543
bando	1,955	40	0.111	279	4,061	209	21	31.555	46	4.922	20.86	1,156	8,144	19.57	1251
Pacemaker	1,187	12	0.022	9	62	19	10	0.663	20	0.325	2.59	116	988	1.994	206
SBR	353	50	0.027	84	751	660	31	117.057	86	2.686	37.16	765	1,211	138.004	211
FDDI	314	36	0.014	11	166	105	34	29.859	51	3.074	9.70	116	272	2.241	128

Overall, TarTar seeded 4.508 faults. This resulted in 175 TDTs in total (60 TDTs due to bound modification, 72 due to operator variation, 27 due to changing the clock reference, 8 due to complementing the reset of clocks and 8 due to the switching of urgent locations). TarTar found 5,355 repairs, out of which 1,169 were admissible. It found at least one admissible repair for 122 of the TDTs. The maximal number of modified constraints in the admissible repairs computed for a single TDT using all types of analysis was 25.

Table 2. Experimental results according to type of repair.

Repair	$\#R$	$\#A$	$\#S$	T_{QE}	#O	T_R	M_R	$\#Vr$	$\#Cn$	T_{Adm}	M_A
Bound Modification	533	364	85	15.209	8	4.922	20.86	1,156	2,498	138.004	525
Operator Variation	3,929	96	51	117.057	44	2.686	37.16	996	8,144	59.117	543
Clock Reference	693	625	35	33.282	61	3.074	14.13	1,120	5,355	116.944	206
Reset Clock	45	37	13	89.346	113	0.911	14.53	996	2,836	2.051	45
Urgent Location	155	47	37	0.107	0	0.135	3.16	1,120	2,502	58.551	1,251

Interpretation. Few of the seeded faults resulted in a property violation. TarTar seeded 4.508 faults which led to 175 TDTs, thus only 3.9% of these faults result in a TDT. This supports the hypothesis that, in practice, often times only few time constraints have an impact on a property violation. TarTar computes at least one admissible repair by bound modification for 85 (48%) of the 175 TDTs, by operator variation for 51 (29%), by clock reference for 35 (20%), by clock reset for 13 (7%) and by urgent location for 37 (21%). Every analysis on its own computes less admissible repairs than the combination of all repair analyses, which solves 122 (69%) of the 175 TDTs. The largest number of modified constraints in all the admissible repairs for a single TDT was 25, which is less than anticipated. This low number of modified constraints infer that, for the examples that we considered, only a few constraints of each TDT combined to admissible repairs. The number of modified constraints determines the number of possible repairs that have an impact on whether a property is violated or not. Since it was observed in [KLW19] that the computational effort for the repair computation is largely determined by the quantifier elimination step, we expect that in light of

the observed 226 timeouts a more efficient quantifier elimination would lead to a significantly higher number of repairs. Furthermore, the number of timeouts, and thus the computation time needed for the repair, rises with the length of the analyzed TDT. The model *SBR* has the most timeouts with 86 and the third longest trace with a length of 84 steps. The model *bando* has the third most timeouts with 46 and the longest trace. Obviously, the longer the TDT, the larger the resulting constraint system, leading to increased computational effort. The *bando* model has the largest constraint system with $1,156$ variables and $8,144$ constraints. The *SBR* model has the second largest constraint system with 765 variables and $1,211$ constraints. The model *FDDI* has a shorter trace of length of 11 and a much smaller constraint system with 116 variables and 272 constraints. From this we conclude that the complexity of a repair depends not only on the trace length, but also on the intrinsic complexity of the model. Modifying states from urgent to non-urgent during fault seeding resulted in only 8 TDTs. This low number is due to the observation that the considered models contain only few urgent states. Modifying non-urgent states to urgent ones, however, did not lead to a single property violation resulting in a TDT. The rationale is that urgency ensures to leave a state immediately without a delay which leads to a restriction rather than a relaxation regarding the time budget spent along an execution trace. As a consequence, making a state urgent does not cause a property violation in many models since the type of the checked properties is typically time bounded reachability, and a restricted time budget does not make it more likely that the property is violated. We finally observe that the admissibility check requires more computation resources than the repair computation. The maximal memory used for the admissibility test was $1,251$ MB in contrast to 37.16 MB for the repair computation. This is in line with our expectation since the admissibility test searches the state space of the full NTA, while the repair analyses only considers a single TDT.

6 Conclusion

We have presented the TarTar tool, its architecture and implementation, and illustrated its application to a number of significant case studies. In the course of our work we have extended the repair analysis that is implemented in TarTar for bound modification to modifications of comparison operators, clock references, reset of clocks and missing urgencies. The evaluation of the repair analyses showed that an admissible repair is computed for at least 69% of the analyzed TDTs. The integration of various tools with heterogeneous interfaces posed a particular challenge to the architecture of TarTar which we addressed by the definition of intermediate artifacts.

In future work we plan to explore the interplay between different repairs that are computed for a repaired system that still violates a property, and develop refined strategies to select promising repairs from a repair set. A further generalization of the analysis is to not only compute clock constraint modifications for faulty models but also to compute possible relaxations of clock constraints for correct models in order to support design space exploration.

References

[BFT17] Barrett, C., Fontaine, P., Tinelli, C.: SMT-lib (2017). http://smtlib.cs.uiowa.edu/language.shtml

[BLL+95] Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL—a tool suite for automatic verification of real-time systems. In: Alur, R., Henzinger, T.A., Sontag, E.D. (eds.) HS 1995. LNCS, vol. 1066, pp. 232–243. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0020949

[BY03] Bengtsson, J., Yi, W.: Timed automata: semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27755-2_3

[DDG+11] Daniel, B.: Reassert: a tool for repairing broken unit tests. In: ICSE, pp. 1010–1012. ACM (2011)

[DHJ+11] Dalsgaard, A.E., et al.: opaal: a lattice model checker. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 487–493. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_37

[dMB08] 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

[HZWK18] Hua, J., Zhang, M., Wang, K., Khurshid, S.: SketchFix: a tool for automated program repair approach using lazy candidate generation. In: ESEC/SIGSOFT FSE, pp. 888–891. ACM (2018)

[IHS15] Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 487–495. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_32

[JH11] Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Software Eng. **37**(5), 649–678 (2011)

[JM11] Jose, M., Majumdar, R.: Bug-assist: assisting fault localization in ANSI-C programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 504–509. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_40

[KLW19] Kölbl, M., Leue, S., Wies, T.: Clock bound repair for timed systems. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 79–96. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_5

[KLW20] Kölbl, M., Leue, S., Wies, T.: Tartar: a timed automata repair tool. CoRR, abs/2002.02760 (2020). https://www.sen.uni-konstanz.de/publications

[LCL+17] Le, X.-B.D., Chu, D.-H., Lo, D., Goues, C.L., Visser, W.: S3: syntax- and semantic-guided repair synthesis via programming by examples. In: ESEC/SIGSOFT FSE, pp. 593–604. ACM (2017)

[LPR19] Le Goues, C., Pradel, M., Roychoudhury, A.: Automated program repair. Commun. ACM **62**(12), 56–65 (2019)

[Mav19] Apache Software Foundation. Maven (2019). https://maven.apache.org/

[MYR16] Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: scalable multiline program patch synthesis via symbolic analysis. In ICSE, pp. 691–701. ACM (2016)

[NQRC13] Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: Semfix: program repair via semantic analysis. In: ICSE, pp. 772–781. IEEE Computer Society (2013)

[tar20] Tartar 2019–2020. https://github.com/sen-uni-kn/tartar

Hybrid and Dynamic Systems

SAW: A Tool for Safety Analysis of Weakly-Hard Systems

Chao Huang[1(✉)], Kai-Chieh Chang[2], Chung-Wei Lin[2], and Qi Zhu[1]

[1] Northwestern University, Evanston, USA
{chao.huang,qzhu}@northwestern.edu
[2] National Taiwan University, Taipei, Taiwan
551100kk@gmail.com, cwlin@csie.ntu.edu.tw

Abstract. We introduce SAW, a tool for safety analysis of weakly-hard systems, in which traditional hard timing constraints are relaxed to allow bounded deadline misses for improving design flexibility and runtime resiliency. Safety verification is a key issue for weakly-hard systems, as it ensures system safety under allowed deadline misses. Previous works are either for linear systems only, or limited to a certain type of nonlinear systems (e.g., systems that satisfy exponential stability and Lipschitz continuity of the system dynamics). In this work, we propose a new technique for infinite-time safety verification of general nonlinear weakly-hard systems. Our approach first discretizes the safe state set into grids and constructs a directed graph, where nodes represent the grids and edges represent the reachability relation. Based on graph theory and dynamic programming, our approach can effectively find the safe initial set (consisting of a set of grids), from which the system can be proven safe under given weakly-hard constraints. Experimental results demonstrate the effectiveness of our approach, when compared with the state-of-the-art. An open source implementation of our tool is available at https://github.com/551100kk/SAW. The virtual machine where the tool is ready to run can be found at https://www.csie.ntu.edu.tw/~r08922054/SAW.ova.

Keywords: Weakly-hard systems · Safety verification · Graph theory

1 Introduction

Hard timing constraints, where deadlines should always been met, have been widely used in real-time systems to ensure system safety. However, with the

This work is supported by the National Science Foundation awards 1834701, 1834324, 1839511, 1724341, and the Office of Naval Research grant N00014-19-1-2496. It is also supported by the Asian Office of Aerospace Research and Development (AOARD), jointly with the Office of Naval Research Global (ONRG), award FA2386-19-1-4037, the Taiwan Ministry of Education (MOE) grants NTU-107V0901 and NTU-108V0901, the Taiwan Ministry of Science and Technology (MOST) grants MOST-108-2636-E-002-011 and MOST-109-2636-E-002-022.

C. Huang and K.-C. Chang—Contributed equally.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 543–555, 2020.
https://doi.org/10.1007/978-3-030-53288-8_26

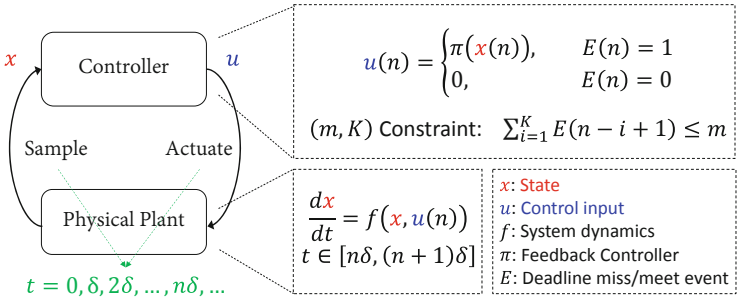


Fig. 1. A weakly-hard system with perfect sensors and actuators.

rapid increase of system functional and architectural complexity, hard deadlines have become increasingly pessimistic and often lead to infeasible designs or over provisioning of system resources [16,20,21,32]. The concept of weakly-hard systems are thus proposed to relax hard timing constraints by allowing occasional deadline misses [2,11]. This is motivated by the fact that many system functions, such as some control tasks, have certain degrees of robustness and can in fact tolerate some deadline misses, as long as those misses are bounded and dependably controlled. In recent years, considerable efforts have been made in the research of weakly-hard systems, including schedulability analysis [1,2,5,12–14,19,25,28,30], opportunistic control for energy saving [18], control stability analysis and optimization [8,10,22,23,26], and control-schedule co-design under possible deadline misses [3,6,27]. Compared with hard deadlines, weakly-hard constraints can more accurately capture the timing requirements of those system functions that tolerate deadline misses, and significantly improve system feasibility and flexibility [16,20]. Compared with soft deadlines, where any deadline miss is allowed, weakly-hard constraints could still provide deterministic guarantees on system safety, stability, performance, and other properties under formal analysis [17,29].

A common type of weakly-hard model is the (m, K) constraint, which specifies that among any K consecutive task executions, at most m instances could violate their deadlines [2]. Specifically, the high-level structure of a (m, K)-constrained weakly-hard system is presented in Fig. 1. Given a sampled-data system $\dot{x} = f(x, u)$ with a sampling period $\delta > 0$, the system samples the state x at the time $t = i\delta$ for $n = 0, 1, 2, \ldots$, and computes the control input u with function $\pi(x)$. If the computation completes within the given deadline, the system applies u to influence the plant's dynamics. Otherwise, the system stops the computation and applies zero control input. As aforementioned, the system should ensure the control input can be successfully computed and applied within the deadline for at least $K-m$ times over any K consecutive sampling periods.

For such weakly-hard systems, a natural and critical question is whether the system is safe by allowing deadline misses defined in a given (m, K) constraint.

There is only limited prior work in this area, while nominal systems have been adequately studied [4,9,15,31]. In [8], a weakly-hard system with linear dynamic is modeled as a hybrid automaton and then the reachability of the generated hybrid automaton is verified by the tool SpaceEx [9]. In [7], the behavior of a linear weakly-hard system is transformed into a program, and program verification techniques such as abstract interpretation and SMT solvers can be applied.

In our previous work [17], the safety of nonlinear weakly-hard systems are considered for the first time. Our approach tries to derive a safe initial set for any given (m, K) constraint, that is, starting from any initial state within such set, the system will always stay within the same safe state set under the given weakly-hard constraint. Specifically, we first convert the infinite-time safety problem into a finite one by finding a set satisfying both *local safety* and *inductiveness*. The computation of such valid set heavily lies on the estimation of the system state evolution, where two key assumptions are made: 1) The system is exponentially stable under nominal cases without any deadline misses, which makes the system state contract with a constant decay rate; 2) The system dynamics are Lipschitz continuous, which helps bound the expansion under a deadline miss. Based on these two assumptions, we can abstract the safety verification problem as a one-dimensional problem and use linear programming (LP) to solve it, which we call *one-dimension abstraction* in the rest of the paper.

In practice, however, the assumptions in [17] are often hard to satisfy and the parameters of exponential stability are difficult to obtain. In addition, while the scalar abstraction provides high efficiency, the experiments demonstrate that the estimation is always over conservative. In this paper, we go one step further and present a new tool SAW for infinite-time safety verification of nonlinear weakly-hard systems **without any particular assumption on exponential stability and Lipschitz bound**, and try to be less conservative than the scalar abstraction. Formally, the problem solved by this tool is described as follows:

Problem 1. Given an (m, K) weakly-hard system with nonlinear dynamics $\dot{x} = f(x, u)$, sampling period δ, and safe set X, find a safe initial set X_0, such that from any state $x(0) \in X_0$, the system will always be inside X.

To solve this problem, we first discretize the safe state set X into grids. We then try to find the grid set that satisfies both local safety and inductiveness. For each property, we build a directed graph, where each node corresponds to a grid and each directed edge represents the mapping between grids with respect to reachability. We will then be able to leverage graph theory to construct the initial safe set. Experimental results demonstrate that our tool is effective for general nonlinear systems.

2 Algorithms and Tool Design

The schematic diagram of our tool SAW is shown in Fig. 2. The input is a model file that specifies the system dynamics, sampling period, safe region and other parameters, and a configuration file of Flow* [4] (which is set by default but can

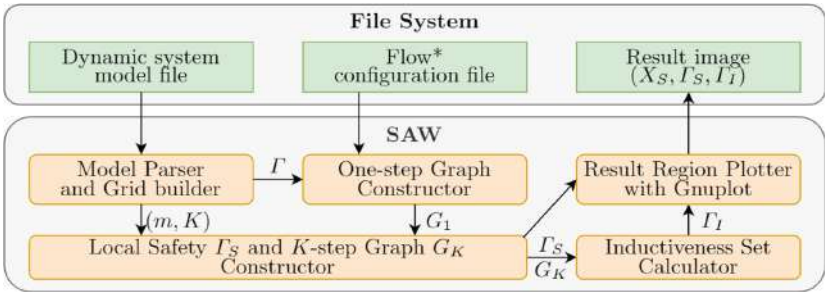


Fig. 2. The schematic diagram of SAW.

Algorithm 1: Overall algorithm of SAW

Data: Dynamic system f with safe state region X, the control law π, weakly-hard constraint (m, K), sampling period δ
Result: Safe initial state set X_0

1 $\Gamma = \textbf{partition}(X, p)$;
/* Search the grid set that satisfies local safety. */
2 $G_1 = \textbf{constructOneStepGraph}()$;
3 $\Gamma_S, G_K = \textbf{calculateLocalSafety}()$;
/* Search the grid set that satisfies inductiveness. */
4 $\Gamma_I = \textbf{calculateInductivenessSet}()$;
5 **return** Γ_I;

also be customized). After fed with the input, the tool works as follows (shown in Algorithm 1). The safe state set X is first uniformly partitioned into small grids $\Gamma = \{v_1, v_2, \ldots, v_{p^d}\}$, where $X = v_1 \cup v_2 \cup \cdots \cup v_{d^p}$, $v_i \cap v_j = \phi$ $(\forall i \neq j)$, d is the dimension of the state space, and p is the number of partitions in each dimension (Line 1 in Algorithm 1). The tool then tries to find the grids that satisfy the local safety. It first invokes a reachability graph constructor to build a one-step reachability graph G_1 to describe how the system evolves in one sampling step (Line 2). Then, a dynamic programming (DP) based approach finds the largest set $\Gamma_S = \{v_{s_1}, v_{s_2}, \ldots, v_{s_n}\}$ from which the system will not go out of the safe region. The K-step reachability graph G_K is also built in the DP process based on G_1 (Line 3). After that, the tool searches the largest subset Γ_I of Γ_S that satisfies the inductiveness by using a reverse search algorithm (Line 4). The algorithm outputs Γ_I as the target set X_0 (Line 5).

The key functions of the tool are the reachability graph constructor, DP-based local safety set search, and reverse inductiveness set search. In the following sections, we introduce these three functions in detail.

2.1 Reachability Graph Construction

Integration in dynamic system equations is often the most time-consuming part to trace the variation of the states. In this function, we use Flow* to get a valid

Algorithm 2: Construct one-step graph: **constructOneStepGraph**()

```
   Data: Dynamic system f, grid set Γ, the control law π, sampling period δ
   Result: Directed graph G_1(Γ, E_1)
   /* Initialize the edge set E_1 of G_1.                                  */
1  E_1 ⟵ ∅;
2  for v ∈ Γ do
       /* Consider deadline miss (e = 1)/meet (e = 0) respectively.       */
3      for e ∈ {0, 1} do
           /* Compute one step reachable set R_1(v) from v.               */
4          R_1(v) = Flow*(v, δ, e);
           /* v is unsafe and no edge is added if X^c ∩ R_1(v) ≠ ∅.        */
5          if X^c ∩ R_1(v) ≠ ∅ then Conitnue;
           /* Add an edge pointing v' from v if v' ∩ R_1(v) ≠ ∅.           */
6          for v' ∈ Γ do
7              if v' ∩ R_1(v) ≠ ∅ then E_1 ⟵ E_1 ∪ {(v, e, v')};
8  return G_1(Γ, E_1);
```

overapproximation of reachable set (represented as flowpipes) starting from every grid after a sampling period δ. Given a positive integer n, the graph constructed by the reachability set after n sampling period, $n \cdot \delta$, is called a n-step graph G_n. Since the reachability for all the grids in any sampling step is independent under our grid assumption, we first build G_1 and then reuse G_1 to construct G_K later without redundant computation of reachable set.

One-step graph is built with Algorithm 2. We consider deadline miss and deadline meet separately, corresponding to two categories of edges (Line 3). For a grid v, if the one-step reachable set $R_1(v)$ intersects with unsafe state X^c, then it is considered as an unsafe grid and we let its reachable grid be $\emptyset$. Otherwise, if $R_1(v)$ intersects with another grid v' under the deadline miss/meet event e, then we add a directed edge (v, e, v') from v' to v with label e. The number of outgoing edges for each grid node v is bounded by p^d. Assuming that the complexity of Flow* to compute flowpipes for its internal clock ϵ is $O(1)$, we can get the overall time complexity as $O(|\Gamma| \cdot p^d \cdot \delta/\epsilon)$.

K-step graph G_K is built for finding the grid set that satisfies local safety and inductiveness. To avoid redundant computation on reachable set, we construct G_K based on G_1 by traversing K-length paths, as the bi-product of local safety set searching procedure.

2.2 DP-Based Local Safety Set Search

We propose a bottom-up dynamic programming for considering all the possible paths, utilizing the overlapping subproblems property (Algorithm 3). The reachable grid set at step K that is derived from a grid v at step $k \leq K$ with respect to the number of deadline misses $n \leq m$ can be defined as $\mathrm{DP}(v, n, k)$. To be consistent with Algorithm 2, this set is empty if and only if it does not satisfy the local safety. We need to derive $\mathrm{DP}(v, 0, 0)$. Initially, the zero-step reachability is

Algorithm 3: Search grid set for local safety: **calculateLocalSafety**()

Data: Directed graph $G_1(\Gamma, E_1)$, weakly-hard constraint (m, K)
Result: Grid set Γ_S, directed graph $G_K(\Gamma, E_K)$

```
1  for v ∈ Γ do
2      for n ← 0 to m do
3          DP(v, n, K) ⟵ {v};
4  for k ← K − 1 to 0 do
5      for v ∈ Γ do
6          for n ← 0 to m do
7              isSafe ← True;
8              for e ∈ {0, 1} do
9                  if n + e ≤ m then
10                     nextGrids(v) ⟵ {v′ | (v, e, v′) ∈ E1};
11                     if nextGrids(v) = ∅ then isSafe ← False; break;
12                     for v′ ∈ nextGrids(v) do
13                         R(v′) ⟵ DP(v, n + e, k + 1);
14                         if R(v) = ∅ then isSafe ← False; break;
15                         DP(v′, n, k) ⟵ DP(v′, n, k) ∪ R(v);
16             if isSafe = false then
17                 DP(v, n, k) ⟵ ∅;
18 Γ_S ⟵ {v | DP(v, 0, 0) ≠ ∅};
19 E_K ⟵ {(v, v′) | v ∈ DP(v, 0, 0)};
20 return Γ_S, G_K(Γ, E_K);
```

straight forward, i.e., $\forall u \in \Gamma, n \in [0, m]$, $\mathrm{DP}(v, n, K) = \{v\}$. The transition is defined as:

$$\forall k \in [0, K-1]: \ \mathrm{DP}(v, n, k) = \bigcup_{\forall v', e:(v,e,v') \in E_1, n+e \le m} \mathrm{DP}(v', n+e, k+1).$$

If there exists an empty set on the right hand side or there is no outgoing edge from v for any e such that $n + e \le m$, we let $\mathrm{DP}(v, n, k) = \emptyset$. Finally, we have $\Gamma_S = \{v \mid \mathrm{DP}(v, 0, 0) \neq \emptyset\}$, $E_K = \{(v, v') \mid v' \in \mathrm{DP}(v, 0, 0)\}$.

We used *bitset* to implement the set union which can accelerate 64 times under the 64-bit architecture. The time complexity is $O(|\Gamma|^2/bits \cdot p^d \cdot K^2 + |\Gamma|^2)$, where *bits* depends on the running environment. $|\Gamma|^2$ is contributed by G_K.

2.3 Reverse Inductiveness Set Search

To find the grid set $\Gamma_I \subseteq \Gamma_S$ that satisfies inductiveness, we propose a reverse search algorithm Algorithm 4. Basically, instead of directly searching Γ_I, we try to obtain Γ_I by removing any grid v within Γ_S, from which there exists a path reaching $\Gamma_U = \Gamma - \Gamma_S$. Specifically, Algorithm 4 starts with initializing $\Gamma_U = \Gamma - \Gamma_S$ (line 1). The Γ_U iteratively absorbs the grid v that can reach Γ_U in K sampling periods, until a fixed point is reached (line 2–3). Finally $\Gamma_I = \Gamma - \Gamma_U$ is the largest set that satisfies inductiveness. It is implemented as a breadth first search (BFS) on the reversed graph of G_K, and the time complexity is $O(|\Gamma|^2)$.

Algorithm 4: Search grid set for inductiveness: **calculateInductiveness-Set**()

Data: Directed graph $G_K(\Gamma, E_K)$, Grid set Γ_S
Result: Grid set Γ_I
$\Gamma_U \longleftarrow \Gamma - \Gamma_S$;
while $\exists (v, v') \in E_K$ *such that* $v \notin \Gamma_U, v' \in \Gamma_U$ **do**
  $\Gamma_U \longleftarrow \Gamma_U \cup \{v\}$;
$\Gamma_I = \Gamma - \Gamma_U$;
return Γ_I;

3 Example Usage

Example 1. Consider the following linear control system from [17]:

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -0.1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + u, \quad \text{where} \quad u = \begin{bmatrix} 0 & 0 \\ -0.375 & -1.15 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

$\delta = 0.2$ and $step\_size = 0.01$. The initial state set is $x_1 \in [-1, 1]$ and $x_2 \in [-1, 1]$. The safe state set is $x_1 \in [-3, 3]$ and $x_2 \in [-3, 3]$. Following the input format shown in Listing 1.1. Thus, we prepare the model file as Listing 1.2.

```
<state_dim> <input_dim> <grid_count>
<state_var_names> <input_var_names>
<state_ode.1>
...
<state_ode.state_dim>
<input_equa.1>
...
<input_equa.input_dim>
<period> <step_size>
<m> <k>
<safe_state.1>
...
<safe_state.state_dim>
<initial_state.1>
...
<initial_state.state_dim>
```

Listing 1.1. Input format

```
2 1 50
x1 x2 u
x2
-0.1 * x2 + u
-0.375 * x1 - 1.15 * x2
0.2 0.01
2 5
-3 3
-3 3
-1 1
-1 1
```

Listing 1.2. example/model1.txt

Then, we run our program with the model file.

```
./saw example/model1.txt
```

To further ease the use of our tool, we also pre-complied our tool for x86_64 linux environment. In such environment, users do not need to compile our tool and can directly invoke **saw_linux_x86_64** instead of **saw** (which is only available after manually compiling the tool).

```
./saw_linux_x86_64 example/model1.txt
```

The program output is shown in Listing 1.3. Line 6 shows the number of edges of G_1. Lines 8–10 provide the information of G_K, including the number of edges and nodes. Line 12 prints the safe initial set X_0. Our tool then determines whether the given initial set is safe by checking if it is the subset of X_0.

```
[Info] Parsing model.
[Info] Building FLOW* configuration.
[Info] Building grids.
[Info] Building one-step graph.
       Process: 100.00%
[Success] Number of edges: 19354
[Info] Building K-step graph.
[Success] Start Region Size: 1908
          End Region: 1208
          Number of Edges: 102436
[Info] Finding the largest closed subgraph.
[Success] Safe Initial Region Size: 1622
[Info] Calculating area.
       Initial state region: 4.000000
       Grids Intersection:   4.000000
       Result: safe
```

Listing 1.3. Verification result

Table 1. Benchmark setting. ODE denotes the ordinary differential equation of the example, π denotes the control law, and δ is the discrete control stepsize.

#	ODE	π	δ	Safe state set	(m, K)
1	$\dot{x}_1 = x_2$ $\dot{x}_2 = -0.1x_2 + u$	$u = -0.375x_1 - 1.15x_2$	0.2	$x_1 \in [-3.0, 3.0]$ $x_2 \in [-3.0, 3.0]$	(2, 5)
2	$\dot{x}_1 = -2x_1 + u_1$ $\dot{x}_2 = -0.9x_2 + u_2$	$u_1 = -x_1$ $u_2 = -x_1 - x_2$	0.3	$x_1 \in [-6.0, 6.0]$ $x_2 \in [-6.0, 6.0]$	(1, 10)
3	$\dot{x}_1 = x_2 + u$ $\dot{x}_2 = -2x_1 - 0.1x_2 + u$	$u = x_1$	1.6	$x_1 \in [-3.0, 3.0]$ $x_2 \in [-3.0, 3.0]$	(2, 10)
4	$\dot{x} = x^2 - x^3 + u$	$u = -2x$	0.6	$x \in [-4.0, 4.0]$	(2, 100)
5	$\dot{x} = 0.2x + 0.03x^2 + u$	$u = -0.3x^3$	1.6	$x \in [-2.0, 2.0]$	(1, 5)
6	$\dot{x}_1 = x_2 - x_1^3 + x_1^2$ $\dot{x}_2 = u$	$u = -1.22x_1 - 0.57x_2$ $-0.129x_2^3$	0.1	$x_1 \in [-5.0, 5.0]$ $x_2 \in [-5.0, 5.0]$	(2, 15)

4 Experiments

We implemented a prototype of SAW that is integrated with Flow*. In this section, we first compare our tool with the one-dimension abstraction [17], on the full benchmarks from [17] (#1–#4) and also additional examples with no guarantee on exponential stability from related works (#5 and #6) [24]. Table 1 shows the benchmark settings, including the (m, K) constraint set for each benchmark. Then, we show how different parameter settings affect the verification results of our tool. All our experiments were run on a desktop, with 6-core 3.60 GHz Intel Core i7.

4.1 Comparison with One-Dimension Abstraction

Table 2 shows the experimental results. It is worth noting that the one-dimension abstraction cannot find the safe initial set in most cases from [17]. In fact, it only

Table 2. Experimental results. ExpParam denotes the parameters of the exponential stability, where "N/A" means that either the system is not exponentially stable or the parameters are not available. Initial state set denotes the set that needs to be verified. The last two columns denote the verification results of the one-dimension abstraction [17] and SAW, respectively. "—" means that no safe initial set X_0 is found by the tool. p represents the partition number for each dimension in SAW. Time (in seconds) represents the execution time of SAW.

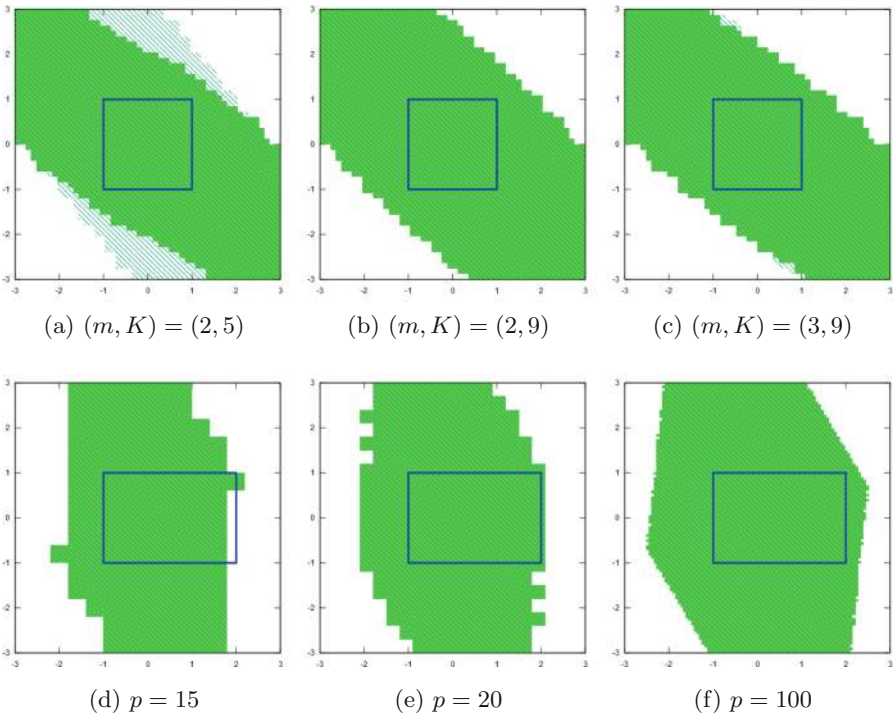
#	ExpParam	Initial state set	One-dimension abstraction	SAW		
			Result	p	Result	Time
1	$\alpha = 1.8$, $\lambda = 0.4$	$x_1 \in [-1.0, 1.0]$ $x_2 \in [-1.0, 1.0]$	—	50	Yes	72.913
2	$\alpha = 1.1$, $\lambda = 1.8$	$x_1 \in [-6.0, 6.0]$ $x_2 \in [-6.0, 6.0]$	No $(X_0 : x_1^2 + x_2^2 \leq 1.947^2)$	30	Yes	10.360
3	$\alpha = 2$, $\lambda = 0.37$	$x_1 \in [-1.0, 2.0]$ $x_2 \in [-1.0, 1.0]$	—	100	Yes	183.30
4	$\alpha = 1.4$, $\lambda = 1$	$x \in [-4.0, 4.0]$	—	30	Yes	80.613
5	N/A	$x \in [-1.56, 1.32]$	—	100	Yes	4.713
6	N/A	$x_1 \in [-5.0, 5.0]$ $x_2 \in [-5.0, 5.0]$	—	50	Yes	750.77

works effectively for a limited set of (m, K), e.g., when no consecutive deadline misses is allowed. For general (m, K) constraints, one-dimension abstraction performs much worse due to the over-conservation. Furthermore, we can see that, without exponential stability, one-dimension abstraction based approach is not applicable for the benchmarks #5 and #6. Note that for benchmark #2, one-dimension abstraction obtains a non-empty safe initial set X_0, which however, does not contain the given initial state set. Thus we use "No" instead of "—" to represent this result. Conversely, for every example, our tool computes a feasible X_0 that contains the initial state set (showing the initial state set is safe), which we denote as "Yes".

4.2 Impact of (m, K), Granularity, and Stepsize

(m, K). We take benchmark #1 (Example 1 in Sect. 3) as an example and run our tool under different (m, K) values. Figures 3a, 3b, 3c demonstrate that, for this example, the size of local safety region Γ_S shrinks when K gets larger. The size of inductiveness region Γ_I grows in contrast. Γ_S becomes the same as Γ_I when K gets larger, in which case m is the primary parameter that influences the size of Γ_I.

Granularity. We take benchmark #3 as an example, and run our tool with different partition granularities. The results (Figs. 3d, 3e, 3f) show that Γ_I grows



(a) $(m, K) = (2, 5)$ (b) $(m, K) = (2, 9)$ (c) $(m, K) = (3, 9)$

(d) $p = 15$ (e) $p = 20$ (f) $p = 100$

Fig. 3. Results under different (m, K) values (3a, 3b, 3c) and different granularities (3d, 3e, 3f). The green solid region is Γ_I. The slashed region is Γ_S. The blue rectangle is the initial state set that needs to be verified. (Color figure online)

when p gets larger. The choice of p has significant impact on the result (e.g., the user-defined initial state set cannot be verified when $p = 15$).

Stepsize. We take benchmark #5 as an example, and run our tool with different stepsizes of Flow*. With the same granularity $p = 100$, we get the safe initial state set $\Gamma_I = [-1.56, 1.32]$ when $step\_size = 0.1$, but Γ_I is empty when $step\_size = 0.3$. The computation times are 4.713 s and 1.835 s, respectively. Thus, we can see that there is a trade-off between the computational efficiency and the accuracy.

5 Conclusion

In this paper, we present a new tool SAW to compute a tight estimation of safe initial set for infinite-time safety verification of general nonlinear weakly-hard systems. The tool first discretizes the safe state set into grids. By constructing a reachability graph for the grids based on existing tools, the tool leverages graph theory and dynamic programming technique to compute the safe initial

set. We demonstrate that our tool can significantly outperform the state-of-the-art one-dimension abstraction approach, and analyze how different constraints and parameters may affect the results of our tool. Future work includes further speedup of the reachability graph construction via parallel computing.

References

1. Ahrendts, L., Quinton, S., Boroske, T., Ernst, R.: Verifying weakly-hard real-time properties of traffic streams in switched networks. In: Altmeyer, S. (ed.) 30th Euromicro Conference on Real-Time Systems (ECRTS 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 106, pp. 15:1–15:22. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2018). https://doi.org/10.4230/LIPIcs.ECRTS.2018.15. http://drops.dagstuhl.de/opus/volltexte/2018/8987
2. Bernat, G., Burns, A., Liamosi, A.: Weakly hard real-time systems. IEEE Trans. Comput. **50**(4), 308–321 (2001). https://doi.org/10.1109/12.919277
3. Bund, T., Slomka, F.: Controller/platform co-design of networked control systems based on density functions. In: ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems, pp. 11–14. ACM (2014)
4. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18
5. Choi, H., Kim, H., Zhu, Q.: Job-class-level fixed priority scheduling of weakly-hard real-time systems. In: IEEE Real-Time Technology and Applications Symposium (RTAS) (2019)
6. Chwa, H.S., Shin, K.G., Lee, J.: Closing the gap between stability and schedulability: a new task model for cyber-physical systems. In: IEEE Real-Time Technology and Applications Symposium (RTAS) (2018)
7. Duggirala, P.S., Viswanathan, M.: Analyzing real time linear control systems using software verification. In: RTSS, pp. 216–226. IEEE (2015)
8. Frehse, G., Hamann, A., Quinton, S., Woehrle, M.: Formal analysis of timing effects on closed-loop properties of control software. In: 2014 IEEE Real-Time Systems Symposium, pp. 53–62, December 2014. https://doi.org/10.1109/RTSS.2014.28
9. Frehse, G., et al.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_30
10. Gaid, M.B., Simon, D., Sename, O.: A design methodology for weakly-hard real-time control. IFAC Proc. Vol. **41**(2), 10258–10264 (2008). https://doi.org/10.3182/20080706-5-KR-1001.01736. http://www.sciencedirect.com/science/article/pii/S1474667016406129, 17th IFAC World Congress
11. Hamdaoui, M., Ramanathan, P.: A dynamic priority assignment technique for streams with (m, k)-firm deadlines. IEEE Trans. Comput. **44**(12), 1443–1451 (1995)
12. Hammadeh, Z.A.H., Ernst, R., Quinton, S., Henia, R., Rioux, L.: Bounding deadline misses in weakly-hard real-time systems with task dependencies. In: Design, Automation Test in Europe Conference Exhibition (DATE), pp. 584–589, March 2017. https://doi.org/10.23919/DATE.2017.7927054

13. Hammadeh, Z.A.H., Quinton, S., Ernst, R.: Extending typical worst-case analysis using response-time dependencies to bound deadline misses. In: Proceedings of the 14th International Conference on Embedded Software, EMSOFT 2014, pp. 10:1–10:10. ACM, New York (2014). https://doi.org/10.1145/2656045.2656059. http://doi.acm.org/10.1145/2656045.2656059
14. Hammadeh, Z.A.H., Quinton, S., Panunzio, M., Henia, R., Rioux, L., Ernst, R.: Budgeting under-specified tasks for weakly-hard real-time systems. In: Bertogna, M. (ed.) 29th Euromicro Conference on Real-Time Systems (ECRTS 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 76, pp. 17:1–17:22. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2017). https://doi.org/10.4230/LIPIcs.ECRTS.2017.17. http://drops.dagstuhl.de/opus/volltexte/2017/7163
15. Huang, C., Chen, X., Lin, W., Yang, Z., Li, X.: Probabilistic safety verification of stochastic hybrid systems using barrier certificates. TECS **16**(5s), 186 (2017)
16. Huang, C., Wardega, K., Li, W., Zhu, Q.: Exploring weakly-hard paradigm for networked systems. In: Workshop on Design Automation for CPS and IoT (DESTION 2019) (2019)
17. Huang, C., Li, W., Zhu, Q.: Formal verification of weakly-hard systems. In: The 22nd ACM International Conference on Hybrid Systems: Computation and Control (HSCC) (2019)
18. Huang, C., Xu, S., Wang, Z., Lan, S., Li, W., Zhu, Q.: Opportunistic intermittent control with safety guarantees for autonomous systems. In: Design Automation Conference (DAC) (2020)
19. Li, J., Song, Y., Simonot-Lion, F.: Providing real-time applications with graceful degradation of QoS and fault tolerance according to (m, k)-firm model. IEEE Trans. Industr. Inf. **2**(2), 112–119 (2006)
20. Liang, H., Wang, Z., Roy, D., Dey, S., Chakraborty, S., Zhu, Q.: Security-driven codesign with weakly-hard constraints for real-time embedded systems. In: 37th IEEE International Conference on Computer Design (ICCD 2019) (2019)
21. Lin, C., Zheng, B., Zhu, Q., Sangiovanni-Vincentelli, A.: Security-aware design methodology and optimization for automotive systems. ACM Trans. Des. Autom. Electron. Syst. (TODAES) **21**(1), 18:1–18:26 (2015). https://doi.org/10.1145/2803174. http://doi.acm.org/10.1145/2803174
22. Marti, P., Camacho, A., Velasco, M., Gaid, M.E.M.B.: Runtime allocation of optional control jobs to a set of CAN-based networked control systems. IEEE Trans. Industr. Inf. **6**(4), 503–520 (2010). https://doi.org/10.1109/TII.2010.2072961
23. Pazzaglia, P., Pannocchi, L., Biondi, A., Natale, M.D.: Beyond the weakly hard model: measuring the performance cost of deadline misses. In: Altmeyer, S. (ed.) 30th Euromicro Conference on Real-Time Systems (ECRTS 2018). Leibniz International Proceedings in Informatics (LIPIcs), vol. 106, pp. 10:1–10:22. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2018). https://doi.org/10.4230/LIPIcs.ECRTS.2018.10. http://drops.dagstuhl.de/opus/volltexte/2018/8993
24. Prajna, S., Parrilo, P.A., Rantzer, A.: Nonlinear control synthesis by convex optimization. IEEE Trans. Autom. Control **49**(2), 310–314 (2004)
25. Quinton, S., Hanke, M., Ernst, R.: Formal analysis of sporadic overload in real-time systems. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2012, EDA Consortium, San Jose, CA, USA, pp. 515–520 (2012). http://dl.acm.org/citation.cfm?id=2492708.2492836
26. Ramanathan, P.: Overload management in real-time control applications using (m, k)-firm guarantee. IEEE Trans. Parallel Distrib. Syst. **10**(6), 549–559 (1999). https://doi.org/10.1109/71.774906

27. Soudbakhsh, D., Phan, L.T., Annaswamy, A.M., Sokolsky, O.: Co-design of arbitrated network control systems with overrun strategies. IEEE Trans. Control Netw. Syst. **5**(1), 128–141 (2016)
28. Sun, Y., Natale, M.D.: Weakly hard schedulability analysis for fixed priority scheduling of periodic real-time tasks. ACM Trans. Embed. Comput. Syst. (TECS) **16**(5s), 171 (2017)
29. Wardega, K., Li, W.: Application-aware scheduling of networked applications over the low-power wireless bus. In: Design, Automation and Test in Europe Conference (DATE), March 2020
30. Xu, W., Hammadeh, Z.A.H., Kröller, A., Ernst, R., Quinton, S.: Improved deadline miss models for real-time systems using typical worst-case analysis. In: 2015 27th Euromicro Conference on Real-Time Systems, pp. 247–256, July 2015. https://doi.org/10.1109/ECRTS.2015.29
31. Yang, Z., Huang, C., Chen, X., Lin, W., Liu, Z.: A Linear programming relaxation based approach for generating barrier certificates of hybrid systems. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 721–738. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_44
32. Zhu, Q., Sangiovanni-Vincentelli, A.: Codesign methodologies and tools for cyber-physical systems. Proc. IEEE **106**(9), 1484–1500 (2018). https://doi.org/10.1109/JPROC.2018.2864271

PIRK: Scalable Interval Reachability Analysis for High-Dimensional Nonlinear Systems

Alex Devonport[1(✉)], Mahmoud Khaled[2], Murat Arcak[1], and Majid Zamani[3,4]

[1] University of California, Berkeley, Berkeley, CA, USA
{alex_devonport,arcak}@berkeley.edu
[2] Technical University of Munich, Munich, Germany
khaled.mahmoud@tum.de
[3] University of Colorado, Boulder, Boulder, CO, USA
majid.zamani@colorado.edu
[4] Ludwig Maximilian University, Munich, Germany

Abstract. Reachability analysis is a critical tool for the formal verification of dynamical systems and the synthesis of controllers for them. Due to their computational complexity, many reachability analysis methods are restricted to systems with relatively small dimensions. One significant reason for such limitation is that those approaches, and their implementations, are not designed to leverage parallelism. They use algorithms that are designed to run serially within one compute unit and they can not utilize widely-available high-performance computing (HPC) platforms such as many-core CPUs, GPUs and Cloud-computing services.

This paper presents PIRK, a tool to efficiently compute reachable sets for general nonlinear systems of extremely high dimensions. PIRK can utilize HPC platforms for computing reachable sets for general high-dimensional non-linear systems. PIRK has been tested on several systems, with state dimensions up to 4 billion. The scalability of PIRK's parallel implementations is found to be highly favorable.

Keywords: Reachability analysis · ODE integration · Runge-Kutta method · Mixed monotonicity · Monte Carlo simulation · Parallel algorithms

1 Introduction

Applications of safety-critical cyber-physical systems (CPS) are growing due to emerging IoT technologies and the increasing availability of efficient computing devices. These include smart buildings, traffic networks, autonomous vehicles, truck platooning, and drone swarms, which require reliable bug-free

A. Devonport and M. Khaled—Contributed equally.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 556–568, 2020.
https://doi.org/10.1007/978-3-030-53288-8_27

software that perform in real-time and fulfill design requirements. Traditional simulation/testing-based strategies may only find a small percentage of the software defects and the repairs become much costly as the system complexity grows. Hence, in-development verification strategies are favorable since they reveal the faults in earlier stages, and guarantee that the designs satisfy the specifications as they evolve through the development cycle. Formal methods offer an attractive alternative to testing- and simulation-based approaches, as they can verify whether the specifications for a CPS are satisfied for all possible behaviors from a set of the initial states of the system. *Reachable sets* characterize the states a system can reach in a given time range, starting from a certain initial set and subjected to certain inputs. They play an important role in several formal methods-based approaches to the verification and controller synthesis. An example of this is *abstraction-based* synthesis [1–4], in which reachable sets are used to construct a finite-state "abstraction" which is then used for formal synthesis.

Computing an exact reachable set is generally not possible. Most practical methods resort to computing over-approximations or under-approximations of the reachable set, depending on the desired guarantee. Computing these approximations to a high degree of accuracy is still a computationally intensive task, particularly for high-dimensional systems. Many software tools have been created to address the various challenges of approximating reachable sets. Each of these tools uses different methods and leverages different system assumptions to achieve different goals related to computing reachable sets. For example, `CORA` [5] and `SpaceEx` [6] tools are designed to compute reachable sets of high accuracy for very general classes of nonlinear systems, including hybrid ones. Some reachability analysis methods rely on specific features of dynamical systems, such as linearity of the dynamics or sparsity in the interconnection structure [7–9]. This allows computing the reachable sets in shorter time or for relatively high-dimensional systems. However, it limits the approach to smaller classes of applications, less practical specifications, or requires the use of less accurate (e.g., linearized) models.

Other methods attack the computational complexity problem by computing reachable set approximations from a limited class of set representations. An example of limiting the set of allowed overapproximations are *interval reachability* methods, in which reachable sets are approximated by Cartesian products of intervals. Interval reachability methods allow for computing the reachable sets of very general non-linear and high-dimensional systems in a short amount of time. They also pose mild constraints on the systems under consideration, usually only requiring some kind of boundedness constraint instead of a specific form for the system dynamics. Many reachability tools that are designed to scale well with state dimension focus on interval reachability methods: these include `Flow*` [10], `CAPD` [11], `C2E2` [12], `VNODE-LP` [13], `DynIbex` [14], and `TIRA` [15].

Another avenue by which reachable set computation time can be reduced, which we believe has not been sufficiently explored, is the use of parallel computing. Although most reachability methods are presented as serial algorithms, many of them have some inherent parallelism that can be exploited. One example

of a tool that exploits parallelism is `XSpeed` [16], which implements a parallelized version of a support function-based reachability method. However, this parallel method is limited to linear systems, and in some cases only linear systems with invertible dynamics. Further, the parallelization is not suitable for massively parallel hardware: only some of the work (sampling of the support functions) is offloaded to the parallel device, so only a relatively small number of parallel processing elements may be employed.

In this paper, we investigate the parallelism for three interval reachability analysis methods and introduce `PIRK`, the Parallel Interval Reachability Kernel. `PIRK` uses *simulation-based* reachability methods [17–19], which compute rigorous approximations to reachable sets by integrating one or more systems of ODEs. `PIRK` is developed in `C++` and `OpenCL` as an open-source[1] *kernel* for `pFaces` [20], a recently introduced acceleration ecosystem. This allows `PIRK` to be run on a wide range of computing platforms, including CPUs clusters, GPUs, and hardware accelerators from any vendor, as well as cloud-based services like AWS.

The user looking to use a reachability analysis tool for formal verification may choose from an abundance of options, as our brief review has shown. What `PIRK` offers in this choice is a tool that allows for massively parallel reachability analysis of high-dimensional systems with an application programming interface (API) to easily interface with other tools. To the best of our knowledge, `PIRK` is the first and the only tool that can compute reachable sets of general non-linear systems with dimensions beyond the billion. As we show later in Sect. 5, `PIRK` computes the reachable set for a traffic network example with 4 billion dimension in only 44.7 min using a 96-core CPU in Amazon AWS Cloud.

2 Interval Reachability Analysis

Consider a nonlinear system with dynamics $\dot{x} = f(t, x, p)$ with state $x \in \mathbb{R}^n$, a set of initial states $\mathcal{X}_0$, a time interval $[t_0, t_1]$, and a set of time-varying inputs $\mathcal{P}$ defined over $[t_0, t_1]$. Let $\Phi(t; t_0, x_0, p)$ denote the state of the system, at time t, of the trajectory beginning at time t_0 at initial state x_0 under input p. We assume the systems are continuous-time.

The finite-time forward reachable set is defined as

$$R_{t_0,t_1} = \{\Phi(t_1; t_0, x, p) | x \in \mathcal{X}_0, p \in \mathcal{P}\}.$$

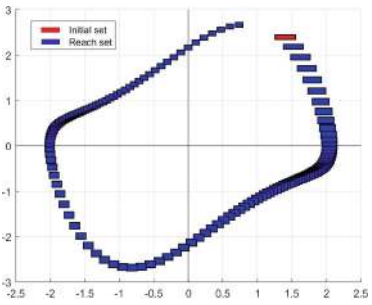


Fig. 1. An example of an Interval Reachability problem for a nonlinear system. Red rectangle: initial set. Blue rectangles: reachable sets for several final times t_1. (Color figure online)

For the problem of *interval* reachability analysis, there are a few more constraints on the problem structure. An *interval set* is a set of the form $[\underline{a}, \overline{a}] =$

[1] `PIRK` is publicly available at https://github.com/mkhaled87/pFaces-PIRK.

$\{a : \underline{a} \leq a \leq \overline{a}\}$, where $\leq$ denotes the usual partial order on real vectors, that is the partial order with respect to the positive orthant cone. The vectors $\underline{a}$ and $\overline{a}$ are the lower and upper bounds respectively of the interval set. An interval set can alternatively be described by its center $a^* = \frac{1}{2}(\overline{a} + \underline{a})$ and half-width $[a] = \frac{1}{2}(\overline{a} - \underline{a})$. In interval reachability analysis, the initial set must be an interval, and inputs values restricted to an interval set, i.e. $p(t) \in [\underline{p}, \overline{p}]$, and the reachable set approximation must also be an interval (Fig. 1). Furthermore, certain methods for computing interval reachable sets require further restrictions on the system dynamics, such as the state and input Jacobian matrices being bounded or sign-stable.

2.1 Methods to Compute Interval Reachable Sets

PIRK computes interval reachable sets using three different methods, allowing for different levels of tightness and speed, and which allow for different amounts of additional problem data to be used.

The *Contraction/Growth Bound* method [4,21,22] computes the reachable set using component-wise contraction properties of the system. This method may be applied to input-affine systems of the form $\dot{x} = f(t, x) + p$. The growth and contraction properties of each component of the system are first characterized by a *contraction matrix* C. The contraction matrix is a component-wise generalization of the matrix measure of the Jacobian $J_x = \partial f / \partial x$ [19,23], satisfying $C_{ii} \geq J_{x,ii}(t, x)$ for diagonal Jacobian elements $J_{x,ii}(t, x)$, and $C_{ij} \geq |J_{x,ij}(t, x)|$ for off-diagonal Jacobian elements $J_{x,ij}(t, x)$. The method constructs a reachable set over-approximation by separately establishing its *center* and *half-width*. The center is found by simulating the trajectory of the center of the initial set, that is as $\Phi(t_1; t_0, x^*, p^*)$. The half width is found by integrating the *growth dynamics* $\dot{r} = g(r, p) = Cr + [p]$, where $[p] = \frac{1}{2}(\overline{p} - \underline{p})$, over $[t_0, t_1]$ with initial condition $r(t_0) = [x] = \frac{1}{2}(\overline{x} - \underline{x})$.

The *Mixed-Monotonicity* method [24] computes the reachable set by separating the increasing and decreasing portions of the system dynamics in an auxiliary system called the *embedding system* whose state dimension is twice that of the original system [25]. The embedding system is constructed using a *decomposition function* $d(t, x, p, \hat{x}, \hat{p})$, which encodes the increasing and decreasing parts of the system dynamics and satisfies $d(t, x, p, x, p) = f(t, x, p)$. The evaluation of a single trajectory of the embedding system can be used to find a reachable set over-approximation for the original system.

The *Monte Carlo* method computes a probabilistic approximation to the reachable set by evaluating the trajectories of a finite number m of pairs *sample points* $(x_0^{(i)}, p^{(i)})$ in the initial set and input set, and selecting the smallest interval that contains the final points of the trajectories. Unlike the other two methods, the Monte Carlo method is restricted to constant-valued inputs, i.e. inputs of the form $p(t) = p$, where $p \in [\underline{p}, \overline{p}]$. Each sampled initial state $x_0^{(i)}$ is integrated over $[t_0, t_1]$ with its input $p^{(i)}$ to yield a final state $x_1^{(i)}$. The interval reachable set is then approximated by the elementwise minimum and maximum

of the $x_1^{(i)}$. This approximation satisfies a probabilistic guarantee of correctness, provided that enough sample states are chosen [26]. Let $[\underline{R}, \overline{R}]$ be the approximated reachable set, $\epsilon, \delta \in (0, 1)$, and $m \geq (\frac{2n}{\epsilon}) \log\left(\frac{2n}{\delta}\right)$. Then, with probability $1-\delta$, the approximation $[\underline{R}, \overline{R}]$ satisfies $P(R_{t_0,t_1} \setminus [\underline{R}, \overline{R}]) \leq \epsilon$, where $P(A)$ denotes the probability that a sampled initial state will yield a final state in the set A, and $\setminus$ denotes set difference. The probability that a sampled initial state will be sent to a state outside the estimate (the "accuracy" of the estimate) is quantified by ϵ. Improved accuracy (lower ϵ) increases the sample size as $O(1/\epsilon)$. The probability that running the algorithm will fail to give an estimate satisfying the inequality (The "confidence") is quantified by δ. Improved confidence (lower δ) increases the sample size by $O(\log(1/\delta))$.

3 Parallelization

The bulk of the computational work in each method is spent in ODE integration. Hence, the most effective approach by which to parallelize the three methods is to design a parallel ODE integration method. There are several available methods for parallelizing the task of ODE integration. Several popular methods for parallel ODE integration are parallel extensions of Runge-Kutta integration methods, which are the most popular serial methods for ODE integration.

`PIRK` takes advantage of the task-level parallelism in the Runge-Kutta equations by evaluating each state dimension in parallel. This parallelization scheme is called *parallelization across space* [27]. `PIRK` specifically uses a space-parallel version of the fourth-order Runge-Kutta method, or space-parallel RK4 for brevity. In space-parallel RK4, each parallel thread is assigned a different state variable to evaluate the intermediate update equations. After each intermediate step, the threads must synchronize to construct the updated state in global memory. Space-parallel RK4 can use as many parallel computation elements as there are state variables: since `PIRK`'s goal is to compute reachable sets for extremely high-dimensional systems, this is sufficient in most cases.

The space-parallel scheme is not hardware-specific, and may be used with any parallel computing platform. `PIRK` is similarly hardware-agnostic: the `pFaces` ecosystem, for which `PIRK` is a kernel, provides a common interface to run on a variety of heterogeneous parallel computing platforms. The only difference between platforms that affects `PIRK` is the number of available parallel processing elements (PEs).

4 Complexity of the Parallelized Methods

The parallelized implementations of the three reachability methods described in Sect. 2.1 use space-parallel RK4 to perform almost all computations other than setting up initial conditions. We can therefore find the time and memory complexity of each method by analyzing the complexity of space-parallel RK4 and counting the number of times each method uses it.

For a system with n dimensions, space-parallel RK4 scales linearly as the number of PEs (denoted by P) increases. In a computer with a single PE (i.e., $P = 1$), the algorithm reduces to the original serial algorithm. Then, suppose that a parallel computer has $P \leq n$ PEs of the same type. We assume a computational model under which instruction overhead and latency from thread synchronization are negligible, memory space has equal access time from all processing elements, and the number of parallel jobs can be evenly distributed among the P processing elements.[2] Under this *parallel random-access machine* model [28], the time complexity of space-parallel RK4 is reduced by a factor of P: each PE is responsible for computing n/P components of the state vector. Therefore, for fixed initial and final times t_0 and t_1, the time complexity of the algorithm is $O(\frac{n}{P})$.

The parallel version of the contraction/growth bound method uses space-parallel RK4 twice. First, it is used to compute the solution of the system's ODE f for the center of the initial set $\mathcal{X}_0$. Then, it is used to compute the growth/contraction of the initial set $\mathcal{X}_0$ by solving the ODE g of the growth dynamics. Since this method uses a fixed number of calls of space-parallel RK4, its time complexity is also $O(\frac{n}{P})$ for a given t_0 and t_1.

The parallelized implementation of the mixed-monotonicity method uses space-parallel RK4 only once, in order to integrate the $2n$-dimensional embedding system. This means that the mixed-monotonicity method also has a time complexity of $O(\frac{n}{P})$ for fixed t_0 and t_1. However, the mixed-monotonicity method requires twice as much memory as the growth bound method, since it runs space-parallel RK4 on a system of dimension $2n$.

The parallelized implementation of the Monte Carlo method uses space-parallel RK4 m times, once for each of the m sampled initial states. The implementation uses two levels of parallelization. The first level is a set of parallel threads over the samples used for simulations. Then, within each thread, another parallel set of threads are launched by space-parallel RK4. This is realized as one parallel job of $m \times n$ threads. Consequently, the Monte Carlo method has a complexity of $O(\frac{mn}{p})$. Since only the elementwise minima and maxima of the sampled states need to be stored, this method only requires as much memory as the growth bound method.

Remark 1. A pseudocode of each parallel algorithm and a detailed discussion of their time and space complexities are provided in an extended version of this paper [29]. The extended version also contains additional details for the case studies that will be presented in the next section.

5 Case Studies

In each of the case studies to follow, we report the time it takes PIRK to compute reachable sets for systems of varying dimension using all three of its methods on

[2] While these non-idealities will be present in real systems and slow down computation, they should not affect the asymptotic complexity.

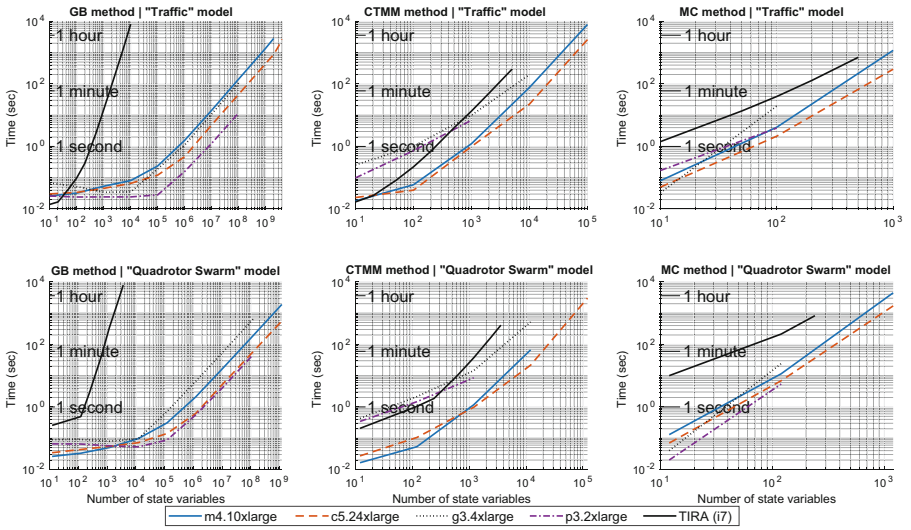


Fig. 2. Logarithmic plots of the results for speed tests of the traffic model (first row) and the quadrotor swarm (second row). Speed test results for the serial interval reachability toolbox `TIRA` are also shown for the traffic model.

a variety of parallel computing platforms. We perform some of the same tests using the serial tool `TIRA`, to measure the speedup gained by `PIRK`'s ability to use massively parallel hardware.

We set a time limit of 1 h for all of the targeted case studies, and report the maximum dimensions that could be reached under this limit. The Monte Carlo method is given probabilistic parameters $\epsilon = \delta = 0.05$ in each case study where it is used. We use four AWS machines for the computations with `PIRK`: `m4.10xlarge` which has a CPU with 40 cores, `c5.24xlarge` which has a CPU with 96 cores, `g3.4xlarge` which has a GPU with 2048 cores, and `p3.2xlarge` which has a GPU with 5120 cores. For the computations with `TIRA`, we used a machine with a 3.6 GHz Intel i7 CPU.

5.1 *n*-link Road Traffic Model

We consider the road traffic analysis problem reported in [30], a proposed benchmark for formal controller synthesis. We are interested in the density of cars along a single one-way lane. The lane is divided into n segments, and the density of cars in each segment is a state variable. The continuous-time dynamics are derived from a spatially discretized version of the Cell Transmission Model [31]. This is a nonlinear system with sparse coupling between state variables.

The results of the speed test are shown in the first row of Figure 2. The machines `m4.10xlarge` and `c5.24xlarge` reach up to 2 billion and 4 billion dimensions, respectively, using the growth/contraction method, in 47.3 min and 44.7 min, respectively. Due to memory limitations of the GPUs, the machines

g3.4xlarge and p3.2xlarge both reach up to 400 million in 106 s and 11 s, respectively.

The relative improvement of PIRK's computation time over TIRA's is significantly larger for the growth bound method than for the other two. This difference stems from how each tool computes the half-width of the reachable set from the radius dynamics. TIRA solves the radius dynamics by computing the full matrix exponential using MATLAB's expm, whereas PIRK directly integrates the dynamics using parallel Runge-Kutta. This caveat applies to Sect. 5.2 as well.

5.2 Quadrotor Swarm

The second test system is a swarm of K identical quadrotors with nonlinear dynamics. The system dynamics of each quadrotor model are derived in a similar way to the model used in the ARCH-COMP 18 competition [32], with the added simplification of a small angle approximation in the angular dynamics and the neglect of Coriolis force terms. A derivation of both models is available in [33]. Similar to the n-link traffic model, this system is convenient for scaling: system consisting of one quadrotor can be expressed with 12 states, so the state dimension of the swarm system is $n = 12K$. While this reachability problem could be decomposed into K separate reachability problems which can be solved separately, we solve the entire $12K$-dimensional problem as a whole to demonstrate PIRK's ability to make use of sparse interconnection.

The results of the speed test are shown in Fig. 2 (second row). The machines m4.10xlarge and c5.24xlarge reach up to 1.8 billion dimensions and 3.6 billion dimensions, respectively, (using the growth/contraction method) in 48 min and 32 min, respectively. The machines g3.4xlarge and p3.2xlarge both reach up to 120 million dimensions in 10.6 min and 46 s, respectively.

5.3 Quadrotor Swarm with Artificial Potential Field

The third test system is a modification of the quadrotor swarm system which adds interactions between the quadrotors. In addition to the quadrotor dynamics described in Sect. 5.2, this model augments each quadrotor with an artificial potential field to guide it to the origin while avoiding collisions. This controller applies nonlinear force terms to the quadrotor dynamics that seek to minimize an *artificial potential* U that depends on the position of all of the quadrotors. Due to the interaction of the state variables in the force terms arising from the potential field, this system has a dense Jacobian. In particular, at least 25% of the Jacobian elements will be nonzero for any number of quadrotors.

Table 1 shows the times of running PIRK using this system on the four machines m4.10xlarge, c5.24xlarge, g3.4xlarge and p3.2xlarge in Amazon AWS. Due to the high density of this example, we focus on the memory-light growth bound and the Monte-Carlo methods. PIRK computed the reach sets of systems up to 120,000 state variables (i.e., 10,000 quadrotors). Up to 1,200 states, all machines solve the problems in less than one second. Some of the

Table 1. Results for running `PIRK` to compute the reach set of the quadrotors swarm with artificial potential field. "N/M" means that the machine did not have enough memory to compute the reachable set.

Method	No. of states	Memory (MB)	Time (seconds)			
			`m4.10xlarge`	`c5.24xlarge`	`g3.4xlarge`	`p3.2xlarge`
GB	1200	2.8	≤ 1.0	≤ 1.0	≤ 1.0	≤ 1.0
GB	12000	275.3	≤ 1.0	≤ 1.0	≤ 1.0	≤ 1.0
GB	120000	27,473.1	69.6	68.3	N/M	N/M
MC	1200	45.7	1.0	≤ 1.0	2.0	≤ 1.0
MC	12000	457.5	56.8	23.7	233.1	40.6
MC	120000	4577.6	$\geq$ 2h	3091.8	N/M	5081.0

machines lack the required memory to solve the problems requiring large memory (e.g., 27.7 GB of memory is required to compute the reach set of the system with 120,000 state variables using the growth bound method).

5.4 Heat Diffusion

The fourth test system is a model for the diffusion of heat in a 3-dimensional cube. The model is based on a benchmark used in [7] to test a method for numerical verification of affine systems. A model of the form $\dot{x} = f(t, x, p)$ which approximates the heat transfer through the cube according to the heat equation can be obtained by discretizing the cube into an $\ell \times \ell \times \ell$ grid, yielding a system with ℓ^3 states. The temperature at each grid point is taken as a state variable. Each spatial derivative is replaced with a finite-difference approximation. Since the heat equation is a linear PDE, the discretized system is linear.

We take a fixed state dimension of $n = 10^9$ by fixing $\ell = 1000$. Integration takes place over $[t_0, t_1] = [0, 20]$ with time step size $h = 0.02$. Using the Growth bound method, `PIRK` solves the problem on `m4.10xlarge` in 472 min, and in 350.2 min on `c5.24xlarge`. This is faster than the time reported in [7] (30 h) using the same machine.

5.5 Overtaking Maneuver with a Single-Track Vehicle

The remaining case studies focus on models of practical importance with low state dimension. Although `PIRK` is designed to perform well on high-dimensional systems, it is also effective at quickly computing reachable sets for low dimensional systems, for applications that require many reachable sets. The first such case study is single-track vehicle model with seven states, presented in [34].

We fix an input that performs a maneuver to overtake an obstacle in the middle lane of a 3-lane highway. To verify that the maneuver was safely completed, we compute reachable sets over a range of points and ensuring that the reachable set does not intersect any obstacles. We consider a step-size of 0.005 s in a time window between 0 and 6.5 s. We compute one reachable set at each time step, resulting in a "reachable tube" comprising 1300 reachable sets. `PIRK`

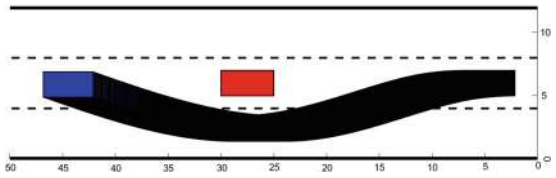


Fig. 3. Reachable tube for the single-track vehicle.

computed the reachable tube in 0.25 s using the growth bound method on an i7 CPU (Fig. 3).

5.6 Performance on ARCH Benchmarks

In order to compare PIRK's performance to existing tools, we tested PIRK's growth bound implementation on three systems from the ARCH-COMP'18 category report for systems with nonlinear dynamics [32]. This report contains benchmark data from several popular reachability analysis tools (C2E2, CORA, Flow*, Isabelle, SpaceEx, and SymReach) on nonlinear reachability problems with state dimensions between 2 and 12.

Table 2. Results from running PIRK (growth bound method) to compute the reach sets for the examples reported in the ARCH-2018 competition.

Benchmark model	PIRK	CORA	CORA/SX	C2E2	Flow*	Isabelle	SymReach
Van der Pol (2 states)	0.13	2.3	0.6	38.5	1.5	1.5	17.14
Laub-Loomis (7 states)	0.04	0.82	0.85	0.12	4.5	10	1.93
Quadrotor (12 states)	0.01	5.2	1.5	–	5.9	30	2.96

Table 2 compares the computation times for PIRK on the three systems to those reported by other tools in [32]. All times are in seconds. PIRK ran on an i9 CPU, while the others ran on i7 and i5: see [32] for more hardware details. PIRK solves each of the benchmark problems faster than the other tools. Both of the i7 and i9 processors used have 6 to 8 cores: the advantage of PIRK is its ability to utilize all available cores.

6 Conclusion

Using a simple parallelization of interval reachability analysis techniques, PIRK is able to compute reachable sets for nonlinear systems faster and at higher dimensions than many existing tools. This performance increase comes from PIRK's ability to use massively parallel hardware such as GPUs and CPU clusters, as well as the use of parallelizable simulation-based methods. Future work will

focus on improving the memory-usage of the mixed monotonicity and Monte-Carlo based methods, including an investigation of adaptive sampling strategies, and on using `PIRK` as a helper tool to synthesize controllers for high-dimensional systems.

References

1. Zamani, M., et al.: Symbolic models for nonlinear control systems with- out stability assumptions. IEEE Trans. Autom. Control **57**(7), 1804–1809 (2012)
2. Belta, C., Yordanov, B., Ebru, G.: Formal Methods for Discrete-Time Dynamical Systems. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-50763-7
3. Tabuada, P.: Verification and Control of Hybrid Systems: A Symbolic Approach. Springer, Cham (2009). https://doi.org/10.1007/978-1-4419-0224-5
4. Reißig, G., Weber, A., Rungger, M.: Feedback refinement relations for the synthesis of symbolic controllers. IEEE Trans. Autom. Control **62**(4), 1781–1796 (2017)
5. Althoff, M.: An introduction to CORA 2015. In: Proceedings of the Workshop on Applied Verification for Continuous and Hybrid Systems (2015)
6. Frehse, G., et al.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_30
7. Bak, S., Tran, H.-D., Johnson, T.T.: Numerical verification of affine systems with up to a billion dimensions. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 23–32. ACM (2019)
8. Kurzhanskiy, A.A., Varaiya, P.: Ellipsoidal toolbox (ET). In: Proceedings of the 45th IEEE Conference on Decision and Control, pp. 1498–1503. IEEE (2006)
9. Dreossi, T.: SAPO: reachability computation and parameter synthesis of polynomial dynamical systems. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, pp. 29–34. ACM (2017)
10. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18
11. The CAPD Group. Computer Assisted Proofs in Dynamics group, a C++ package for rigorous numerics (2019). http://capd.ii.uj.edu.pl/. Accessed 20 Oct 2019
12. Duggirala, P.S., Mitra, S., Viswanathan, M., Potok, M.: C2E2: a verification tool for stateflow models. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 68–82. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_5
13. Nedialkov, N.S.: Implementing a rigorous ODE solver through literate programming. In: Rauh, A., Auer, E. (eds.) Modeling, Design, and Simulation of Systems with Uncertainties. Mathematical Engineering, vol. 3. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-15956-5_1
14. Sandretto, J.A.D., Chapoutot, A.: Validated explicit and implicit Runge-Kutta methods (2016)
15. Meyer, P.-J., Devonport, A., Arcak, M.: TIRA: toolbox for interval reachability analysis. In: arXiv preprint arXiv:1902.05204 (2019)
16. Ray, R., et al.: XSpeed: accelerating reachability analysis on multi- core processors. In: Piterman, N. (eds.) Hardware and Software: Verification and Testing, pp. 3–18. Springer, Cham (2015). ISBN: 978-3-319-26287-1

17. Huang, Z., Mitra, S.: Computing bounded reach sets from sampled simulation traces. In: Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control, pp. 291–294. ACM (2012)
18. Julius, A.A., Pappas, G.J.: Trajectory based verification using local finite-time invariance. In: Majumdar, R., Tabuada, P. (eds.) HSCC 2009. LNCS, vol. 5469, pp. 223–236. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00602-9_16
19. Maidens, J., Arcak, M.: Reachability analysis of nonlinear systems using matrix measures. IEEE Trans. Autom. Control **60**(1), 265–270 (2014)
20. Khaled, M., Zamani, M.: pFaces: an acceleration ecosystem for symbolic control. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 252–257. ACM (2019)
21. Kapela, T., Zgliczyński, P.: A Lohner-type algorithm for control systems and ordinary differential inclusions. In: arXiv preprint arXiv:0712.0910 (2007)
22. Fan, C., et al.: Locally optimal reach set over-approximation for non-linear systems. In: 2016 International Conference on Embedded Software (EMSOFT), pp. 1–10. IEEE (2016)
23. Arcak, M., Maidens, J.: Simulation-based reachability analysis for nonlinear systems using componentwise contraction properties. In: Lohstroh, M., Derler, P., Sirjani, M. (eds.) Principles of Modeling. LNCS, vol. 10760, pp. 61–76. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95246-8_4
24. Coogan, S., Arcak, M., Belta, C.: Formal methods for control of traffic flow: Automated control synthesis from finite-state transition models. IEEE Control Syst. Mag. **37**(2), 109–128 (2017)
25. Gouzé, J.-L., Hadeler, K.P.: Monotone flows and order intervals. In: Nonlinear World 1, pp. 23–34 (1994)
26. Devonport, A., Arcak, M.: Data-Driven Reachable Set Computation using Adaptive Gaussian Process Classification and Monte Carlo Methods (2019). arXiv: 1910.02500 [eess.SY]
27. Solodushkin, S.I., Iumanova, I.F.: Parallel numerical methods for ordinary differential equations: a survey. In: CEUR Workshop Proceedings, vol. 1729. CEUR-WS, pp. 1–10 (2016)
28. Jaja, J.: An Introduction to Parallel Algorithms. Addison-Wesley, Reading (1992)
29. Devonport, A., et al.: PIRK: Scalable Interval Reachability Analysis for High-Dimensional Nonlinear Systems (2020). arXiv: 2001.10635 [eess.SY]
30. Coogan, S., Arcak, M.: A benchmark problem in transportation networks. In: arXiv preprint arXiv:1803.00367 (2018)
31. Coogan, S., Arcak, M.: A compartmental model for traffic networks and its dynamical behavior. IEEE Trans. Autom. Control **60**(10), 2698–2703 (2015)
32. Immler, F., et al.: ARCH-COMP18 category report: continuous and hybrid systems with nonlinear dynamics. In: Proceedings of the 5th International Workshop on Applied Verification for Continuous and Hybrid Systems (2018)
33. Beard, R.: Quadrotor dynamics and control rev 0.1. Technical report Brigham Young University (2008)
34. Althoff, M.: CommonRoad: vehicle models. In: Technische Universität München, Garching, pp. 1–25 (2017)

AEON: Attractor Bifurcation Analysis of Parametrised Boolean Networks

Nikola Beneš, Luboš Brim, Jakub Kadlecaj, Samuel Pastva(✉), and David Šafránek

Faculty of Informatics, Masaryk University, Brno, Czech Republic
{xbenes3,brim,xkadlec2,xpastva, safranek}@fi.muni.cz

Abstract. Boolean networks (BNs) provide an effective modelling tool for various phenomena from science and engineering. Any long-term behaviour of a BN eventually converges to a so-called attractor. Depending on various logical parameters, the structure and quality of attractors can undergo a significant change, known as a bifurcation. We present a tool for analysing bifurcations in asynchronous parametrised Boolean networks. To fight the state-space and parameter-space explosion problem the tool uses a parallel semi-symbolic algorithm.

Keywords: Boolean networks · Attractors · Bifurcation analysis

1 Introduction

Boolean networks (BNs) provide an effective mathematical tool to model computational processes and other phenomena from science and engineering. BNs represent a generalisation of other relevant mathematical models, which appeared previously as cellular automata (CA), suggested by Wolfram [39] for computation modelling, or formal genetic nets [24] and Thomas networks [37], proposed for gene regulatory networks. This gives an idea of the versatility of BNs in different applications (mathematics, physics chemistry, biology, ecology, etc.) and engineering (computation, artificial intelligence, electronics, circuits, etc.).

The development of formal methods for analysis and synthesis of Boolean networks has recently attracted a lot of attention [11,18,20,28,36]. In this paper, we are primarily interested in BN models for computational systems biology [29]. In general, biological processes are emerging from complex inter- and intra-cellular interactions and they cannot be sufficiently understood and controlled without the help of powerful computer-aided modelling and analysis methods [38]. BNs serve an important purpose of describing overall interactions within a living cell at an appropriate level of abstraction and they provide a systematic approach to model crucial states of cell dynamics – so-called *phenotypes* [22].

Supported by the Czech Science Foundation grant No. 18-00178S.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 569–581, 2020.
https://doi.org/10.1007/978-3-030-53288-8_28

The level of abstraction provided by BNs makes them an important tool for design of targetted therapeutic procedures such as cell reprogramming [36] based on changing one cell phenotype to another, allowing regeneration of tissues or neurons [21]. Since phenotypes are determined by long-term behaviour of biological systems, fully automatised identification of phenotypes by employing BN models is a necessary step towards the future of modern medicine. Owing to the fact there is a continuous lack of sufficiently detailed (mechanistic) information on biological processes, there is definitely a need to work with models involving uncertain (or insufficient) knowledge. In this paper, we present a unique tool that makes a significant contribution towards fully automatised analysis of long-term behaviour of BN models with uncertain knowledge.

We start with giving some intuition on BNs. A BN consists of a set of Boolean *variables* whose state is determined by other variables in the network through a set of Boolean *update functions* assigned to the variables (different update functions can be assigned to different variables) and *regulations* placed on them. If at each point of time all the update functions are applied simultaneously we speak about *synchronous dynamics*, if only one of the update functions is chosen non-deterministically to modify the corresponding Boolean variable, we speak of *asynchronous dynamics*. In this paper we consider asynchronous Boolean networks only.

In real-world applications, the update functions for some of the variables are typically (partially) unknown and are represented as logical *parameters* of the network. We speak of *parametrised Boolean networks* [40] in this case. If all the parameters are fixed to a concrete Boolean function, a parametrised BN turns into a (non-parametrised) BN.

The long-term behaviour of a BN, starting from an initial state, has three possible outcomes. Briefly, the first situation is when the network evolves to a single stable state. Such states are the fixed points or *point attractors* or *stable states*. The second situation is that the network periodically oscillates through a finite sequence of states—an *oscillating attractor* or *attractive cycle* (the discrete equivalent of a limit cycle in continuous systems). The third case is what we call a *disordered attractor* (or chaotic oscillation [32]), an attractor that is neither stable not periodically oscillating and in which the system may behave unpredictably, due to the nondeterminism of the asynchronous semantics of BNs. Attractors are particularly relevant in the context of biological modelling as they are used to represent differentiated cellular types or tissues (in the case of fixed points) [2] and biological rhythms or oscillations (in the case of cycles) [17].

The set of network states that converge to the same attractor forms the *basin of attraction* of that attractor [7]. Attractors (and their basins) are disjoint entities and the state space is compartmentalised by imaginary "attractor boundaries". The entire dynamics of a Boolean network can be represented as a state transition system in which the trajectories from initial states are depicted, revealing the basins of attraction and associated attractors. We call such a representation the *attractor landscape* of the network [13].

In parametrised BNs the attractor landscape changes as the parameters are varied. Some of these changes may lead to a qualitatively different landscape (defined, e.g., in the count and/or quality of attractors). Such a qualitative

change is called a *bifurcation* and the values of parameters for which it occurs are called *bifurcation points*. Determining (all) bifurcation points for a network, called *attractor bifurcation analysis*, is an important task in the analysis of BNs [4].

While BN models are intuitive, mathematically simple to describe, and supported by analytical methods [12], analysis of large models appearing in real cases is severely limited by the lack of robust computational tools running efficiently on high-performance hardware. Several computational tools have been developed for construction, visualisation and analysis of attractors in non-parametrised BNs. Amongst them, the established tools include ATLANTIS [34], Bio Model Analyzer (BMA) [6], BoolNet [31], PyBoolNet [27], lnet [7], The Cell Collective [23], CellNetAnalyzer [25], and ASSA-PBN [30]. Another group of existing tools targets the parameter synthesis problem for parametrised BNs. The most prominent tools here are GRNMC [20], GINsim [10] (indirectly through NuSMV [14]), and TREMPPI [35]. In general, parameter synthesis tools can be used to identify parameters producing a specified long-term behaviour (depending on the logics employed), however, they do not provide a sufficient solution for identification and classification of all attractors in the system. Finally, it is worth noting that there have recently appeared several tools aiming at control of cell behaviour through BNs (i.e., driving a cell into the desired state). A well-known representative of these tools is ViSiBooL [33].

To the best of our knowledge, none of the existing tools is capable of performing attractor bifurcation analysis in parametrised models. Bifurcation analysis has been recently recognised as a fundamental approach that provides a new framework for understanding the behaviour of biological networks. The ability to make a dramatic change in system behaviour is often essential to organism function, and bifurcations are therefore ubiquitous in biological networks such as the switches of the cell cycle. The tool AEON is supposed to fill in the gap in the existing tools supporting analysis of Boolean network models.

AEON builds on methods and algorithms for asynchronous parametrised BNs we have introduced in our previous research [1,3–5]. To deal with the state-space and parameter-space explosion problem, the tool implements a shared-memory parallel semi-symbolic algorithm. The results the tool provides to the user can be used for example to the design of "wet" experiments, better understanding of the system's dynamics, or to control or re-program the system. As attractors model phenotypes, one of the most urgent needs for computer aided support, such as AEON can provide, is in applications in therapeutic innovations.

We believe that attractor bifurcation computed by AEON will shift the current technology toward a comprehensive method when integrated with tools aimed at control or other analysis methods.

2 Attractors in Parametrised Boolean Networks

In this section, we define precisely the problem of attractor bifurcation analysis. We also give an overview of the necessary technical background needed to

describe the algorithmic solution and its implementation. More details can be found in [4].

A *Boolean network (BN)* consists of a finite set of *state variables* $\mathcal{V}$ (whose elements we denote by $\mathtt{A}$, $\mathtt{B}$, ...), a set of *regulations* $R \subseteq \mathcal{V} \times \mathcal{V}$, and a family of Boolean *update functions* $\mathcal{F} = \{F_{\mathtt{A}} \mid \mathtt{A} \in \mathcal{V}\}$. If $(\mathtt{B}, \mathtt{A}) \in R$, we say that $\mathtt{B}$ is a *regulator* of $\mathtt{A}$. For each $\mathtt{A} \in \mathcal{V}$, we call the set $\mathcal{C}(\mathtt{A}) = \{\mathtt{B} \in \mathcal{V} \mid (\mathtt{B}, \mathtt{A}) \in R\}$ of its regulators the *context* of $\mathtt{A}$. A *state* of the BN is an assignment of Boolean values to the variables, i.e. a function $\mathcal{V} \to \{0, 1\}$. The type signature of each update function $F_{\mathtt{A}}$ is given by the context of $\mathtt{A}$ as $F_{\mathtt{A}} : \{0, 1\}^{\mathcal{C}(\mathtt{A})} \to \{0, 1\}$.

In Boolean networks, one often describes various properties of the network regulations. Here, we focus on three most basic types of regulation: We say that $(\mathtt{A}, \mathtt{B}) \in R$ is *observable* if there exists a state where changing the value of $\mathtt{A}$ also changes the value of $F_{\mathtt{B}}$. In the tool, edges that might be non-observable are drawn using dashed lines.

We say that a regulation $(\mathtt{A}, \mathtt{B}) \in R$ is *activating* if by increasing $\mathtt{A}$, one cannot decrease the value of $F_{\mathtt{B}}$. Symmetrically, the regulation is *inhibiting* if by increasing $\mathtt{A}$, one cannot increase the value of $F_{\mathtt{B}}$. In the tool, activating edges are denoted using green colour and sharp arrow tips, inhibiting edges are denoted using red colour and flat arrow tips, and edges that might be neither activating nor inhibiting are denoted using grey colour.

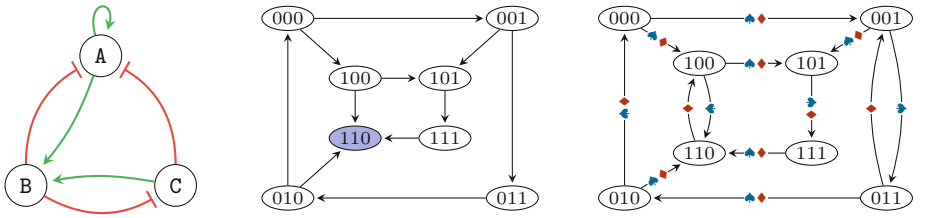


Fig. 1. Illustration of a (parametrised) BN and its state transition graph. (Color figure online)

Let us now consider an example of a BN with $\mathcal{V} = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$, the regulations R as denoted in Fig. 1 (left) and the update functions: $F_{\mathtt{A}} = \mathtt{A} \vee \neg\mathtt{B} \vee \neg\mathtt{C}$, $F_{\mathtt{B}} = \mathtt{A} \vee \mathtt{C}$, $F_{\mathtt{C}} = \neg\mathtt{B}$. We can see that all regulations are observable and the colour (and shape) of the arrows respects the properties of activation and inhibition, e.g. $(\mathtt{B}, \mathtt{A})$ is an inhibition, because by increasing the value of $\mathtt{B}$, we cannot increase the value of $F_{\mathtt{A}}$.

The semantics of a Boolean network is given as a directed *state transition graph*. The state space of the graph is the set of all possible assignments of Boolean values to the variables, i.e. $\{0, 1\}^{\mathcal{V}}$. We consider the state of the Boolean network to evolve in an *asynchronous* manner, i.e. each variable is updated independently. We thus add a transition $s \to t$ if $s \neq t$ and if there exists a variable $\mathtt{A}$ such that $t(\mathtt{A}) = F_{\mathtt{A}}(s)$ and $t(\mathtt{X}) = s(\mathtt{X})$ for all $\mathtt{X} \in \mathcal{V} \setminus \{\mathtt{A}\}$. We

also use the notation $\rightarrow^*$ to denote the reflexive and transitive closure of $\rightarrow$, i.e. $s \rightarrow^* t$ means that the state t is reachable from the state s.

The semantics of the BN from our example is illustrated in Fig. 1 (middle). The states are represented as Boolean triples denoting the values assigned to the variables $\mathtt{A}$, $\mathtt{B}$, and $\mathtt{C}$, respectively.

The long-term behaviour that we are interested in is captured by the notion of *attractors*. In discrete-state systems, attractors are represented by terminal strongly connected components (TSCCs) of the graph. A TSCC is a maximal set of states S such that for all $s, t \in S$, $s \rightarrow^* t$, and for all $s \in S$, $s \rightarrow t$ implies $t \in S$.

To classify the attractors of a given BN, we consider three primary kinds of long-term behaviour:

- *stability* ($\odot$) We say that an attractor is stable, if it consists of a single state, in which the network stays forever.
- *oscillation* ($\circlearrowleft$) We consider an attractor to be oscillating if it is a single cycle of states. The size of such cyclic attractor is often referred to as its *period*.
- *disorder* ($\rightleftarrows$) Finally, an attractor is said to be disordered if it is neither stable nor oscillating. This means that although the network will stay in the attractor forever, it will behave somewhat unpredictably due to nondeterminism.

The long-term behaviour of a BN is then characterised by a multi-set over the universe of the three behaviours $\{\odot, \circlearrowleft, \rightleftarrows\}$. We call such multi-set a *behaviour class* and we denote the set of all possible behaviour classes $\mathfrak{C}$. In our example, the BN has only one attractor, and this attractor is stable; it consists of the single state 110, see Fig. 1 (middle).

To deal with the fact that the update function family $\mathcal{F}$ might not be fully known, we extend the Boolean network with a set of *logical parameters* which determine the exact behaviour of each update function. These parameters have the form of uninterpreted Boolean functions, which can be used as part of the update functions' description.

Formally, we assume a finite set of *parameter names* $\mathfrak{P}$, whose elements we denote by $\mathtt{P}$, $\mathtt{Q}$, ...; we assume that every $\mathtt{P} \in \mathfrak{P}$ has an associated arity $a_{\mathtt{P}}$ meaning that $\mathtt{P}$ is an $a_{\mathtt{P}}$-ary uninterpreted function over Boolean values. Note that nullary uninterpreted functions are also allowed and can be seen as simply Boolean parameters. We call an interpretation that assigns to each $\mathtt{P} \in \mathfrak{P}$ an $a_{\mathtt{P}}$-ary Boolean function a *parametrisation*. We usually work with a subset of parametrisations, called the *valid* parametrisations and denoted by P.

A *parametrised Boolean network* consists of a set of variables $\mathcal{V}$, a set of regulations $R \subseteq \mathcal{V} \times \mathcal{V}$ as in the non-parametrised case, a set of parameter names $\mathfrak{P}$, its associated set of valid parametrisations P, and a family of *parametrised update functions* $\mathfrak{F} = \{\widehat{F}_{\mathtt{A}} \mid \mathtt{A} \in \mathcal{V}\}$. Each $\widehat{F}_{\mathtt{A}}$ is written as a Boolean expression that may contain the uninterpreted functions of $\mathfrak{P}$.

Let us now modify the previous example so that we view the BN from Fig. 1 (left) as a parametrised one with the following update functions: $\widehat{F}_{\mathtt{A}} = \mathtt{A} \vee \neg\mathtt{B} \vee \neg\mathtt{C}$, $\widehat{F}_{\mathtt{B}} = \mathtt{P}(\mathtt{A}, \mathtt{C})$, $\widehat{F}_{\mathtt{C}} = \neg\mathtt{B}$, where $\mathtt{P}$ is a parameter name with arity 2. The set

of valid parametrisations is constrained symbolically using the description of activations and inhibitions in Fig. 1 (left). In this case, there are only two possible parametrisations p_1 (denoted by ♠) and p_2 (denoted by ♦). The parametrisation p_1 assigns to P the function $(x, y) \mapsto x \vee y$, while p_2 assigns to P the function $(x, y) \mapsto x \wedge y$. Note that other assignments would violate the description, namely that both (A, B) and (C, B) are observable and activating.

By fixing a concrete parametrisation $p \in P$, we can interpret all the parameter names and thus transform the parametrised update functions into non-parametrised ones, obtaining a (non-parametrised) BN, called the *p-instantiation* of the parametrised BN. We then generalise the definition of attractors to parametrised BNs, saying that a set of states S is an *attractor in parametrisation* $p \in P$ if S is an attractor in the p-instantiation.

The asynchronous semantics of a parametrised BN can be described using an *edge-coloured* state transition graph. The transitions of this graph are assigned a set of so-called *colours*—in our case, the colours correspond exactly to the parametrisations. The states are given as in the non-parametrised case. We then say that $s \to t$ if there exists a parametrisation p such that $s \to t$ in the p-instantiation. The set of colours of $s \to t$ is the set of all such parametrisations. In our example, the graph is depicted in Fig. 1 (right; the edges are annotated with ♠, ♦, or both).

Problem Formulation. We now formulate the problem of *attractor bifurcation analysis of parametrised BN* as follows: Given a parametrised BN with a set of valid parametrisations P, compute the *bifurcation function* $\mathcal{A} : P \to \mathfrak{C}$ that assigns to each parametrisation p the behaviour class of the p-instantiation of the given parametrised BN.

In our example, the function $\mathcal{A}$ maps p_1 (♠) to $\{\odot\}$ (one stable attractor $\{110\}$) and p_2 (♦) to $\{\circlearrowleft\}$ (one oscillating attractor $\{100, 101, 111, 110\}$).

3 Attractor Bifurcation Analysis with AEON

The workflow of our approach, as implemented in the tool, is illustrated in Fig. 2. As an input, we take a parametrised BN including a graphical description of the regulations. The tool computes its asynchronous semantics as a symbolic edge-coloured graph represented using BDDs [8]. This is then used as an input to a parallel TSCC detecting algorithm based on [1], which extracts the attractors on the fly. Each attractor is classified as one of the three above-mentioned types and this information is used to incrementally build the bifurcation function $\mathcal{A}$, also represented symbolically using BDDs. More details about the algorithm as well as the classification procedure can be found in [4].

The bifurcation function induces a partitioning of the parameter space in which two parametrisations are equivalent if their p-instantiations have the same behaviour class. This partitioning is presented to the user as a list of behaviour classes together with the cardinality of the respective parameter space partitions, see Fig. 3. The user can select one of these classes and obtain a *witness BN*,

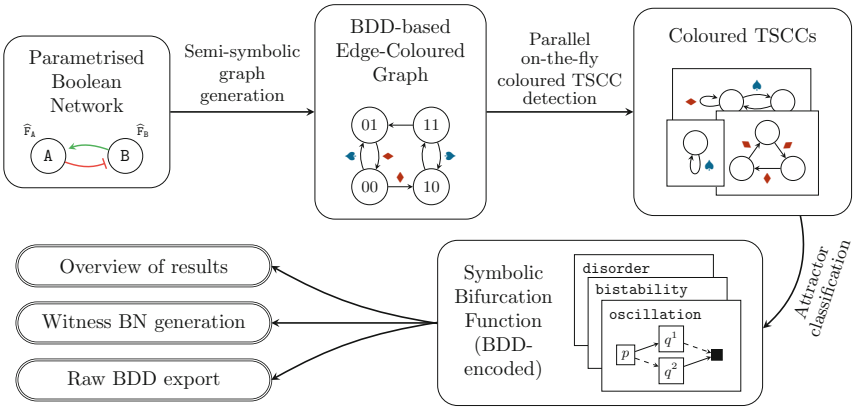


Fig. 2. The workflow of the AEON tool.

i.e. a p-instantiation of the parametrised BN where p is one of the corresponding parametrisations. Finally, the tool also provides the whole bifurcation function encoded as BDDs—this output can be used for post-processing by further tools.

4 Implementation

The tool architecture consists of two components as seen in Fig. 4: the *compute engine*, and a web-based, user-facing GUI application (*the client*). The engine is responsible for the actual computation and acts as a web server to which the client establishes a connection. Using web-based GUI enables portability across different platforms, and the separation of the user interface from the compute engine enables the user to run the computation remotely on high-performance hardware.

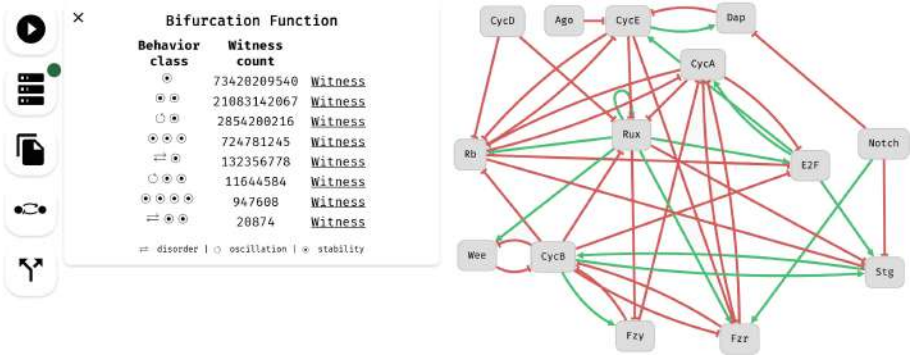


Fig. 3. Screenshot of the tool displaying a parametrised BN together with the bifurcation analysis results.

One of the responsibilities of the client is to provide a user friendly, multi-platform editor of parametrised BNs, since no popular BN editors currently support parameters. Architecturally, the client consists of several modules:

- `Live Model`: In-memory representation of the currently displayed model.
- `Compute Engine Connection` maintains the communication between the client and the compute engine.
- `Network Editor`: An interactive drag-and-drop editor for drawing the structure of the BN (variables, regulations). The implementation is based on the popular Cytoscape [19] library for graph visualisation and manipulation.
- `Parametrised BN Editor`: The update functions can be modified in a separate parametrised BN editor tab. This module is also responsible for basic integrity checks and static analysis of the BN, some of which is asynchronously deferred to the compute engine.
- `Import/Export` facilitates serialisation and transfer of the BNs to other tools. We currently provide a compact text-based format specifically designed for AEON and a universally adopted XML-based SBML level 3 qual standard [9].

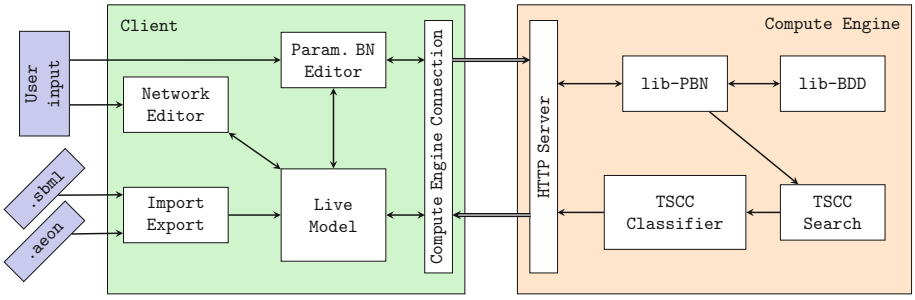


Fig. 4. Overview of the tool architecture showing the main components of the GUI client and the compute engine. Arrows represent the general flow of information between individual components.

The compute engine is written entirely in Rust to ensure fast and reliable operation (as well as easy portability). The functionality of the engine is split into separate libraries to allow later reuse:

- `lib-BDD`: Our own robust, thread-safe, scalable Rust-based implementation of BDDs.
- `lib-PBN`: A general purpose library for working with parametrised BNs. It provides serialisation to and from the AEON text format as well as SBML. Most importantly, it provides a parameter encoder that maps sets of parametrisations of the parametrised BN to BDDs. Using this encoder, the library implements an on-the-fly generation of the edge-coloured state transition graph corresponding to the asynchronous semantics of the given parametrised BN.

- `TSCC Search` algorithm implements the component search algorithm as presented in [1]. The algorithm uses parallel reachability procedures as well as asynchronous processing of independent parts of the state space to fully utilise available CPUs and thus speed up the computation. The algorithm is extended with appropriate cancellation points so that the user can stop the computation when needed.
- `TSCC Classifier` classifies and stores information about the discovered components. Specifically, for each non-empty behaviour class, we store a BDD representation of the parametrisations that result in this type of behaviour.

Aside from the general overview of the tool, we would like to highlight two additional aspects of AEON:

On-the-Fly Results: The attractors are discovered gradually. At any time during the computation the user may inspect the partial result, i.e. the bifurcation function computed so far. Although this is not the final outcome, such partial information can still prove useful, e.g. if unexpected attractor behaviour is found and the update functions of the model need to be adjusted.

SBML with Parameters: In our implementation, when dealing with fully instantiated networks, we always output valid SBML. Unfortunately, the current SBML standard does not allow parameters or uninterpreted functions inside the update function terms. In fact, the update functions in SBML are represented using MathML[1] which in general allows arbitrary mathematical expressions, but its use in SBML is restricted. To export parametrised BNs, we intentionally disregard the restriction and our tool produces MathML formulae with parameters. Note that existing SBML implementations can be easily extended to also support parametrised BNs, since they already contain MathML parsers.

Both the client[2] and the compute engine[3] are released as open source under the MIT License. Furthermore, an online version of the client is available at https://biodivine.fi.muni.cz/aeon/, including links to pre-built binaries of the computation engine for all major OSes.

5 Evaluation

We evaluated the efficiency and applicability of AEON tool on a set of real biological models taken from the GINsim model database [10], ranging from small toy examples to large real world models. The experiments were performed on a 32-core AMD Ryzen workstation with 64 GB of memory. All tested models are available in AEON source code repository (see footnote 3) as benchmark models.

[1] https://www.w3.org/TR/MathML3/.
[2] https://github.com/sybila/biodivine-aeon-client.
[3] https://github.com/sybila/biodivine-aeon-server.

The results are reported in Table 1. In general, the results show that the combination of symbolic representation of parametrisations and shared-memory parallel exploration of the state space allowed us to handle realistic BNs with large parameter spaces and non-trivial number of attractor bifurcations in reasonable time. Finally, let us note that the findings provided by AEON are in line with known properties of these biological models and even have a potential to provide new insights on the modelled biological processes.

Table 1. The evaluation results. Number of classes refers to the number of distinct behaviour classes discovered by the algorithm. The times in the form `minutes:seconds` refer to total runtime on 1 and two 32 CPU cores respectively.

Model name	State space size	Param. space size	No. of classes	Time (1cpu)	Time (32cpu)
Asymmetric Cell Division	2^5	$\sim 2^{18}$	11	`0:05.62`	`0:03.39`
Budding Yeast (Orlando)	2^9	$\sim 2^{18}$	6	`0:35.22`	`0:02.93`
TCR Signalisation	2^{10}	$\sim 2^{14}$	17	`0:26.61`	`0:04.42`
Drosophila Cell Cycle	2^{14}	$\sim 2^{36}$	8	`27:48.1`	`1:42.28`
Fission Yeast Cell Cycle	2^{10}	$\sim 2^{31}$	201	`25:20.9`	`4:00.29`
Mammalian Cell Cycle	2^{10}	$\sim 2^{44}$	176	`38:39.6`	`8:02.14`
Budding Yeast (Irons)	2^{18}	$\sim 2^{26}$	7	`Timeout`	`52:28.1`

In particular, in the case of the *TCR Signalisation* model, the authors have shown in [26] that their non-parametrised model produces seven possible stable states and one non-trivial attractor. By using AEON, we were able to confirm their findings as well as analyse a fully parametrised version of the model, finding sixteen other possible behaviours. Interestingly, in this model, all discovered seventeen behaviour classes consist of exactly eight attractors.

For the *Budding Yeast (Orlando)* model [16], the authors state that for several different parametrisations, the model always reaches a stable state (based on simulation). Our analysis performed with AEON has confirmed that the original instantiation of the model has indeed a single stable attractor. Moreover, we have found that in the fully parametrised version of the model, almost ninety thousand instantiations have a single stable attractor. Additionally, we have also found there is almost an equal number of instantiations producing disordered attractors and also several oscillating attractors. AEON is capable to generate witnesses

for all of these situations thus opening the biological questions targeting the existence of the corresponding phenotypes in nature.

The *Fission Yeast Cell Cycle* model [15] is known to contain one primary stable attractor as well as eleven artificial attractors. It is known that various multi-valued modifications of the original model exist that remove these artificial stable attractors from the model while preserving the only single stable attractor [16]. By parametrising the model adequately and applying our method using AEON, we have discovered that a large portion of the parameter space of the model also produces a single stable attractor.

References

1. Barnat, J., et al.: Detecting attractors in biological models with uncertain parameters. In: Feret, J., Koeppl, H. (eds.) CMSB 2017. LNCS, vol. 10545, pp. 40–56. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67471-1_3
2. Baudin, A., Paul, S., Su, C., Pang, J.: Controlling large Boolean networks with single-step perturbations. Bioinformatics **35**(14), i558–i567 (2019)
3. Beneš, N., Brim, L., Demko, M., Pastva, S., Šafránek, D.: A model checking approach to discrete bifurcation analysis. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 85–101. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_6
4. Beneš, N., Brim, L., Pastva, S., Poláček, J., Šafránek, D.: Formal analysis of qualitative long-term behaviour in parametrised Boolean networks. In: Ait-Ameur, Y., Qin, S. (eds.) ICFEM 2019. LNCS, vol. 11852, pp. 353–369. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32409-4_22
5. Beneš, N., Brim, L., Pastva, S., Šafránek, D.: Parallel parameter synthesis algorithm for hybrid CTL. Sci. Comput. Program. **185**, 102321 (2020)
6. Benque, D., et al.: Bma: visual tool for modeling and analyzing biological networks. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 686–692. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_50
7. Berntenis, N., Ebeling, M.: Detection of attractors of large boolean networks via exhaustive enumeration of appropriate subspaces of the state space. BMC Bioinformatics **14**, 361 (2013)
8. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
9. Chaouiya, C., et al.: SBML qualitative models: a model representation format and infrastructure to foster interactions between qualitative modelling formalisms and tools. BMC Syst. Biol. **7**(1), 135 (2013)
10. Chaouiya, C., Naldi, A., Thieffry, D.: Logical modelling of gene regulatory networks with GINsim. In: van Helden, J., Toussaint, A., Thieffry, D. (eds.) Bacterial Molecular Networks. Methods in Molecular Biology, vol. 804, pp. 463–479. Springer, New York (2012). https://doi.org/10.1007/978-1-61779-361-5_23
11. Chatain, T., Haar, S., Paulevé, L.: Boolean networks: beyond generalized asynchronicity. In: Baetens, J.M., Kutrib, M. (eds.) AUTOMATA 2018. LNCS, vol. 10875, pp. 29–42. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92675-9_3
12. Cheng, D., Qi, H., Li, Z.: Analysis and Control of Boolean Networks. CCE. Springer, London (2011). https://doi.org/10.1007/978-0-85729-097-7

13. Choi, M., Shi, J., Jung, S.H., Chen, X., Cho, K.H.: Attractor landscape analysis reveals feedback loops in the p53 network that control the cellular response to DNA damage. Sci. Signal. **5**(251), ra83 (2012)
14. Cimatti, A., et al.: NuSMV 2: an opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_29
15. Davidich, M.I., Bornholdt, S.: Boolean network model predicts cell cycle sequence of fission yeast. PloS ONE **3**, e1672 (2008)
16. Fauré, A., Thieffry, D.: Logical modelling of cell cycle control in eukaryotes: a comparative study. Mol. BioSyst. **5**(12), 1569–1581 (2009)
17. Feillet, C., et al.: Phase locking and multiple oscillating attractors for the coupled mammalian clock and cell cycle. Proc. Natl. Acad. Sci. **111**(27), 9828–9833 (2014)
18. Fisher, J., Köksal, A.S., Piterman, N., Woodhouse, S.: Synthesising executable gene regulatory networks from single-cell gene expression data. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 544–560. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_38
19. Franz, M., Lopes, C.T., Huck, G., Dong, Y., Sumer, O., Bader, G.D.: Cytoscape.js: a graph theory library for visualisation and analysis. Bioinformatics **32**(2), 309–311 (2016)
20. Giacobbe, M., Guet, C.C., Gupta, A., Henzinger, T.A., Paixão, T., Petrov, T.: Model checking the evolution of gene regulatory networks. Acta Informatica **54**(8), 765–787 (2016). https://doi.org/10.1007/s00236-016-0278-x
21. Graf, T., Enver, T.: Forcing cells to change lineages. Nature **7273**(462), 587–594 (2009)
22. Hartmann, A., Ravichandran, S., del Sol, A.: Modeling cellular differentiation and reprogramming with gene regulatory networks. In: Cahan, P. (ed.) Computational Stem Cell Biology. MMB, vol. 1975, pp. 37–51. Springer, New York (2019). https://doi.org/10.1007/978-1-4939-9224-9_2
23. Helikar, T., et al.: The cell collective: toward an open and collaborative approach to systems biology. BMC Syst. Biol. **6**(1), 96 (2012)
24. Kauffman, S.A.: Metabolic stability and epigenesis in randomly constructed genetic nets. J. Theor. Biol. **22**(3), 437–467 (1969)
25. Klamt, S., Saez-Rodriguez, J., Gilles, E.D.: Structural and functional analysis of cellular networks with CellNetAnalyzer. BMC Syst. Biol. **1**(1), 2 (2007)
26. Klamt, S., Saez-Rodriguez, J., Lindquist, J.A., Simeoni, L., Gilles, E.D.: A methodology for the structural and functional analysis of signaling and regulatory networks. BMC Bioinformatics **7**(1), 56 (2006)
27. Klarner, H., Streck, A., Siebert, H.: PyBoolNet: a Python package for the generation, analysis and visualization of Boolean networks. Bioinformatics **33**(5), 770–772 (2016)
28. Kolčák, J., Šafránek, D., Haar, S., Paulevé, L.: Parameter space abstraction and unfolding semantics of discrete regulatory networks. Theor. Comput. Sci. **765**, 120–144 (2019)
29. Le Novère, N.: Quantitative and logic modelling of molecular and gene networks. Nat. Rev. Genet. **16**, 146–158 (2015)
30. Mizera, A., Pang, J., Su, C., Yuan, Q.: ASSA-PBN: a toolbox for probabilistic Boolean networks. IEEE/ACM Trans. Comput. Biol. Bioinf. **15**(4), 1203–1216 (2018)
31. Müssel, C., Hopfensitz, M., Kestler, H.A.: BoolNet-an R package for generation, reconstruction and analysis of Boolean networks. Bioinformatics **26**(10), 1378–1380 (2010)

32. de Cavalcante, H.L.D.S., Gauthier, D.J., Socolar, J.E.S., Zhang, R.: On the origin of chaos in autonomous Boolean networks. Philos. Trans. R. Soc. A Math. Phys. Eng. Sci. **368**, 495–513 (2010)
33. Schwab, J.D., Kestler, H.A.: Automatic screening for perturbations in Boolean networks. Front. Physiol. **9**, 431 (2018)
34. Shah, O.S., et al.: ATLANTIS - attractor landscape analysis toolbox for cell fate discovery and reprogramming. Sci. Rep. **8**(1), 3554 (2018)
35. Streck, A., Thobe, K., Siebert, H.: Comparative statistical analysis of qualitative parametrization sets. In: Abate, A., Šafránek, D. (eds.) HSB 2015. LNCS, vol. 9271, pp. 20–34. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26916-0_2
36. Su, C., Paul, S., Pang, J.: Controlling large Boolean networks with temporary and permanent perturbations. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 707–724. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_41
37. Thomas, R.: Boolean formalization of genetic control circuits. J. Theor. Biol. **42**(3), 563–585 (1973)
38. Waddington, C.H.: Towards a theoretical biology. Nature **218**, 525–527 (1968)
39. Wolfram, S.: Cellular automata as models of complexity. Nature **311**, 419–424 (1984)
40. Zou, Y.M.: Boolean networks with multiexpressions and parameters. IEEE/ACM Trans. Comput. Biol. Bioinf. **10**, 584–592 (2013)

A Novel Approach for Solving the BMI Problem in Barrier Certificates Generation

Xin Chen[1], Chao Peng[2(✉)], Wang Lin[3(✉)], Zhengfeng Yang[2(✉)], Yifang Zhang[1], and Xuandong Li[1]

[1] State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
{chenxin,zhangyifan,lxd}@nju.edu.cn
[2] Shanghai Key Lab of Trustworthy Computing, East China Normal University, Shanghai, China
{cpeng,zfyang}@sei.ecnu.edu.cn
[3] School of Information Science and Technology, Zhejiang Sci-Tech University, Hangzhou, China
linwang@zstu.edu.cn

Abstract. Barrier certificates generation is widely used in verifying safety properties of hybrid systems because of the relatively low computational complexity it costs. Under sum of squares (SOS) relaxation, the problem of barrier certificate generation is equivalent to that of solving a bilinear matrix inequality (BMI) with a particular type. The paper reveals the special feature of the problem, and adopts it to build a novel computational method. The proposed method introduces a sequential iterative scheme that is able to find analytical solutions, rather than the nonlinear solving procedure to produce numerical solutions used by general BMI solvers and thus is more efficient than them. In addition, different from popular LMI solving based methods, it does not make the verification conditions more conservative, and thus reduces the risk of missing feasible solutions. Benefitting from these two appealing features, it can produce barrier certificates not amenable to existing methods, which is supported by a complexity analysis as well as the experiment on some benchmarks.

Keywords: Formal verification · Hybrid systems · Barrier certificates · Bilinear matrix inequalities

This work was supported by the National Key Research and Development Program of China under Grant No. 2017YFA0700604, the National Natural Science Foundation of China under Grant 61772203, 61751210, 61632015, Scientific and Technological Innovation 2030 Major Projects under Grant 2018AAA0100902, the Shanghai Natural Science Foundation, China under Grant 17ZR1408300, Zhejiang Provincial Natural Science Foundation of China under Grant LY20F020020.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 582–603, 2020.
https://doi.org/10.1007/978-3-030-53288-8_29

1 Introduction

Cyber-physical systems (CPS) consists of tightly coupled physical components such as electrical, mechanical, hydraulic, and biological components and software systems. They are deeply involved in many safety-critical systems, for example, high confidence medical devices, traffic control and safety systems, advanced automotive systems and critical infrastructure control systems. Safety verification helps to ensure them not to behave dangerously.

Hybrid systems are popular models used in the verification of Cyber-physical systems, for its ability to describe interacting discrete transitions and continuous dynamics [18]. Safety verification contributes to checking safety properties by determining whether a system can evolve to some states violating desired safety properties when it starts at some initial conditions. A successful verification of a hybrid system can raise our confidence in its corresponding Cyber-physical system.

For Cyber-physical systems with real time constraints, fast verification is a vital requirement. For example, a online verification module in a monitoring system should return the result before the deadline is reached. The paper aims at fast verification of hybrid systems to satisfy the requirement of fast verification of Cyber-physical systems.

Intuitively, safety verification of hybrid systems can be performed by computing the reachable set. Reachable set computation based approaches explicitly computes either exact or approximate reachable sets corresponding to the dynamics in the model, and then compares them with unsafe regions. It has been successfully adopted in verifying behaviors of a system within a finite horizon. However, due to their intrinsic computational difficulty, approaches of this kind can hardly scale up to complex non-linear systems.

Many research efforts have been devoted to barrier certificate generation. A barrier certificate is a function, of which the zero level set separates the unsafe region from all reachable states of a system. It requires all system trajectories starting from some initial conditions fall into one side of the barrier certificate while the unsafe region resides on the other. As the existence of a barrier certificate implies that the unsafe region is not reachable, the safety verification problem can be transformed into the problem of barrier certificate generation. Compared with reachable set computation [31], barrier certificate generation requires much less computation, since the unsafe region leads to seeking a barrier certificate. Especially, it behaves very well when a safety property concerns infinite time horizon [21,34].

Barrier certificate generation is a computation intensive task. A set of verification conditions corresponding to a specific type of barrier certificates is given at first. Then they are encoded into some constraints on state variables and unknown coefficients of barrier certificates of a specific type. Finally, those unknown coefficients are determined by solving the constraints [27]. Thus, how to encode verification conditions and solve them in an effective way is a critical and challenging problem in barrier certificate based verification.

Acting as the barrier between reachable states and the unsafe region, a barrier certificate should always evaluate to be nonnegative or negative accordingly in spite of what type it is. To achieve this, the most popular computational method utilizes the theory of Putinar's Positivstellensatz to derive a *sum of squares* (SOS) program of the barrier certificate, which results in a *bilinear matrix inequality* (BMI) solving problem belonging to the class of NP-hard problems [20,21]. An effective and efficient BMI solver is a prerequisite for success in exploiting SOS relaxation based methods.

The general BMI problem can be solved by the commercial BMI solver PENBMI [14] at the cost of a very high computational complexity, where the (exterior) penalty and (interior) barrier method incorporates with the augmented Lagrangian method. To make it more tractable, the convex SOS relaxation based methods become popular. They transform the BMI problem (non-convex) to a *linear matrix inequality* (LMI) problem (convex) by fixing some multipliers and then solve it quickly via convex optimization such as semidefinite programming (SDP). Unfortunately, the removal of non-convexity may yield too conservative verification conditions so that the solution to the original BMI problem is invisible to the derived LMI problem.

The paper focuses on quickly solving the BMI problem derived from SOS relaxation by directly attacking the problem without relaxing it to a LMI one. Taking advantage of the special feature of the problem, that is all bilinear terms are cross ones between different parameter vectors, a sequential iterative scheme is proposed. It treats the non-convex BMI problem directly so as to avoid the loss of precision accompanied with non-convexity removing. Meanwhile, it provides much lower computational complexity than the PENBMI solver. Hence, the proposed method spends much less time in computation and has the potential to find solutions beyond the reach of existing methods.

To be specific, a feasible solution to the BMI problem can be found by a dual augmented Lagrangian iterative framework. At each iteration, the minimization over the four sets of primal variables is divided into four sequential minimization problems with respect to one set of primal variables by fixing the other three sets. On the theoretical side, we show that our method returns the feasible solution in cubic time, while the PENBMI solver in quartic time. We have developed a prototyping tool implementing the proposed method and compared it with the PENBMI solver and the LMI solver: SOSTOOLS [22] over a set of benchmarks gathered from the literature. The experiment shows that our tool is more effective than them and provides a much lower computational complexity than the PENBMI solver.

The paper is organized as follows. Section 2 describes the connection between safety verification and barrier certificate generation. Section 3 addresses how to transform the problem of barrier certificate generation into a BMI solving problem. In Sect. 4, a sequential iterative scheme is presented followed by a complexity analysis. Section 5 contains detailed examples illustrating the use of our method as well as the experiment on benchmarks. We compare with related works in Sect. 6 before concluding in Sect. 7.

2 Preliminaries

Notations. Let $\mathbb{R}$ be the field of real number. $\mathbb{R}[\mathbf{x}]$ denotes the polynomial ring with coefficients in $\mathbb{R}$ over variables $\mathbf{x} = [x_1, x_2, \cdots, x_n]^T$. Let $\Sigma[\mathbf{x}] \subset \mathbb{R}[\mathbf{x}]$ be the space of SOS polynomials. S^n denotes the set of $n \times n$ symmetric matrices, and the notation $B \succeq 0$ means that the matrix $B \in S^n$ is positive semidefinite. $\langle A, B \rangle$ denotes the inner product between A and B.

A continuous dynamical system is modeled by a finite number of first-order ordinary differential equations

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}), \tag{1}$$

where $\dot{\mathbf{x}}$ denotes the derivative of $\mathbf{x}$ with respect to the time variable t, and $\mathbf{f}(\mathbf{x})$ is called vector field $\mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}), \cdots, f_n(\mathbf{x})]^T$ defined on an open set $\Psi \subseteq \mathbb{R}^n$. We assume that $\mathbf{f}$ satisfies the local Lipschitz condition, which ensures that given $\mathbf{x} = \mathbf{x}_0$, there exists a time $T > 0$ and a unique function $\tau : [0, T) \mapsto \mathbb{R}^n$ such that $\tau(0) = \mathbf{x}_0$. And $\mathbf{x}(t)$ is called a solution of (1) that starts at a certain initial state $\mathbf{x}_0$, that is, $\mathbf{x}(0) = \mathbf{x}_0$. Namely, $\mathbf{x}(t)$ is also called a trajectory of (1) from $\mathbf{x}_0$.

Definition 1 (Continuous System). *A continuous system over* $\mathbf{x}$ *consists of a tuple* $\mathbf{S} : \langle \Theta, \mathbf{f}, \Psi \rangle$*, wherein* $\Theta \subseteq \mathbb{R}^n$ *is a set of initial states,* $\mathbf{f}$ *is a vector field over the domain* $\Psi \subseteq \mathbb{R}^n$*.*

A hybrid system is a system which exhibits mixed discrete-continuous behaviors. A popular model for representing hybrid systems is hybrid automata [1], which combine finite state automata modeling the discrete dynamics, and differential equations modeling the continuous dynamics.

Definition 2 (Hybrid Automata). *A* hybrid automaton *is a tuple* $\mathbf{H} : \langle L, X, F, \Psi, E, \Xi, \Delta, \Theta, \ell_0 \rangle$*, where*

- L*, a finite set of locations (or models);*
- $X \subseteq \mathbb{R}^n$ *is the continuous state space. The hybrid state space of the system is defined by* $\mathcal{X} = L \times X$ *and a state is defined by* $(\ell, \mathbf{x}) \in \mathcal{X}$*;*
- $F : L \to (\mathbb{R}^n \to \mathbb{R}^n)$*, assigns to each location* $\ell \in L$ *a locally Lipschitz continuous vector field* $\mathbf{f}_\ell$*;*
- Ψ *assigns to each location* $\ell \in L$ *a* location condition (location invariant) $\Psi(\ell) \subseteq \mathbb{R}^n$*;*
- $E \subseteq L \times L$ *is a finite set of discrete transitions;*
- Ξ *assigns to each transition* $e \in E$ *a switching guard* $\Xi_e \subseteq \mathbb{R}^n$*;*
- Δ *assigns to each transition* $e \in E$ *a reset function* $\Delta_e : \mathbb{R}^n \to \mathbb{R}^n$*;*
- $\Theta \subseteq \mathbb{R}^n$*, an initial continuous state set;*
- $\ell_0 \in L$*, the initial location. The initial state space of the system is defined by* $\ell_0 \times \Theta$*.*

Trajectories of hybrid systems combine continuous flows and discrete transitions. Concretely, a trajectory of $\mathbf{H}$ is an infinite sequence of states $\sigma = \{s_0, s_1, s_2, \cdots\}$ such that

- **[Initiation]** $s_0 = (\ell_0, \mathbf{x}_0)$, with $\mathbf{x}_0 \in \Theta$;
  Furthermore, for each pair of consecutive state $(s_i, s_{i+1}) \in \sigma$ with $s_i = (\ell_i, \mathbf{x}_i)$ and $s_{i+1} = (\ell_{i+1}, \mathbf{x}_{i+1})$ satisfies the following one of the two *consecution* conditions:
- **[Discrete Consecution]** $e = (\ell_i, \ell_{i+1}) \in E$, $\mathbf{x}_i \in \Xi_e$ and $x_{i+1} = \Delta_e(\mathbf{x}_i)$;
- **[Continuous Consecution]** $\ell_i = \ell_{i+1}$, and there exists a time interval $\delta > 0$ such that the solution $\mathbf{x}(\mathbf{x}_i; t)$ to $\dot{\mathbf{x}} = \mathbf{f}_{\ell_i}$ evolves from $\mathbf{x}_i$ to $\mathbf{x}_{i+1}$, while satisfying the location invariant $\Psi(\ell_i)$. Formally, $\mathbf{x}(\mathbf{x}_i, \delta) = \mathbf{x}_{i+1}$ and $\forall t \in [0, \delta], \mathbf{x}(\mathbf{x}_i, t) \in \Psi(\ell_i)$.

If Σ is the set of all possible trajectories of $\mathbf{H}$, the reachable set is defined by $R = \{s | \exists \varsigma \in \Sigma : s \in \varsigma\}$, i.e., R contains all states that are elements of at least one trajectory ς.

In this paper, we focus on semi-algebraic hybrid systems, that is, the corresponding vector fields are polynomials and the sets $\Theta, \Psi(\ell), \Xi_e, \Delta_e$ in $\mathbf{H}$ are semi-algebraic, represented by polynomial equations and inequalities. The semi-algebraic sets Θ, $\Psi(\ell)$, Ξ_e, and Δ_e in Definition 2 are represented as follows:

$$\begin{cases} \Theta := \{\mathbf{x} \in \mathbb{R}^n \,|\, \theta(\mathbf{x}) \geq 0\}, \\ \Psi(\ell) := \{\mathbf{x} \in \mathbb{R}^n \,|\, \psi_\ell(\mathbf{x}) \geq 0\}, \\ \Xi_e := \{\mathbf{x} \in \mathbb{R}^n \,|\, \rho_e(\mathbf{x}) \geq 0\}, \\ \Delta_e := \{\mathbf{x}' \in \mathbb{R}^n \,|\, \delta_e(\mathbf{x}') \geq 0\}, \end{cases}$$

where $\ell \in L$, $e \in E$, $\theta(\mathbf{x})$, $\psi_\ell(\mathbf{x})$, $\rho_e(\mathbf{x})$, and $\delta_e(\mathbf{x}')$ are vectors of polynomials, and the inequalities are satisfied entry-wise. Suppose that X_u assigns to each location $\ell \in L$ an unsafe region $X_u(\ell)$, defined by

$$X_u(\ell) := \{\mathbf{x} \in \mathbb{R}^n \,|\, \zeta_\ell(\mathbf{x}) \geq 0\},$$

where ζ_ℓ is a vector of polynomials. The safety specification is described over the trace of state $(\ell, \mathbf{x})$ w.r.t. unsafe regions $X_u(\ell)$.

Definition 3 (Safety). *Given a hybrid system* $\mathbf{H} : \langle L, X, F, \Psi, E, \Xi, \Delta, \Theta, \ell_0 \rangle$ *and unsafe regions* $X_u(\ell)$*, the safety property holds if there exist no trajectories of* $\mathbf{H}$ *starting from the initial set* $\ell_0 \times \Theta$*, can evolve to any state specified by* $X_u(\ell)$*, i.e.,* $\forall \ell \in L \, \forall \sigma \in \Sigma. \, s \in \sigma \models s \notin X_u(\ell)$.

For safety verification of hybrid systems, the notion of barrier certificates [21] plays an important role. A barrier certificate maps all the states in the reachable set R to non-negative reals and all the states in the unsafe region to negative reals, thus can be employed to prove safety of hybrid systems. However, the exact reachable set R is usually intractable for most hybrid systems. In [21], a sufficient inductive condition for barrier certificates is defined as follows.

Definition 4 (Barrier Certificate). *A barrier certificate of hybrid system* $\mathbf{H}$ *for safety w.r.t. unsafe regions* $X_u(\ell)$ *is a set of real functions* $\{B_\ell(\mathbf{x})\}$ *such that, for all* $\ell \in L$ *and* $e = (\ell, \ell') \in E$*, the following conditions hold:*

$$\begin{cases} \forall \mathbf{x} \in \Theta : \ B_{\ell_0}(\mathbf{x}) \geq 0, \\ \forall \mathbf{x} \in \Psi(\ell) : \ B_\ell(\mathbf{x}) = 0 \models \left\langle \frac{\partial B_\ell}{\partial \mathbf{x}}(\mathbf{x}), \mathbf{f}_\ell(\mathbf{x}) \right\rangle > 0, \\ \forall \mathbf{x} \in \Xi_e, \forall \mathbf{x}' \in \Delta_e(\mathbf{x}) : \ B_\ell(\mathbf{x}) \geq 0 \models B_{\ell'}(\mathbf{x}') \geq 0, \\ \forall \mathbf{x} \in X_u(\ell) : \ B_\ell(\mathbf{x}) < 0. \end{cases} \tag{2}$$

Note that $\left\langle \frac{\partial B_\ell}{\partial \mathbf{x}}(\mathbf{x}), \mathbf{f}_\ell(\mathbf{x}) \right\rangle$ is the Lie derivative of $B_\ell(\mathbf{x})$ with respect to the vector field $\mathbf{f}_\ell(\mathbf{x})$.

3 Transfer to BMI

The problem of generating barrier certificates in Definition 4 is an infinite-dimensional problem. In order to make it amenable to polynomial optimization, the barrier certificate $\{B_\ell(\mathbf{x})\}$ should be restricted to a set of polynomials with a priori degree bound. Putinar's Positivstellensatz provides a powerful representation for polynomial positivity on semi-algebraic sets, which helps to transform the problem of barrier certificate generation into solving a semidefinite programming via SOS relaxation.

Arising from the second and third conditions of Definition 4, where the parameters of $\{B_\ell(\mathbf{x})\}$ appear on the antecedent sides, the associated SOS representations using Putinar's Positvstellensatz form non-convex BMI constraints, yielded from the polynomial products between the barrier certificate and its polynomial multipliers.

In what follows, the procedure for transforming barrier certificate generation into BMI solving is recapped in detail. Firstly, SOS relaxation is applied to encode the entailment checking in condition (2) as an SOS program. In fact, all the conditions of Definition 4 can be expressed as a unified type, say, a polynomial is nonnegative (positive) on a semi-algebraic set, which can be characterized by Putinar's Positivstellensatz.

Let $\mathbb{K}$ be a basic semi-algebraic set defined by:

$$\mathbb{K} = \{\mathbf{x} \in \mathbb{R}^n \mid g_1(\mathbf{x}) \geq 0, \ldots, g_s(\mathbf{x}) \geq 0\}, \tag{3}$$

where $g_j \in \mathbb{R}[\mathbf{x}], 1 \leq j \leq s$. Given the finite family $\mathbf{g} = \{g_1(\mathbf{x}), \ldots, g_s(\mathbf{x})\}$,the polynomial set defined by

$$M(\mathbf{g}) =:= \{\sigma_0 + \sum_{i=1}^{s} \sigma_i g_i \mid \sigma_i \in \Sigma[\mathbf{x}], \ 0 \leq i \leq s\}$$

is called the quadratic module generated by $\mathbf{g}$.

Theorem 1. *[Putinar's Positivstellensatz] Let $\mathbb{K} \subset \mathbb{R}[\mathbf{x}]$ be as in (3). Assume that the quadratic module $M(\mathbf{g})$ is archimedean, namely, there exists $u(\mathbf{x}) \in M(\mathbf{g})$ such that the set $\{\mathbf{x} \in \mathbb{R}^n | u(\mathbf{x}) \geq 0\}$ is compact. If $f(\mathbf{x})$ is strictly positive on $\mathbb{K}$, then $f(\mathbf{x})$ can be represented as*

$$f(\mathbf{x}) = \sigma_0(\mathbf{x}) + \sum_{i=1}^{s} \sigma_i(\mathbf{x}) g_i(\mathbf{x}), \tag{4}$$

where $\sigma_i \in \Sigma[\mathbf{x}], 0 \leq i \leq s$.

Following Theorem 1, the existence of the representation (4) provides a sufficient and necessary condition of polynomial positivity on a semi-algebraic set $\mathbb{K}$ [23]. Although the number of auxiliary polynomials in the representation (4) is only one more than the number of polynomials that define $\mathbb{K}$, the degree bound for $\sigma_i(\mathbf{x})$ is exponential with n and $\deg(\mathbf{f})$. From a computational point of view, the method for finding the above representation has some degree of conservativeness, say, by fixing a priori much smaller degree bound D for $\sigma_i(\mathbf{x})$. Thus, a sufficient condition for the nonnegativity of the given polynomial $f(\mathbf{x})$ on the semi-algebraic set $\mathbb{K}$ is provided as

$$f(\mathbf{x}) = \sigma_0(\mathbf{x}) + \sum_{i=1}^{s} \sigma_i(\mathbf{x}) g_i, \tag{5}$$

with $\deg(\sigma_i) \leq D$, $\sigma_i \in \Sigma[\mathbf{x}], 1 \leq i \leq s$. The representation (5) ensures that a polynomial is nonnegative on a given semi-algebraic set. At this point, all conditions in Definition 4 can be derived as a unified type, i.e., polynomial nonnegativity on a semi-algebraic set. The representation (5) is used to characterize the conditions of barrier certificate generation, for they are more tractable.

Theorem 2. *Let the semi-algebraic hybrid system* $\mathbf{H}$ *and the unsafe regions* $X_u(\ell)$ *be defined as the above. Let* D *be a positive integer. Suppose there exist polynomials* $\{B_\ell(\mathbf{x})\}$ *and* $\{\nu_\ell(\mathbf{x})\}$ *with* $\deg(\nu_\ell) \leq D$*, positive numbers* $\epsilon_{\ell,1}$ *and* $\epsilon_{\ell,2}$*, and vectors of sums of squares* $\sigma(\mathbf{x})$*,* $\lambda_{e,i}(\mathbf{x})$*,* $\gamma_e(\mathbf{x})$*,* $\eta_e(\mathbf{x})$*,* $\phi_\ell(\mathbf{x})$*,* $\mu_\ell(\mathbf{x})$ *with the degree bound* D*, such that the following expressions:*

$$\begin{array}{l} B_{\ell_0}(\mathbf{x}) - \sigma(\mathbf{x})\theta(\mathbf{x}) \\ B_{\ell'}(\mathbf{x}') - \lambda_e(\mathbf{x})\rho_e(\mathbf{x}) - \gamma_e(\mathbf{x}')\delta_e(\mathbf{x}') - \eta_e(\mathbf{x})B_\ell(\mathbf{x}) \\ \left\langle \frac{\partial B_\ell}{\partial \mathbf{x}}(\mathbf{x}), \mathbf{f}_\ell(\mathbf{x}) \right\rangle - \phi_\ell(\mathbf{x})\psi_\ell(\mathbf{x}) - \nu_\ell(\mathbf{x})B_\ell(\mathbf{x}) - \epsilon_{\ell,1} \\ -B_\ell(\mathbf{x}) - \mu_\ell(\mathbf{x})\zeta_\ell(\mathbf{x}) - \epsilon_{\ell,2} \end{array} \tag{6}$$

are SOSes for each $\ell \in L$ *and* $e \in E$*. Then* $\{B_\ell(\mathbf{x})\}$ *satisfies the conditions in Definition 4, and therefore guarantees the safety of* $\mathbf{H}$*.*

Remark that a polynomial $f(\mathbf{x})$ with $\deg(f) = 2d$ is a sum of squares if and only if there exists a real symmetric and positive semidefinite matrix Q, called as the Gram matrix, such that $f(\mathbf{x}) = \mathbf{v}_d(\mathbf{x})^T Q \mathbf{v}_d(\mathbf{x})$, where $\mathbf{v}_d(\mathbf{x})$ is the vector consisting of all the monomials of degree less than or equal to d. In view of the conditions (6) in Theorem 2, the problem of generating the barrier certificates requires introducing the auxiliary (Gram matrices) variables. In fact, the decision variables in the SOS program (6) are the coefficients of all the unknown polynomials in (6), such as $B_\ell(\mathbf{x}), \sigma(\mathbf{x})$, $\lambda_e(\mathbf{x})$ and the associated Gram matrices. The polynomial products, i.e., $B_\ell(\mathbf{x})\eta_e(\mathbf{x})$ and $B_\ell(\mathbf{x})\nu_\ell(\mathbf{x})$, derive some quadratic terms of the products of these unknown coefficients, which occur in the second and third constraints of (6). As a consequence, the problem for generating barrier certificates in Theorem 2 derives a non-convex BMI problem. We now show the transformation by a simple example.

Example 1. Consider the system $\dot{x} = -x$ with location invariant $\Psi = \{x \in \mathbb{R} : x^2 - 1 \leq 0\}$. Suppose the barrier certificate $B(x)$ with $\deg(B) = 1$, we predetermine its template as $B(x) = u_0 + u_1\, x$ with $u_0, u_1 \in \mathbb{R}$ and $u_1 \neq 0$. For simplicity, here we consider the second condition in Definition 4, that is, to find $B(x)$ which satisfies

$$\forall x \in \Psi :\ B(x) = 0 \models \frac{\partial B}{\partial x} \cdot (-x) \geq 0.$$

Following the SOS relaxation in (6), we need to find $B(x)$ such that

$$\phi_0(x) := \frac{\partial B}{\partial x} \cdot (-x) - \phi_1(x) \cdot (1 - x^2) - \phi_2(x) \cdot B(x) - \epsilon \tag{7}$$

and $\phi_1(x)$ are SOSes, $\phi_2(x) \in \mathbb{R}[x]$, $\epsilon \in \mathbb{R}_{>0}$. We assume that $\phi_1 = u_2$ and $\phi_2 = v$, with $u_2 \in \mathbb{R}_{\geq 0}$ and $v \in \mathbb{R}$. Then (7) yields $\phi_0(x) = u_2 x^2 - (u_1 v + u_1)x - u_0 v - u_2 - \epsilon$, and its Gram matrix representation $\phi_0(x) = \mathbf{v}_1(x)^T\, Q\, \mathbf{v}_1(x)$, where

$$Q = \begin{bmatrix} u_2 & -\frac{1}{2}u_1\, v - \frac{1}{2}u_1 \\ -\frac{1}{2}u_1\, v - \frac{1}{2}u_1 & -u_0\, v - u_2 - \epsilon \end{bmatrix} \text{ and } \mathbf{v}_1(x) = \begin{bmatrix} x \\ 1 \end{bmatrix}.$$

Since $\phi_0(x)$ and $\phi_1(x)$ must be SOSes, we have $Q \succeq 0$ and $u_2 \geq 0$, which is equivalent to

$$\mathcal{B}(u_0, u_1, u_2, v) = \begin{bmatrix} u_2 & 0 & 0 \\ 0 & u_2 & -\frac{1}{2}u_1\, v - \frac{1}{2}u_1 \\ 0 & -\frac{1}{2}u_1\, v - \frac{1}{2}u_1 & -u_0\, v - u_2 - \epsilon \end{bmatrix} \succeq 0.$$

Therefore, the requirement that $\phi_0(x)$ and $\phi_1(x)$ are SOSes is translated into the BMI constraint of the form

$$\mathcal{B} = B_{0,0} + \sum_{i=0}^{2} u_i B_{i,0} + v B_{0,1} + \sum_{i=0}^{2} u_i v\, B_{i,1} \succeq 0, \tag{8}$$

where all $B_{i,j} \in S^3$ are constant matrices. □

As illustrated in Example 1, the problem of generating barrier certificates satisfying condition (6) can be transformed into a BMI problem of the form

$$\begin{aligned} &\text{Find } \mathbf{u} \in \mathbb{R}^p,\ \mathbf{v} \in \mathbb{R}^q \\ &\text{s.t. } \mathcal{B}(\mathbf{u}, \mathbf{v}) = B_{0,0} + \sum_{i=1}^{p} u_i B_{i,0} + \sum_{j=1}^{q} v_j B_{0,j} + \sum_{i=1}^{p}\sum_{j=1}^{q} u_i v_j B_{ij} \succeq 0, \end{aligned} \tag{9}$$

where all $B_{i,j} \in S^t$ are constant matrices, $\mathbf{u} = [u_1, \ldots, u_p]^T$, $\mathbf{v} = [v_1, \ldots, v_q]^T$ are parameter coefficients of the unknown polynomials occurring in the original SOS program. Essentially, the BMI problem (9) is NP-hard. To simplify the problem considerably, the canonical approach is to swap $\mathbf{v}$, corresponding to the polynomial multipliers $\eta_e(\mathbf{x})$ and $\nu_\ell(\mathbf{x})$, with the fixed vector. This strategy

can reduce the BMI constraint into the associated LMI one. Unfortunately, the resulting LMI problem is considerably more conservative than the original BMI one. To be specific, the fixed $\eta_e(\mathbf{x})$ and $\nu_\ell(\mathbf{x})$ may result in too conservative verification conditions that rule out barrier certificates satisfy the non-convex conditions but not the stronger convex conditions.

By investigating (9), we can find a crucial feature of $\mathcal{B}(\mathbf{u}, \mathbf{v})$, that is, all cross terms between parameters of $\mathbf{u}$ and $\mathbf{v}$ are of the form $u_i\, v_j$. The feature motivates us to design a more efficient approach for the specific type of BMI problems.

4 A Sequential Iterative Scheme for Solving BMI Problems

The conventional approaches for solving the BMI problem typically employ the augmented Lagrangian iterative framework, wherein each iteration involves two optimization problems for primal and dual variables. Due to the existence of nonlinear terms (quartic terms) in the associated Lagrangian function, the analytical solutions to the first problem do not exist. The iterative-based nonlinear solving procedure is introduced to obtain the numerical solutions which results in a time-consuming computing process.

Observing the BMI problem (9), we can see that all nonlinear terms are the cross terms between $\mathbf{u}$ and $\mathbf{v}$. As a result, the associated dual augmented Lagrangian function is *quartic* for all variables, but is *quadratic* with respect to each single variable. Having this crucial feature, if we choose one variable as the independent variable and assign the others with fixed values, we may get the problem of minimizing the quadratic function. According to the first-order optimality condition, given a quadratic function $f(\mathbf{x})$, the sufficient and necessary condition that $\tilde{\mathbf{x}}$ is a minimizer of $f(\mathbf{x})$ requires that the gradient of $f(\mathbf{x})$ to be zero at $\tilde{\mathbf{x}}$, i.e., $\nabla f(\tilde{\mathbf{x}}) = 0$. As a consequence, the analytical solutions to our studied optimization problem can be easily formulated, since the gradient of the associated Lagrangian function is affine.

The analytical optimal solutions can be obtained by calling simple matrix computation, and thus are much more efficient than numerical solutions whose computation relies on complicated nonlinear optimization methods. The computational advantage is further demonstrated by a complexity analysis of our scheme against the existing BMI solving algorithm that combines the (exterior) penalty and (interior) barrier method with the augmented Lagrangian method, presented later in this section.

To utilize the computational advantage of analytical optimal solutions, for the first optimization problem (w.r.t primal variables) involved in each iteration of the augmented Lagrangian iterative framework, rather than using the usual joint minimization for all primal variables, we introduce a sequential minimization scheme, that is, dividing it into four sequential sub-optimization problems over one independent variable while keeping the others fixed. More concretely, the sub-optimization problem with one single primal variable is constructed by replacing the other variables with their optimal solutions obtained from the current iteration (if available) or the last iteration.

This section first introduces an iterative scheme to solve the BMI problem and then illustrates how to derive analytical solutions to the sub-problems in each iteration followed by a complexity analysis against the existing algorithm.

4.1 An Iterative Scheme

We start by presenting a straightforward reformulation of the BMI problem (9) as follows:

$$\begin{cases} \lambda^* = \min \lambda \\ \quad s.t.\ Z = \lambda \cdot I + \mathcal{B}(\mathbf{u}, \mathbf{v}) \\ \quad\quad Z \succeq 0. \end{cases} \tag{10}$$

Clearly, there exists a feasible solution $(\mathbf{u}, \mathbf{v})$ to the BMI problem (9) if and only if the optimal value of problem (10) is non-positive, i.e., $\lambda^* \leq 0$. We try to build an iterative scheme for dealing with the optimization problem (10).

The augmented Lagrangian function $\mathcal{L}$ associated with (10) is defined as:

$$\mathcal{L}_\mu(\lambda, \mathbf{u}, \mathbf{v}, Z, U) = \lambda + \langle U, Z - \lambda I - \mathcal{B}(\mathbf{u}, \mathbf{v})\rangle + \frac{1}{2\mu}\|Z - \lambda I - \mathcal{B}(\mathbf{u}, \mathbf{v})\|_F^2, \tag{11}$$

where $\mu > 0$, $\langle \cdot, \cdot \rangle$ means the inner product operator, and $\| \cdot \|_F$ denotes the Frobenius norm of a matrix. Let $U \in S^t$ be the Lagrangian multiplier associated with the equality constraint, the dual function is defined as

$$g(U) = \inf_{(\lambda, \mathbf{u}, \mathbf{v}, Z)} \mathcal{L}_\mu(\lambda, \mathbf{u}, \mathbf{v}, Z, U),$$

and the *Lagrange dual problem* associated with (10) is to maximize this dual function $g(U)$, i.e., $\max\limits_{U} g(U)$. Clearly, the dual function yields lower bounds on the optimal value λ^* of the problem (10), that is, $g(U) \leq \lambda^*$ for any U.

Applying the dual ascent [17] to the augment Lagrangian function yields the iterative scheme, consisting of the following updates

$$\left.\begin{aligned} (\lambda^{k+1}, \mathbf{u}^{k+1}, \mathbf{v}^{k+1}, Z^{k+1}) &:= \operatorname*{argmin}_{\lambda, \mathbf{u}, \mathbf{v}, Z} \mathcal{L}_\mu(\lambda, \mathbf{u}, \mathbf{v}, Z, U^k), \\ &\qquad s.t. \quad Z \succeq 0, \\ U^{k+1} &:= \operatorname*{argmax}_{U} \mathcal{L}_\mu(\lambda^{k+1}, \mathbf{u}^{k+1}, \mathbf{v}^{k+1}, Z^{k+1}, U), \end{aligned}\right\} \tag{12}$$

where the first step is the primal variables update, and the second step is the dual variable update.

The first step in (12) consists of quartic terms and is lack of analytical solution. Thus, it requires jointly minimizing $\mathcal{L}_\mu(\lambda, \mathbf{u}, \mathbf{v}, Z, U^k)$ with respect to $\lambda, \mathbf{u}, \mathbf{v}$ and Z, which can be directly solved by applying the iterative-based nonlinear optimization procedure at the cost of a high computational complexity. Instead of the usual joint minimization solving, we separate the minimization over the

primal variables $\lambda, \mathbf{u}, \mathbf{v}, Z$ into four steps, that is, $\lambda, \mathbf{u}, \mathbf{v}$ and Z are updated in an alternating scheme, that is, minimizing $\mathcal{L}_\mu$ with respect to one primal variable given the others fixed. In detail, the sequential iterative scheme consists of the following new iterations:

$$\lambda^{k+1} := \underset{\lambda}{\operatorname{argmin}} \mathcal{L}_\mu(\lambda, \mathbf{u}^k, \mathbf{v}^k, Z^k, U^k), \tag{13}$$

$$\mathbf{u}^{k+1} := \underset{\mathbf{u}}{\operatorname{argmin}} \mathcal{L}_\mu(\lambda^{k+1}, \mathbf{u}, \mathbf{v}^k, Z^k, U^k), \tag{14}$$

$$\mathbf{v}^{k+1} := \underset{\mathbf{v}}{\operatorname{argmin}} \mathcal{L}_\mu(\lambda^{k+1}, \mathbf{u}^{k+1}, \mathbf{v}, Z^k, U^k), \tag{15}$$

$$Z^{k+1} := \underset{Z \succeq 0}{\operatorname{argmin}} \mathcal{L}_\mu(\lambda^{k+1}, \mathbf{u}^{k+1}, \mathbf{v}^{k+1}, Z, U^k), \tag{16}$$

$$U^{k+1} := \underset{U}{\operatorname{argmax}} \mathcal{L}_\mu(\lambda^{k+1}, \mathbf{u}^{k+1}, \mathbf{v}^{k+1}, Z^{k+1}, U). \tag{17}$$

The above iterative scheme introduces a sequential minimization that treats the four primal variables one by one. Benefited from the fact that the explicit formulae for the minimizer or maximizer (13–17) are available, the analytical solutions can be directly derived. Furthermore, as the computation of those analytical solutions involves only simple matrix computation, such as eigenvalue decomposition and matrix inverse, it will be very efficient.

4.2 Analytical Solutions for the Sequential Iteration

In this subsection, we focus on how to find analytical solutions to problems (13–17) in terms of the first-order optimality conditions.

Theorem 3. *The minimizer λ^{k+1} of (13),i.e.,*

$$\lambda^{k+1} := \underset{\lambda}{\operatorname{argmin}} \mathcal{L}_\mu(\lambda, \mathbf{u}^k, \mathbf{v}^k, Z^k, U^k),$$

has the following analytical formula:

$$\lambda^{k+1} := \frac{1}{t} \sum_{i=1}^{t} (Z_{i,i}^k - \mathcal{B}_{i,i}(\mathbf{u}^k, \mathbf{v}^k)) + \frac{\mu}{t} \cdot (\mathrm{Tr}(\mathrm{U^k}) - 1), \tag{18}$$

where $\mathrm{Tr}(\mathrm{U^k})$ *denotes the trace of* U^k.

Proof. The first-order optimality condition for (13) is

$$\nabla_\lambda \mathcal{L}_\mu = 1 - \mathrm{Tr}(U^k) + \frac{t}{\mu}\lambda - \frac{1}{\mu} \sum_{i=1}^{t} (Z_{i,i}^k - \mathcal{B}_{i,i}(\mathbf{u}^k, \mathbf{v}^k)) = 0.$$

It follows that the specified λ^{k+1} in (18) is the optimal solution of (13), which concludes the proof. □

The first-order optimality condition resembling Theorem 3 can also be invoked to produce the corresponding analytical solutions to (14) and (15), respectively.

Theorem 4. *Let* $\mathbf{v}^k = [v_1^k, \ldots, v_q^k]^T \in \mathbb{R}^q$, *and define* $X^{[i]} = B_{i,0} + \sum_{\ell=1}^q v_\ell^k B_{i,\ell}$ *for* $0 \le i \le p$. *Let* $\mathbf{u}^{k+1}$ *be the minimizer of (14). Then*

$$\mathbf{u}^{k+1} := S^{-1} \cdot [r_1, \ldots, r_p]^T, \tag{19}$$

where $S = [s_{ij}] \in \mathbb{R}^{p\times p}$ *with* $s_{ij} = \frac{1}{\mu}\langle X^{[i]}, X^{[j]}\rangle$, *and*

$$r_i = \langle U^k + \frac{1}{\mu}(Z^k - \lambda^{k+1} I - X^{[0]}), X^{[i]}\rangle, 1 \le i \le p.$$

Proof. The first-order optimality condition for (14) is

$$\nabla_{\mathbf{u}} \mathcal{L}_\mu(\lambda^{k+1}, \mathbf{u}, \mathbf{v}^k, Z^k, U^k) = (\nabla_{\mathbf{u}_1} \mathcal{L}_\mu, \nabla_{\mathbf{u}_2} \mathcal{L}_\mu, \cdots, \nabla_{\mathbf{u}_p} \mathcal{L}_\mu)^T = 0,$$

and the i-th gradient function $\nabla_{\mathbf{u}_i} \mathcal{L}_\mu(\lambda^{k+1}, \mathbf{u}, \mathbf{v}^k, Z^k, U^k)$, $1 \le i \le p$ is

$$\langle U^k, -\sum_{\ell=1}^q \mathbf{v}_\ell^k B_{i,\ell} - B_{i,0}\rangle + \frac{1}{\mu}\langle Z^k - \lambda^{k+1} I - \mathcal{B}(\mathbf{u}, \mathbf{v}^k), -\sum_{\ell=1}^q \mathbf{v}_\ell^k B_{i,\ell} - B_{i,0}\rangle.$$

Then we have

$$\nabla_{\mathbf{u}_i} \mathcal{L}_\mu(\lambda^{k+1}, \mathbf{u}, \mathbf{v}^k, Z^k, U^K) = \langle U^k, -X^{[i]}\rangle + \frac{1}{\mu}\langle Z^k - \lambda^{k+1} I - \mathcal{B}(\mathbf{u}, \mathbf{v}^k), -X^{[i]}\rangle$$

for $i = 1 \ldots, p$.

Thus, $\nabla_{\mathbf{u}} \mathcal{L}_\mu(\lambda^{k+1}, \mathbf{u}, \mathbf{v}^k, Z^k, U^k) = 0$ yields (19), which proves the claim. □

Theorem 5. *Let* $\mathbf{u}^{k+1} = [u_1^{k+1}, \ldots, u_p^{k+1}]^T \in \mathbb{R}^p$, *and define* $Y^{[j]} = B_{0,j} + \sum_{\ell=1}^p u_\ell^{k+1} B_{\ell,j}$, *for* $0 \le j \le q$. *Let* $\mathbf{v}^{k+1}$ *be the minimizer of (15). Then*

$$\mathbf{v}^{k+1} := T^{-1} \cdot [w_1, \ldots, w_q]^T, \tag{20}$$

where $T = [t_{ij}] \in \mathbb{R}^{q\times q}$ *with* $t_{ij} = \frac{1}{\mu}\langle Y^{[i]}, Y^{[j]}\rangle$, *and*

$$w_i = \langle U^k + \frac{1}{\mu}(Z^k - \lambda^{k+1} I - Y^{[0]}), Y^{[i]}\rangle, \quad 1 \le i \le q.$$

Proof. Similar to the proof of Theorem 4. □

The theorems below demonstrate the analytical solutions to the Z-minimization and U-maximization, respectively.

Theorem 6. *Let* Z^{k+1} *be the minimizer of (16), and* U^{k+1} *be the solution of (17). Denote by* P^{k+1} *the matrix* $P^{k+1} := \lambda^{k+1} I + \mathcal{B}(\mathbf{u}^{k+1}, \mathbf{v}^{k+1}) - \mu U^k$. *Suppose* $P^{k+1} = Q\Sigma Q^T$ *is a spectral decomposition, namely,*

$$P^{k+1} = Q\Sigma Q^T = \begin{bmatrix} Q_\dagger & Q_\ddagger \end{bmatrix} \begin{bmatrix} \Sigma_+ & 0 \\ 0 & \Sigma_- \end{bmatrix} \begin{bmatrix} Q_\dagger^T \\ Q_\ddagger^T \end{bmatrix},$$

where Σ_+ and $Q_\dagger$ are the nonnegative eigenvalues and the associated orthogonal eigenvectors, while Σ_- and $Q_\ddagger$ are the negative eigenvalues and the associated orthogonal eigenvectors. Then we have

$$Z^{k+1} := Q_\dagger \Sigma_+ Q_\dagger^T, \tag{21}$$

$$U^{k+1} := -\tfrac{1}{\mu} Q_\ddagger \Sigma_- Q_\ddagger^T. \tag{22}$$

Proof. The first-order optimality condition for (16) is

$$\nabla_Z \mathcal{L}_\mu(\lambda^{k+1}, \mathbf{u}^{k+1}, \mathbf{v}^{k+1}, Z, U^k) = 0. \tag{23}$$

In view of the terms of (23), the problem (16) is translated to

$$Z^{k+1} = \underset{Z \succeq 0}{\operatorname{argmin}} \|Z - \lambda^{k+1} I - \mathcal{B}(\mathbf{u}^{k+1}, \mathbf{v}^{k+1}) + \mu U^k\|_F^2, \tag{24}$$

which reads as

$$Z^{k+1} = \underset{Z \succeq 0}{\operatorname{argmin}} \|Z - P^{k+1}\|_F^2.$$

According to the spectral decomposition of P^{k+1}, the result (21) immediately follows.

From (17), we have

$$\begin{aligned} U^{k+1} &= U^k + \frac{1}{\mu}(Z^{k+1} - \lambda^{k+1} I - \mathcal{B}(\mathbf{u}^{k+1}, \mathbf{v}^{k+1})) \\ &= \frac{1}{\mu}(Z^{k+1} - P^{k+1}), \end{aligned}$$

which yields the result (22). □

4.3 Algorithm and Complexity Analysis

From the above observation in Sect. 4.1 and Sect. 4.2, the detailed procedure for the sequential iterative scheme is summarized in Algorithm 1.

Remark 1. At the beginning of Algorithm 1, $\mathbf{u}^0 \in \mathbb{R}^p$, $\mathbf{v}^0 \in \mathbb{R}^q$ are selected randomly, $Z^0 = M_0^\top \cdot M_0$ where $M_0 \in \mathbb{R}^t$ is chosen randomly, and heuristically $U^0 = \delta \cdot I_t$ with $\delta > 0$.

Remark 2. There are several options for the stopping criterion of the loop in Algorithm 1. That is, Algorithm 1 will stop and return the current result when one of the following cases occurs:

- $|\lambda^{k+1} - \lambda^k| \leq \epsilon$,
- $\|Z^{k+1} - Z^k\| \leq \epsilon$,

where ϵ is a given tolerance. A reasonable value for the stopping criterion might be $\epsilon = 10^{-6}$.

Algorithm 1: Sequential Iterative Scheme for solving a BMI (SISBMI)

Input: Problem (9); initial values $\mathbf{u}^0$, $\mathbf{v}^0$, Z^0 and U^0.
Output: A feasible solution $(\mathbf{u}^*, \mathbf{v}^*)$ of (9).

while *stopping criterion not met* **do**
  Compute λ^{k+1} according (18);
  Compute $\mathbf{u}^{k+1}$ and $\mathbf{v}^{k+1}$ according to (19) and (20), respectively;
  $\mathcal{B}^{k+1} \leftarrow \mathcal{B}(\mathbf{u}^{k+1}, \mathbf{v}^{k+1})$;
  Get the minimal eigenvalue of $\mathcal{B}^{k+1}$, denoted by $\hat{\lambda}$;
  if $\hat{\lambda} \geq 0$ **then**
    $(\mathbf{u}^*, \mathbf{v}^*) \leftarrow (\mathbf{u}^{k+1}, \mathbf{v}^{k+1})$;
    return $(\mathbf{u}^*, \mathbf{v}^*)$;
  Compute Z^{k+1} according to (21);
  Compute U^{k+1} according to (22).

Complexity Analysis

We analyze the complexity of Algorithm 1 and further compare it with the algorithm in PENBMI solver [14], which combines the (exterior) penalty and (interior) barrier method with the augmented Lagrangian method. The BMI problem we study corresponds to a nonconvex optimization problem with quartic terms. For the BMI problems of the special form, neither of the two algorithms can guarantee to converge. A complete complexity analysis is not available as the number of iterations is not predictable. Therefore, the computational complexity of one iteration becomes a safe baseline for performance evaluation. In this paper, we follow the same complexity analysis as that in [14], i.e. analyzing the complexity in one iteration.

Recall that the dimension of the matrix $\mathcal{B}(\mathbf{u}, \mathbf{v})$ in (9) is t, and the numbers of variables $\mathbf{u}$ and $\mathbf{v}$ are p and q, respectively. We see that each iteration in Algorithm 1 can be divided into five steps. Firstly, the step of updating λ costs $O(t)$ flops, which is carried out by $3t + 3$ adds. In the step of $\mathbf{u}-$update, the complexity is clearly dominated by the computation of the inverse of $A_{\mathbf{u}} \in \mathbb{R}^{p \times p}$, which costs $O(p^3)$ flops [5]. Analogously, $\mathbf{v}-$update can be done in $O(q^3)$ flops. In the step of $Z-$update, the critical issue is to compute the eigenvalue decomposition of matrix $V^{k+1} \in \mathbb{R}^{t \times t}$, at a cost of about $\frac{4}{3}t^3$ flops. So the step of $Z-$update requires $O(t^3)$ flops. Finally, the step of $U-$update requires about $O(t)$ flops by performing U^{k+1}.

Now, the complexity for the above steps in each iteration of Algorithm 1 is summarized as follows:

- Calculation of $\lambda \rightarrow O(t)$;
- Calculation of $\mathbf{u} \rightarrow O(p^3)$;
- Calculation of $\mathbf{v} \rightarrow O(q^3)$;
- Calculation of $Z \rightarrow O(t^3)$;
- Calculation of $U \rightarrow O(t)$.

The total cost of each iteration in Algorithm 1 is then $O(p^3+q^3+t^3)$, while the cost of the algorithm adopted in PENBMI is approximately $O((p+q)t^3 + (p+q)^2t^2 + (p+q)^3)$, as shown in [14]. Assume that p, q and t are bounded by $T \in \mathbb{Z}$, i.e., $T = \max\{p, q, t\}$, the complexity of Algorithm 1 is approximately $O(T^3)$, whereas the complexity of PENBMI is approximately $O(T^4)$.

5 Experiments

In this section, we first show our method by verifying a nonlinear continuous system and then compare our Sequential Iterative Scheme tool: SISBMI solver with the other two solvers: PENBMI and SOSTOOLS.

Example 2. Consider the following nonlinear continuous system [28]

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \end{bmatrix} = \begin{bmatrix} 10(x_2 - x_1) \\ x_1(28 - x_3) - x_2 \\ x_1x_2 - \frac{8}{3}x_3 \end{bmatrix}$$

with the location invariant

$$\Psi = \{\mathbf{x} \in \mathbb{R}^3 \mid -20 \leq x_1, x_3 \leq 20, -20 \leq x_2 \leq 0\}.$$

It is required to verify that all trajectories of the system starting from the initial set

$$\Theta = \{\mathbf{x} \in \mathbb{R}^3 \,|\, (x_1 + 14.5)^2 + (x_2 + 14.5)^2 + (x_3 - 12.5)^2 \leq 16\}$$

will never enter the unsafe region

$$X_u = \{\mathbf{x} \in \mathbb{R}^3 \,|\, (x_1 + 16.5)^2 + (x_2 + 14.5)^2 + (x_3 - 2.5)^2 \leq 38.44\}.$$

It suffices to find a barrier certificate $B(\mathbf{x})$, which satisfies all the conditions in Definition 3. Suppose that the degree of $B(\mathbf{x})$ is 4, and the degree bound $D = 6$. Firstly, we construct a bilinear SOS program (6), which is further transformed into a BMI problem of the form (9) where the dimension of $\mathcal{B}(\mathbf{u}, \mathbf{v})$ is 78, and the number of decision variables is 396. By applying our algorithm, we succeed to solve the BMI problem and obtain the following barrier certificate

$$B(\mathbf{x}) = \underbrace{-0.0020x_1^4 - 0.0013x_3^4 - 0.0131x_1^2x_3^2 - 0.0022x_1x_2x_3^2 + \cdots + 0.0938x_1 + 62.5702}_{28 \ \ terms}.$$

As shown in Fig. 1, the zero level set of the barrier certificate $B(\mathbf{x})$ (the steelblue surface) separates X_u (the red ball) from all trajectories starting from Θ (the green ball). Therefore, the safety of the above system is verified.

Alternatively, by applying the PENBMI solver to compute the solution of the problem (9), we cannot find barrier certificates with degree less than 6. □

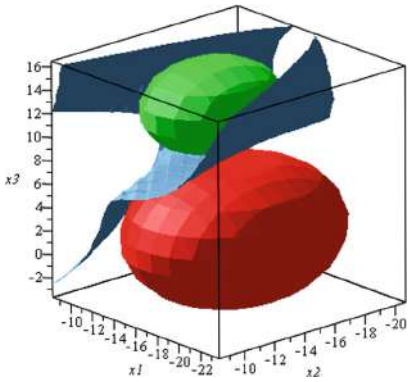


Fig. 1. Phase portrait of the system in Example 2. (Color figure online)

Example 3. Consider the following hybrid system [20] depicted in Fig. 2, where

$$\mathbf{f}_1 = \begin{bmatrix} -x_2 \\ -x_1 + x_3 \\ x_1 + (2x_2 + 3x_3)(1 + x_3^2) \end{bmatrix}, \quad \mathbf{f}_2 = \begin{bmatrix} -x_2 \\ -x_1 + x_3 \\ -x_1 - 2x_2 - 3x_3 \end{bmatrix}.$$

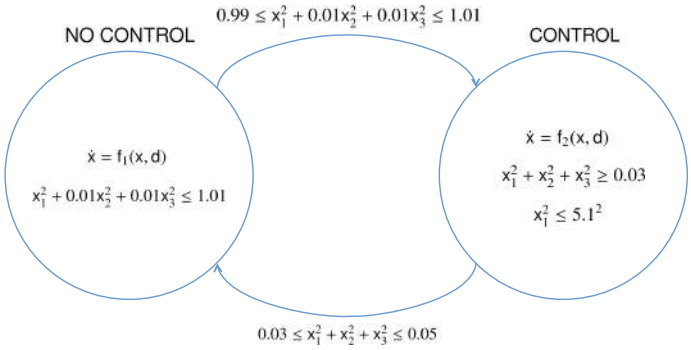


Fig. 2. The hybrid automata of the system in Example 3

The system starts in location ℓ_1 with the initial set

$$\Theta = \{\mathbf{x} \in \mathbb{R}^3 : x_1^2 + x_2^2 + x_3^2 \le 0.01\}.$$

Our task is to verify that the system will never enter the unsafe set

$$X_u(\ell_2) = \{\mathbf{x} \in \mathbb{R}^3 : 5 < x_1 < 5.1\}.$$

Applying our SISBMI solver, we obtain the polynomial barrier certificate with degree 4:

$$B_{\ell_1}(\mathbf{x}) = \underbrace{0.0551x_1^4 + 0.0392x_2^4 + 0.0079x_3^4 + 0.0696x^2x_3^2 + \cdots - 1.1134x_1 + 2.701}_{35\ \ terms},$$

$$B_{\ell_2}(\mathbf{x}) = \underbrace{0.0273{x_1}^4 + 0.0541x_1^3x_2 - 1.098x_1x_2^2 - 0.521x_1x_2x_3 + \cdots - 2.725x_1 + 8.197}_{35\ \ terms}.$$

□

Our SISBMI solver was implemented in Matlab (2018b), and was compared with two solvers PENBMI and SOSTOOLS over a set of benchmarks in the literature on barrier certificates generation. Among these benchmark examples, examples C1–C15 are semi-algebraic continuous systems and examples H1–H7 are semi-algebraic hybrid systems. The performance is reported in Table 1. All the experiments were performed on 2.6 GHz Intel i5 processor under Windows 10 with 8 GB RAM.

Table 1. Algorithm performance on benchmarks

ID	n	$\lvert L\rvert$	$d_{\mathbf{f}}$	BMI									LMI	
				t	N	SISBMI			PENBMI				SOSTOOLS	
						d_s	I_s	T_s	d_p	I_p	T_p		d_l	T_l
C1 from [33]	2	1	3	21	33	2	32	0.2189	2	24	0.9198		2	0.1949
C2 from [24]	2	1	1	30	58	4	73	0.5475	—				—	
C3 from [21]	2	1	3	21	39	2	29	0.2761	2	22	1.3353		—	
C4 from [30]	3	1	2	32	72	2	44	0.4126	2	23	1.8237		2	0.3245
C5 from [26]	3	1	3	32	72	2	47	0.4761	2	28	1.5435		2	0.3362
C6 from [3]	3	1	2	78	396	4	83	4.3598	—				—	
C7 from [28]	4	1	3	50	145	2	72	3.9577	2	28	21.0502		2	3.8658
C8 from [9]	3	1	2	32	72	—			2	40	2.4555		—	
C9 from [6]	4	1	2	31	86	—			2	42	4.6909		—	
C10 from [13]	7	1	2	73	394	2	112	10.7156	2	44	108.5615		2	7.2807
C11 from [13]	9	1	2	102	908	2	264	20.6856	2	30	272.4551		2	15.8167
C12 from [8]	12	1	1	70	123	2	108	3.2712	—				—	
H1 from [25]	2	2	2	38	65	2	61	0.4899	2	25	2.1499		2	0.2074
H2 from [36]	2	2	3	42	69	2	77	0.6331	2	24	2.2786		2	0.2265
H3 from [15]	2	2	2	75	138	2	115	3.7394	—				—	
H4 from [2]	2	3	1	42	89	1	70	0.5326	1	21	0.9968		2	0.1856
H5 from [1]	3	3	1	67	64	2	112	1.0864	—				—	
H6 from [7]	4	6	2	840	2736	2	616	48.0548	—				—	
H7 from [20]	3	2	3	170	899	4	219	18.7912	4	32	243.9832		—	

In Table 1, n denotes the number of the system variables, and $|L|$ denotes the number of locations; $d_{\mathbf{f}}$ denotes the maximal degree of the polynomials in the vector fields; t is the dimension of the matrix $\mathcal{B}(\mathbf{u}, \mathbf{v})$, and N refers to the number of decision variables appearing in the BMI problem (9), namely, $\dim(\mathbf{u})+\dim(\mathbf{v})$; d_s, d_p and d_l denote the degrees of the barrier certificates obtained via SISBMI, PENBMI and SOSTOOLS, respectively; I_s and I_p are the numbers of iterations used by SISBMI and PENBMI, respectively; T_s, T_p and T_l record the time spent by computation in seconds; the symbol—means that the solver was unable to return a feasible solution with the degree bound $\deg(B) \leq 6$.

Table 1 shows that for the 19 examples, our SISBMI solver can successfully handle 17 of them while the numbers of successful examples of PENBMI and SOSTOOLS are 13 and 9, respectively. Our SISBMI solver seems to provide the best solving capability. There are 10 examples that can be treated by BMI solvers (either SISBMI or PENBMI) unable to be solved by the LMI solver SOSTOOLS due to the more conservative conditions in the corresponding LMI problems. To evaluate the best performance of SOSTOOLS, we have tried some widely used multipliers [16,20], such as $0, \pm 1, \pm(1+x_1^2+\cdots+x_n^2)$, as well as some polynomial multipliers with random coefficients and the prior degree bound that guarantee the degrees of the polynomials involved in the verification conditions (6) do not increase. Examples C8-C9 show the case where the solver PENBMI performs better than our SISBMI solver as a result of the fact that both SISBMI and PENBMI solvers only find local optimal solutions to the BMI problems.

The above analysis on effectiveness can also be used to support that our SISBMI solver is a necessary complement to the existing tools. As shown in Table 1, PENBMI solver can cover 13 examples. To solve the remaining 6 examples, it has to resort to the SISBMI solver.

Considering the efficiency, the solver SOSTOOLS performs the best for almost all the successful examples because of the much lower computational complexity for solving the relaxed LMI problems. The efficiency comparison between SISBMI and PENBMI solvers can be made by examining the ratio between the execution times of these two solvers in Table 1. For the 11 examples that are solved by both tools, on average, our SISBMI solver costs 3.4 times than PENBMI solver in the number of iterations while only costs 0.27 times than PENBMI solver in time. That is for all the successful examples, our SISBMI solver takes much less time than PENBMI solver even it spends more iterations, which complies with the complexity analysis of the underlying algorithms. Both the theoretical analysis and the experiments support that our SISBMI solver is more efficient than PENBMI solver.

6 Related Work

In theory, the problem of barrier certificate generation is a quantifier elimination problem. The verification conditions corresponding to a barrier certificate can be encoded into a set of constraints on state variables and coefficients where the unknown coefficients are existentially quantified and state variables are universally quantified. Hence, several symbolic computation approaches [11,19,29],

such as cylindrical algebraic decomposition (CAD) or Grönber bases computation, have been directly applied to attack the associated quantifier elimination problems. However, due to the high computational complexity, they suffer from the scalability problem.

Due to the relatively low computational complexity, SOS relaxation based methods become popular. Rather than directly handling quantified constraints, they transform them to a non-convex bilinear matrix inequality. Z. Yang et al. [35] relied on the BMI solver PENBMI to compute exact polynomial barrier certificates. O. Bouissou et al. [3] applied interval analysis to handle the BMI problem derived from the dynamical systems whose initial and unsafe regions are restricted to the box form. G. Jessica et al. [10] presented an augmented Lagrangian framework for the special case of bilinear programs that arise from data flow constraints and correspond to the construction of numerical abstract domains aiming at safety verification.

To alleviate its computational intractability, a convex surrogate has been proposed that behaves fairly well. Specifically, once the multipliers are fixed, the BMI problem is further transformed into a LMI problem that can be quickly solved by convex optimization. S. Prajna et al. [20] had first put the idea forward. A. Sogokon et al. [34] employed the comparison principle associated with the convex verification conditions, to generate vector barrier certificates in safety verification.

Inspired by the fact that it is the non-convex feature of verification conditions prevents well-developed convex optimization to be directly applied, many convex but stronger verification conditions are studied. H. Kong et al. [16] proposed an exponential condition for semi-algebraic hybrid systems. Kapinski et al. [12] diagnosed convex verification conditions to Lyapunov-based barrier certificates. C. Sloth et al. [32] considered convex barrier certificates associated with compositional conditions for a group of interconnected hybrid systems. L. Dai et al. [4] studied how to balance the convexity of verification conditions with the expressiveness of barrier certificates. All these convex verification conditions are equivalent forms of LMI problems. They facilitate problem-solving at the risk of losing feasible solutions.

7 Conclusion

We have presented a sequential iterative scheme for solving the BMI problem derived from the barrier certificate generation of semi-algebraic hybrid systems. Taking advantage of the special feature of the bilinear terms, the proposed approach is more efficient than the existing BMI solver. Furthermore, compared with popular LMI solving based methods, the solving procedure does not make the verification condition more conservative, and thus reduces the risk of missing solutions. In virtue of the two appealing features, our approach can produce barrier certificates not amenable to existing methods, which is evidenced by a theoretical complexity analysis as well as the experiment on some benchmarks.

References

1. Alur, R., et al.: The algorithmic analysis of hybrid systems. Theoret. Comput. Sci. **138**(1), 3–34 (1995). https://doi.org/10.1016/0304-3975(94)00202-T
2. Alur, R., Dang, T., Ivančić, F.: Predicate abstraction for reachability analysis of hybrid systems. ACM Trans. Embedded Comput. Syst. (TECS) **5**(1), 152–199 (2006). https://doi.org/10.1145/1132357.1132363
3. Bouissou, O., Chapoutot, A., Djaballah, A., Kieffer, M.: Computation of parametric barrier functions for dynamical systems using interval analysis. In: Proceedings of the IEEE 53rd Annual Conference on Decision and Control (CDC), pp. 753–758. IEEE (2014). https://doi.org/10.1109/CDC.2014.7039472
4. Dai, L., Gan, T., Xia, B., Zhan, N.: Barrier certificates revisited. J. Symbol. Comput. **80**, 62–86 (2017). https://doi.org/10.1016/j.jsc.2016.07.010
5. Demmel, J.: Matrix computations. SIAM Rev. **28**(2), 252–255 (1986)
6. Ferragut, A., Gasull, A.: Seeking Darboux polynomials. Acta Applicandae Mathematicae **139**(1), 167–186 (2014). https://doi.org/10.1007/s10440-014-9974-0
7. Fisher, M.E.: A semiclosed-loop algorithm for the control of blood glucose levels in diabetics. IEEE Trans. Biomed. Eng. **38**(1), 57–61 (1991). https://doi.org/10.1109/10.68209
8. Gao, S.: Quadcopter model. https://github.com/dreal/benchmarks
9. Goubault, E., Jourdan, J.H., Putot, S., Sankaranarayanan, S.: Finding non-polynomial positive invariants and Lyapunov functions for polynomial systems through Darboux polynomials. In: Proceedings of the 2014 American Control Conference (ACC), pp. 3571–3578. IEEE (2014). https://doi.org/10.1109/ACC.2014.6859330
10. Gronski, J., Ben Sassi, M.-A., Becker, S., Sankaranarayanan, S.: Template polyhedra and bilinear optimization. Formal Methods Syst. Des. **54**(1), 27–63 (2018). https://doi.org/10.1007/s10703-018-0323-1
11. Gulwani, S., Tiwari, A.: Constraint-based approach for analysis of hybrid systems. In: Proceedings of the 20th International Conference on Computer Aided Verification (CAV), pp. 190–203 (2008). https://doi.org/10.1007/978-3-540-70545-1_18
12. Kapinski, J., Deshmukh, J.V., Sankaranarayanan, S., Aréchiga, N.: Simulation-guided Lyapunov analysis for hybrid dynamical systems. In: Proceedings of the Hybrid Systems: Computation and Control (HSCC), pp. 133–142. ACM (2014). https://doi.org/10.1145/2562059.2562139
13. Klipp, E., Herwig, R., Kowald, A., Wierling, C., Lehrach, H.: Systems Biology in Practice: Concepts, Implementation and Application. Wiley-Blackwell, Weinheim (2005)
14. Kočvara, M., Stingl, M.: PENNON: a code for convex nonlinear and semidefinite programming. Optim. Methods Softw. **18**(3), 317–333 (2003). https://doi.org/10.1080/1055678031000098773
15. Kong, H., Bogomolov, S., Schilling, C., Jiang, Y., Henzinger, T.A.: Safety verification of nonlinear hybrid systems based on invariant clusters. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, pp. 163–172. ACM (2017). https://doi.org/10.1145/3049797.3049814
16. Kong, H., He, F., Song, X., Hung, W.N.N., Gu, M.: Exponential-condition-based barrier certificate generation for safety verification of hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 242–257. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_17

17. Nocedal, J., Wright, S.: Numerical Optimization. Springer, Heidelberg (2006). https://doi.org/10.1007/978-0-387-40065-5
18. Platzer, A.: Virtual substitution & real arithmetic. Logical Foundations of Cyber-Physical Systems, pp. 607–628. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-63588-0_21
19. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. Formal Methods Syst. Des. **35**(1), 98–120 (2009). https://doi.org/10.1007/s10703-009-0079-8
20. Prajna, S., Jadbabaie, A.: Safety verification of hybrid systems using barrier certificates. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 477–492. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24743-2_32
21. Prajna, S., Jadbabaie, A., Pappas, G.J.: A framework for worst-case and stochastic safety verification using barrier certificates. IEEE Trans. Autom. Control **52**(8), 1415–1429 (2007). https://doi.org/10.1109/TAC.2007.902736
22. Prajna, S., Papachristodoulou, A., Parrilo, P.A.: SOSTOOLS: sum of squares optimization toolbox for MATLAB (2002). http://www.cds.caltech.edu/sostools
23. Putinar, M.: Positive polynomials on compact semi-algebraic sets. Indiana Univ. Math. J. **42**, 968–984 (1993)
24. Ratschan, S., She, Z.: Constraints for continuous reachability in the verification of hybrid systems. In: Calmet, J., Ida, T., Wang, D. (eds.) AISC 2006. LNCS (LNAI), vol. 4120, pp. 196–210. Springer, Heidelberg (2006). https://doi.org/10.1007/11856290_18
25. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation-based abstraction refinement. ACM Trans. Embedded Comput. Syst. **6**(1), 573–589 (2007). https://doi.org/10.1145/1210268.1210276
26. Ratschan, S., She, Z.: Providing a basin of attraction to a target region of polynomial systems by computation of Lyapunov-like functions. SIAM J. Control Optim. **48**(7), 4377–4394 (2010). https://doi.org/10.1137/090749955
27. Roux, P., Voronin, Y.-L., Sankaranarayanan, S.: Validating numerical semidefinite programming solvers for polynomial invariants. Formal Methods Syst. Des. **53**(2), 286–312 (2017). https://doi.org/10.1007/s10703-017-0302-y
28. Sankaranarayanan, S., Chen, X., Abrahám, E.: Lyapunov function synthesis using Handelman representations. In: The 9th IFAC Symposium on Nonlinear Control Systems, pp. 576–581 (2013). https://doi.org/10.3182/20130904-3-FR-2041.00198
29. Sankaranarayanan, S., Sipma, H., Manna, Z.: Constructing invariants for hybrid systems. Formal Methods Syst. Des. **32**(1), 25–55 (2008). https://doi.org/10.1007/s10703-007-0046-1
30. Sassi, M.A.B., Sankaranarayanan, S.: Stability and stabilization of polynomial dynamical systems using bernstein polynomials. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC 2015, Seattle, WA, USA, 14–16 April 2015, pp. 291–292 (2015). https://doi.org/10.1145/2728606.2728639
31. Sibai, H., Mitra, S.: State estimation of dynamical systems with unknown inputs: entropy and bit rates. In: Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control, pp. 217–226 (2018). https://doi.org/10.1145/3178126.3178150
32. Sloth, C., Pappas, G.J., Wisniewski, R.: Compositional safety analysis using barrier certificates. In: Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control, pp. 15–24. ACM (2012). https://doi.org/10.1145/2185632.2185639

33. Sogokon, A., Ghorbal, K., Johnson, T.T.: Non-linear continuous systems for safety verification (benchmark proposal). In: Applied Verification for Continuous and Hybrid Systems Workshop (ARCH) (2016)
34. Sogokon, A., Ghorbal, K., Tan, Y.K., Platzer, A.: Vector barrier certificates and comparison systems. In: Proceedings of the 22nd International Symposium on Formal Methods, pp. 418–437 (2018). https://doi.org/10.1007/978-3-319-95582-7_25
35. Yang, Z., Lin, W., Wu, M.: Exact verification of hybrid systems based on bilinear SOS representation. ACM Trans. Embedded Comput. Syst. **14**(1), 1–19 (2015). https://doi.org/10.1145/2629424
36. Zeng, X., Lin, W., Yang, Z., Chen, X., Wang, L.: Darboux-type barrier certificates for safety verification of nonlinear hybrid systems. In: Proceedings of the 2016 International Conference on Embedded Software (EMSOFT), pp. 1–10 (2016). https://doi.org/10.1145/2968478.2968484

Reachability Analysis Using Message Passing over Tree Decompositions

Sriram Sankaranarayanan(✉)

University of Colorado, Boulder, CO, USA
srirams@colorado.edu

Abstract. In this paper, we study efficient approaches to reachability analysis for discrete-time nonlinear dynamical systems when the dependencies among the variables of the system have low treewidth. Reachability analysis over nonlinear dynamical systems asks if a given set of target states can be reached, starting from an initial set of states. This is solved by computing conservative over approximations of the reachable set using abstract domains to represent these approximations. However, most approaches must tradeoff the level of conservatism against the cost of performing analysis, especially when the number of system variables increases. This makes reachability analysis challenging for nonlinear systems with a large number of state variables. Our approach works by constructing a dependency graph among the variables of the system. The tree decomposition of this graph builds a tree wherein each node of the tree is labeled with subsets of the state variables of the system. Furthermore, the tree decomposition satisfies important structural properties. Using the tree decomposition, our approach abstracts a set of states of the high dimensional system into a tree of sets of lower dimensional projections of this state. We derive various properties of this abstract domain, including conditions under which the original high dimensional set can be fully recovered from its low dimensional projections. Next, we use ideas from message passing developed originally for belief propagation over Bayesian networks to perform reachability analysis over the full state space in an efficient manner. We illustrate our approach on some interesting nonlinear systems with low treewidth to demonstrate the advantages of our approach.

1 Introduction

Reachability analysis asks whether a target set of states is reachable over a finite or infinite time horizon, starting from an initial set for a dynamical system. This problem is fundamental to the verification of systems, and is known to be challenging for a wide variety of models. This includes cyber-physical systems, physical and biological processes. In this paper, we study reachability analysis algorithms for nonlinear, discrete-time dynamical systems. The key challenge in analyzing such systems arises from the difficulty of representing the reachable sets of these systems. As a result, we resort to over-approximations of reachable sets using tractable set representations such as intervals [16], ellipsoids,

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 604–628, 2020.
https://doi.org/10.1007/978-3-030-53288-8_30

polyhedra [19], and low degree semi-algebraic sets [2]. Whereas these representations are useful for reachability analysis, they also trade off the degree of overapproximation in representing various sets against the complexity of performing operations such as intersections, unions, projections and image computations over these sets. The theory of abstract interpretation allows us to design various abstract domains that serve as representations for sets of states in order explore these tradeoffs [17,18,34]. However, for nonlinear dynamical systems, these representations often become too conservative or too expensive as the number of state variables grow.

In this paper, we study reachability analysis using the idea of tree decompositions over the dependency graph of a dynamical system. Tree decompositions are a well-known idea from graph theory [37], used to study properties of various types of graphs. The treewidth of a graph is an intrinsic property of a graph that relates to how "far away" a given graph is from a tree. For instance, trees are defined to have a treewidth of 1. Many commonly occurring families of graphs such as *series-parallel graphs* have treewidth 2 and so on. Formally, a tree decomposition of a graph is a tree whose nodes are associated with subsets of vertices of the original graph along with some key conditions that will be described in Sect. 2. We use tree decompositions to build an abstract domain. The abstraction operation projects a set of states in the full system state space along each of the nodes of the tree, yielding various projections of this set. The concretization combines projections back into the high dimensional set. We study various properties of this abstract domain. First, we characterize abstract elements that can potentially be generated by projecting some concrete elements along the nodes of the tree (so called *canonical* elements, Definition 10). Next we characterize those sets which can be abstracted along the tree decomposition and reconstructed without any loss in information (tree decomposable sets, Definition 11). In this process, we also derive a *message passing* approach wherein nodes of the tree can exchange information to help refine sets of states in a sound manner. However, as we will demonstrate, the abstraction is "lossy" in general since projections of tree decomposable sets are not necessarily tree decomposable. We discuss some interesting ways in which precision can be regained by carefully analyzing this situation.

We combine these ideas together into an approach for reachability analysis of nonlinear systems using a grid domain that represents complex non convex sets as a union of fixed size cells using a gridding of the state-space. Although such a domain would be prohibitively expensive, we show that the tree decomposition abstract domain can drastically cut down on the complexity of computing reachable set overapproximations in this domain, yielding precise reachable set estimation for some nonlinear systems with low treewidth. We demonstrate our approach using a prototype implementation to show that for a restricted class of systems whose dependency graphs have low treewidth, our approach can be quite efficient and precise at the same time. Although some interesting systems have low treewidth property, it is easy to see that many systems will have treewidths that are too high for our approach. Our future work will consider how systems

whose dependency graphs do not have sufficiently low treewidth can still be tackled in a conservative manner using some ideas from this paper.

1.1 Related Work

As mentioned earlier, the concept of tree decompositions and treewidth originated in graph theory [37]. The concept of treewidth gained popularity when it was shown that many NP-complete problems on graphs such as graph coloring could be solved efficiently for graphs with small treewidths [5]. Courcelle showed that the problem of checking if a given graph satisfies a formula in the monadic second order logic of graphs can be solved in linear time on graphs with bounded treewidth [15]. Several NP-complete problems such as 3-coloring can be expressed in this logic. Tree decompositions are also used to solve inference problems over Bayesian networks leading to representations of the Bayesian networks such as junction trees that share many of the properties of a tree decomposition [29]. In fact, belief propagation over junction trees is performed by passing messages that marginalize the probability distributions at various nodes of the tree. This is analogous to the message passing approach described here.

Tree decomposition techniques have been applied to model checking problems over finite state systems. For instance, Obdržálek show that the μ-calculus model checking problem can be solved in linear time in the size of a finite-state system whose graph has a bounded treewidth [35]. However, as Ferrara et al. point out, requiring the state graph of a system to have a bounded treewidth is often restrictive [24]. Instead, they study concurrent finite state systems wherein the communication graph has a bounded tree width. However, they conclude that while it is more reasonable to assume that the communication graph has a bounded tree width, it does not confer much advantages to verification problems. For instance, they show that the unrolling of these systems over time potentially results in unbounded treewidth. In this paper, we consider a different approach wherein we study the treewidth of dependency graphs of the system. We find that many systems have small treewidth and exploit this property. At the same time, we note that some of the benchmarks studied have "sparse" dependency graphs but treewidths that are too large for our approach.

Tree decomposition techniques have also been studied in static analysis of programs. The control and data flow graphs of structured programs without goto-statements or exceptional control flow are known to have small treewidth that can be exploited to perform compiler optimizations such as register allocation quite efficiently [38]. Chatterjee et al. have shown how to exploit small treewidth property of the control flow graphs of procedures in programs to perform interprocedural dataflow analysis by modeling the execution of programs with procedures as recursive state machines [11]. However, this approach seems restricted to control dominated properties such as sequence of function calls. In a followup work, they study control and data flow analysis problems for concurrent systems, wherein each component has constant treewidth [10]. In contrast, our approach studies dynamical system and consider tree decompositions of the data dependency graph.

The use of message passing in this paper closely resembles past work by Gulwani and Jojic [27]. Therein, a program verification problem involving the verification pre/post and intermediate assertions in a program is solved by passing messages that can propagate information between assertions along program paths in a randomized fashion. The approach is shown to be similar to loopy belief propagation used in Bayesian inference. The key differences are (a) we use data dependencies and tree decompositions rather than control flow paths to pass information along; and (b) we formally prove properties of the message passing algorithm.

Our approach is conceptually related to a well-known idea of speeding up static analysis of large programs using "packing" of program variables [4,28]. This approach was used successfully in the Astreé static analyzer [3,4,21]. Therein, clusters of variables representing small sets of dependent local and global are extracted. The remaining program variables are abstracted away and the abstract interpretation process is carried out over just these variables. The usefulness of this approach has borne out in other abstract interpretation efforts, including Varvel [28]. The key idea in this paper can be seen as a formalization of the rather informal "clustering" approach using tree decompositions. We demonstrate theoretical properties as well as the ability to pass messages to improve the results of the abstract interpretation.

The use of the dependency graph structure to speed up reachability analysis approaches has been explored in the past for speeding up Hamilton-Jacobi-based approaches by Mo Chen et al. [12] as well as flowpipe based approaches by Xin Chen et al. [13]. Both approaches consider the directed dependency graph wherein x_i is connected to x_j if the former appears in the dynamical update equation of the latter variable. The approaches perform a strongly connected component (SCC) decomposition and analyze each SCC in a topological sorted order. However, this approach breaks as soon as the system has large SCCs, which is common. As a result, Xin Chen et al. show how SCCs can themselves be broken into numerous subsets at the cost of a more conservative solution. In contrast, the tree decomposition approach can be applied to exploit sparsity even when the entire dependency graph is a single SCC.

2 Preliminaries

In this section, we will describe the system model under analysis, the dependency graph structure and the basics of tree decompositions. Let $X : \{x_1, \ldots, x_n\}$ be a set of *system variables* and $\mathbf{x} : X \mapsto \mathbb{R}$ represent a valuation to these system variables. Let D be the domain of all valuations of X, that describes the *state space* of the system. For convenience let $\mathbf{x}_i$ denote $\mathbf{x}(x_i)$. Also, let $W : \{w_1, \ldots, w_m\}$ represent disturbance variables and $\mathbf{w} : W \mapsto \mathbb{R}$ represent a vector of $m \geq 0$ external disturbance inputs that take values in some compact disturbance space $\mathcal{W}$.

Definition 1 (Dynamical Model). *A model Π is a tuple $\langle X, W, D, \mathcal{W}, f, X_0, U\rangle$, wherein $X, W, D, \mathcal{W}$ are as defined above, f is an arithmetic expression*

over variables in X, W describing the dynamics, X_0 is a set of possible initial valuations (states) and U is a designated set of unsafe states.

The dynamics are given by $\mathbf{x}(t+1) = \mathsf{eval}(f, \mathbf{x}, \mathbf{w})$, wherein eval *evaluates a given an expression f, a set of valuations to the system variables $\mathbf{x} \in D$ and disturbances $\mathbf{w} \in \mathcal{W}$, and returns a new set of valuations for each variable in X, denoted by $\mathbf{x}(t+1)$.*

For simplicity, we write $f(\mathbf{x}, \mathbf{w})$ to denote $\mathsf{eval}(f, \mathbf{x}, \mathbf{w})$ for a function expression f. A state of the system is a valuation $\mathbf{x} : X \mapsto \mathbb{R}$ such that $\mathbf{x} \in D$. Given a finite sequence of disturbance inputs $\mathbf{w}(0), \ldots, \mathbf{w}(T)$, for some $T \geq 0$ and $\mathbf{w}(i) \in \mathcal{W}$ for all $i \in [0, T]$, an execution of the system is a sequence of states $\mathbf{x}(0), \ldots, \mathbf{x}(T+1)$, such that $\mathbf{x}(0) \in X_0$, $\mathbf{x}(t) \in D$ for $t \in [0, T+1]$ and $\mathbf{x}(t+1) = f(\mathbf{x}(t), \mathbf{w}(t))$ for all $t \in [0, T]$. According to these semantics, the system may fail to have an execution for a given disturbance sequence $\mathbf{w}(t)$, $t \in [0, T]$ and initial state $\mathbf{x}(0)$ if for some state $\mathbf{x}(t)$, we have $f(\mathbf{x}(t), \mathbf{w}(t)) \notin D$.

A state $\mathbf{x}(t)$ is reachable (at time t) if there is an execution of the form $\mathbf{x}(0), \ldots, \mathbf{x}(t)$, satisfying the constraints above. We say that the unsafe state U is reachable iff some state $\mathbf{x} \in U$ is reachable. Furthermore, we say that U is reachable within a finite time horizon T, iff some state $\mathbf{x} \in U$ is reachable at time $t \in [0, T]$.

Example 1. Consider a nonlinear example of a dynamical model Π with state space $\mathbf{x} : (x_1, x_2, x_3)$ and $\mathbf{w} : (w_1)$. The dynamics can be written as parallel assignments to the state variables:

$$x_1 \ := \ x_1 + 0.25x_2 - 0.05x_1 sin(x_2), \ x_2 \ := \ x_2 + w_1, \ x_3 \ := x_3 - 0.2x_3x_2 \, ,$$

The assignments are all evaluated in parallel to update the current state $\mathbf{x}(t)$ to a new state $\mathbf{x}(t+1)$. The domain D is $x_i \in [-3, 3]$ for $i = 1, 2, 3$ and the disturbance $w_1 \in [-0.1, 0.1]$. The initial set X_0 is $x_1 \in [-0.2, 0.2] \ \wedge \ x_2 \in [-0.3, 0] \ \wedge \ x_3 \in [0, 0.4]$.

We will now define the dependency (hyper)graph of the system Π. For convenience, we write the update function (expression) f of a system Π in terms of individual updates $(f_1, \ldots, f_n)$, wherein $x_j' = f_j(\mathbf{x}, \mathbf{w})$. We say that system variable x_i (or disturbance variable w_j) is a *proper input* to the expression f_k if x_i (or w_j) occurs as a subterm in f_k. Let $\mathsf{inps}(f_k)$ denote the set of all proper input variables to the function (expression) f_k.

As an example, consider $X = \{x_1, \ldots, x_4\}$ and $W = \{w_1, w_2\}$ and the expression $f : x_1x_4 - w_1$. The proper inputs to f are $\{x_1, x_4, w_1\}$. We exclude cases such as $g : \frac{\sin^2(x_1) + \cos^2(x_1)}{\sin^2(x_2) + \cos^2(x_2)}$ that has $\{x_1, x_2\}$ as proper inputs. However a simplification using elementary trigonometric rules can eliminate them. We will assume that all expressions are simplified to involve the least number of variables.

Definition 2 (Dependency Hypergraph). *A dependency hypergraph of a system Π has vertices $V : X \cup W$, given by the union of the system and disturbance variables with hyperedge set $E \subseteq 2^V$ given by $E = \{e_1, \ldots, e_n\}$,*

wherein for each update $x_k := f_k(\mathbf{x}, \mathbf{w})$ *(*$k = 1, \ldots, n$*), we have the hyperedge* $e_k : \{x_k\} \cup \mathsf{inps}(f_k)$*. In other words, each update* $x_k := f_k(\mathbf{x}, \mathbf{w})$ *yields an edge that includes* x_k *along with all the system/disturbance variables that are proper inputs to* f_k*.*

Example 2. The dependency hypergraph for the system from Example 1 has the vertices $V : \{x_1, x_2, x_3, w_1\}$ and the edges $\{e_1 : \{x_1, x_2\},\ e_2 :\ \{x_2, w_1\}$ and $e_3 :\ \{x_2, x_3\}\}$.

2.1 Tree Decomposition

We will now discuss tree decompositions and the associated concept of treewidth of a hypergraph $G : (V, E)$. The tree decomposition will be applied to the dependency hypergraphs (Definition 2) for systems Π (Definition 1).

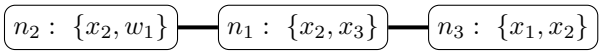
Definition 3 (Tree Decomposition and Treewidth). *Given a hypergraph* $G : (V, E)$*, a tree decomposition is a tree* $T : (N, C)$ *and a mapping* $\textsc{verts} : N \mapsto 2^V$*, wherein* N *is the set of tree nodes,* C *is the set of tree edges and* $\textsc{verts}(\cdot)$ *associates each node* $u \in N$ *with a set of graph vertices* $\textsc{verts}(n) \subseteq V$*. The tree decomposition satisfies the following conditions:*

1. *For vertex* $v \in V$ *there exists (at least one)* $n \in N$ *such that* $v \in \textsc{verts}(n)$*.*
2. *For each hyperedge* $e \in E$ *there exists (at least one)* $n \in N$*:* $e \subseteq \textsc{verts}(n)$*.*
3. *For each vertex* v*, for any two nodes* n_1, n_2 *such that* $v \in \textsc{verts}(n_1)$ *and* $v \in \textsc{verts}(n_2)$*, then* $v \in \textsc{verts}(n)$ *for each node* n *along the unique path between* n_1 *and* n_2 *in the tree. Stated another way, the subset of nodes* $N_v : \{n \in N \mid v \in \textsc{verts}(n)\}$ *induces a subtree of* T *(denoted* T_v*).*

The width of a tree decomposition is given by $\max\{|\textsc{verts}(n)| \mid n \in N\} - 1$*. In other words, we find the node* n *in the tree whose associated set of vertices has the largest cardinality. We subtract one from this maximal cardinality to obtain the treewidth. A tree decomposition is optimal for a graph* G *if no other tree decomposition exists with a strictly smaller width. The treewidth of a hypergraph* G *is given by width of an optimal tree decomposition.*

It is easy to show that if the graph G is a tree, it has treewidth 1. Likewise, a cycle has tree width 2.

Example 3. The tree decomposition of the hypergraph G from Example 2 has three nodes $\{n_1, n_2, n_3\}$ with edges (n_1, n_2) and (n_2, n_3). The nodes along with the associated vertex sets are as follows:



Although the tree decomposition is not a rooted tree, we often designate an arbitrary node $r \in N$ as the root node, and consider the tree T as a rooted tree with root r.

Finding a Tree Decomposition: Interestingly, the problem of finding the treewidth of a graph is itself a NP-hard problem. However, many practical approaches exist for graphs with small treewidths. For instance, Bodlaender presents an algorithm that runs in time $O(k^{O(k^3)})$ to construct a tree decomposition of width at most k or conclude that the treewidth of the graph is at least $k+1$ [6]. Such an approach can be quite useful if a given graph is suspected to have a small tree width in the first place. Besides this, many efficient algorithms exist to approximate the treewidth of a graph to some constant factor. A detailed survey of these results is available elsewhere [7,8]. Open-source packages such as HTD can compute treewidth for graphs with thousands of nodes [1]. Finally, we note that if a tree decomposition of width k can be found, then one can be found with at most $|V|$ nodes.

Lemma 1. *Let T be a tree decomposition for a (multi)graph G with vertices V and treewidth k. There exists a tree decomposition $\hat{T}$ of G with the same treewidth k, and at most $|V|$ nodes.*

A proof is provided in the extended version of the paper.

3 Abstract Domains Using Tree Decompositions

In this section, we will define abstract domains using tree decompositions of the dependency hypergraph of the system under analysis. Let Π be a transition system over system variables X. The concrete states are given by $\mathbf{x} \in D$, wherein $\mathbf{x} : X \mapsto \mathbb{R}$ maps each state variable $x_j \in X$ to its value $\mathbf{x}(x_j)$ (denoted $\mathbf{x}_j$).

Definition 4 (Projections). *The projection of a state $\mathbf{x}$ to a subset of state variables $J \subseteq X$, denoted as $\mathsf{proj}(\mathbf{x}, J)$, is a valuation $\hat{\mathbf{x}} : J \mapsto \mathbb{R}$ such that $\hat{\mathbf{x}}(x_i) = \mathbf{x}(x_i)$ for all $x_i \in J$. For a set of states $S \subseteq D$ and a subset of state variables $J \subseteq X$, we denote the projection of S along (the dimensions of) J as $\mathsf{proj}(S, J) : \{\mathsf{proj}(\mathbf{x}, J) \mid \mathbf{x} \in S\}$.*

Definition 5 (Extensions). *Let R be a set of states involving just the variables in the set $J_1 \subseteq X$, i.e, $R \subseteq \mathsf{proj}(D, J_1)$. We define the* extension *of R into a set of variables $J_2 \supseteq J_1$ as $\mathsf{ext}_{J_2}(R) : \{\mathbf{x} \in \mathsf{proj}(D, J_2) \mid \mathsf{proj}(\mathbf{x}, J_1) \in R\}$.*

In other words, the extension of a set embeds each element in the larger dimensional space defined by J_2 allowing "all possible values" for the dimensions in $J_2 \setminus J_1$.

We will use the notation $\mathsf{ext}(S)$ to denote the set $\mathsf{ext}_X(S)$, i.e, its extension to the entire set of state variables X. For a state $\mathbf{x}_S$, we will use $\mathsf{ext}(\mathbf{x}_S)$ denote $\mathsf{ext}(\{\mathbf{x}_S\})$.

Definition 6 (Product (Join) of Sets). *Let $R_1 \subseteq \mathsf{proj}(D, J_1)$ and $R_2 \subseteq \mathsf{proj}(D, J_2)$. We define $R_1 \otimes R_2 : \{\mathbf{x} : J_1 \cup J_2 \mapsto \mathbb{R} \mid \mathsf{proj}(\mathbf{x}, J_1) \in R_1$ and $\mathsf{proj}(\mathbf{x}, J_2) \in R_2\}$.*

Let $T : (N, C)$ be a tree decomposition of the dependency hypergraph of the system. Recall that for each node $n \in N$ we associate a set of system/disturbance variables denoted by $\text{VERTS}(n)$. Let $\text{VERTS}_X(n)$ denote the set of system variables: $\text{VERTS}(n) \cap X$. We say that an update function $x_k := f_k(\mathbf{x}, \mathbf{w})$ is associated with a node n in the tree iff $\{x_k\} \cup \mathsf{inps}(f_k) \subseteq \text{VERTS}(n)$.

Lemma 2. *For every system variable x_k, its update $x_k := f_k(\mathbf{x}, \mathbf{w})$ is associated with at least one node $n \in N$.*

Proof. This follows from those of a tree decomposition that states that every hyperedge in the dependency hypergraph must belong to $\text{VERTS}(n)$ for at least one node $n \in N$.

3.1 Abstraction and Concretization

We consider subsets of the concrete states for the system Π, i.e, the set 2^D, ordered by set inclusion as our *concrete domain*. Given a tree decomposition, T, we define an abstract domain through projection of a concrete set along $\text{VERTS}(n)$ for each node n of T.

Definition 7 (Abstract Domain). *Each element s of the abstract domain $\mathbb{A}_T$ is a mapping that associates each node $n \in N$ with a set $s(n) \subseteq \mathsf{proj}(D, \text{VERTS}_X(n))$.*

For $s_1, s_2 \in \mathbb{A}_T$, $s_1 \sqsubseteq s_2$ iff $s_1(n) \subseteq s_2(n)$ for each $n \in N$.

We will use the notation $\mathsf{proj}(S, n)$ for a node $n \in N$ to denote $\mathsf{proj}(S, \text{VERTS}_X(n))$.

Definition 8 (Abstraction Map). *Given a tree decomposition T, the abstraction map α_T takes a set of states $S \subseteq D$ and produces a mapping that associates tree node $n \in N$ to a projection of S along the variables $\text{VERTS}_X(n)$. Formally,*

$$\alpha_T(S) : \ \lambda n : N.\ \mathsf{proj}(S, n)\,.$$

Thus, an abstract state s is a map that associates each node n of the tree to a set $s(n) \subseteq D_n$. We now define the concretization map γ_T.

Definition 9 (Concretization Map). *The concretization $\gamma_T(s)$ of an abstract state is defined as $\gamma_T(s) : \bigcap_{n \in N} \mathsf{ext}(s(n))$. In other words, we take $s(n)$ for every node $n \in N$, extend it to the full dimensional space of all system variables and intersect the result over all nodes $n \in N$.*

Example 4. Consider a simple tree decomposition T with 2 nodes n_1, n_2 and a single edge (n_1, n_2). Let $\text{VERTS}(n_1) : \{x_1, x_2\}$ and $\text{VERTS}(n_2) : \{x_2, x_3\}$. Let the domain D be the set $\mathbf{x}_i \in \{1, 2, 3\}$ for $i = 1, 2, 3$. We use the notation $(\underset{\_}{v_1}^{x_1}, \underset{\_}{v_2}^{x_2}, \underset{\_}{v_3}^{x_3})$ to denote a state $\mathbf{x}$ that maps x_1 to the value v_1, x_2 to the value v_2 and so on.

Now consider the set $S = \{(\overset{x_1}{1}, \overset{x_2}{1}, \overset{x_3}{1}), (\overset{x_1}{1}, \overset{x_2}{1}, \overset{x_3}{2}), (\overset{x_1}{1}, \overset{x_2}{2}, \overset{x_3}{3})\}$. We have that $s : \alpha(S)$ is the mapping that projects S onto the dimensions (x_1, x_2) for node n_1 and (x_2, x_3) for node n_2:

$$n_1 \mapsto \{(\overset{x_1}{1}, \overset{x_2}{1}), (\overset{x_1}{1}, \overset{x_2}{2})\},\ n_2 \mapsto \{(\overset{x_2}{1}, \overset{x_3}{1}), (\overset{x_2}{1}, \overset{x_3}{2}), (\overset{x_2}{2}, \overset{x_3}{3})\}.$$

Likewise, we verify that the concretization map $\gamma(s)$ will yields us:

$$\gamma(s) : \{(\overset{x_1}{1}, \overset{x_2}{1}, \overset{x_3}{1}), (\overset{x_1}{1}, \overset{x_2}{1}, \overset{x_3}{2}), (\overset{x_1}{1}, \overset{x_2}{2}, \overset{x_3}{3})\}.$$

For convenience, if the tree T is clear from the context, we will drop the subscripts to simply write α and γ for the abstraction and concretization map, respectively.

Theorem 1. *For any tree decomposition T, the maps α and γ form a Galois connection. I.e, for all $S \subseteq D$ and $s \in \mathbb{A}_T$: $\alpha(S) \sqsubseteq s$ iff $S \subseteq \gamma(s)$.*

Proof. Let S, s be such that $\alpha(S) \sqsubseteq s$. Therefore, $\mathsf{proj}(S, n) \subseteq s(n)\ \forall n \in N$ by the definition of $\sqsubseteq$. Pick any, $\mathbf{x} \in S$. First, $\mathsf{proj}(\mathbf{x}, n) \in \mathsf{proj}(S, n)$ and therefore, $\mathsf{proj}(\mathbf{x}, n) \in s(n)$ for all $n \in N$. Thus, $\mathbf{x} \in \mathsf{ext}(s(n))$ for each node $n \in N$. Therefore, $\mathbf{x} \in \bigcap_{n \in N} \mathsf{ext}(s(n))$, and hence, $\mathbf{x} \in \gamma(s)$, by defn. of γ. Therefore, $S \subseteq \gamma(s)$.

Conversely, assume $S \subseteq \gamma(s)$. Since $\gamma(s) = \bigcap_{n \in N} \mathsf{ext}(s(n))$ (from Definition 9). Therefore, $S \subseteq \mathsf{ext}(s(n))$ forall $n \in N$. Therefore, for all $\mathbf{x} \in S$, $\mathsf{proj}(\mathbf{x}, n) \in s(n)$. Therefore, $\mathsf{proj}(S, n) \subseteq s(n)$ for every $n \in N$. Finally, this yields $\alpha(S) \sqsubseteq s$.

The meet operation is defined as $s_1 \sqcap s_2 :\ \lambda n.\ s_1(n) \cap s_2(n)$, and likewise, the join is defined as $s_1 \sqcup s_2 :\ \lambda n.\ s_1(n) \cup s_2(n)$. We recall two key facts that follow from Galois connection between α and γ.

1. For any set $S \subseteq D$, we have $S \subseteq \gamma(\alpha(S))$. Abstracting a concrete set and concretizing it back again "loses information". To see why, we start from $\alpha(S) \sqsubseteq \alpha(S)$ and apply the Galois connection to derive $S \subseteq \gamma(\alpha(S))$.
2. Likewise, for any abstract domain object $s \in \mathbb{A}$, we have $\alpha(\gamma(s)) \sqsubseteq s$. I.e, for any element s, taking its concretization and abstracting it "gains information". To prove this, we start from $\gamma(s) \subseteq \gamma(s)$ and conclude that $\alpha(\gamma(s)) \sqsubseteq s$.

Example 5. Returning back to Example 4, now consider the set

$$\hat{S} = \{(\overset{x_1}{1}, \overset{x_2}{1}, \overset{x_3}{2}), (\overset{x_1}{1}, \overset{x_2}{2}, \overset{x_3}{3}), (\overset{x_1}{2}, \overset{x_2}{1}, \overset{x_3}{2}), (\overset{x_1}{2}, \overset{x_2}{2}, \overset{x_3}{4})\}.$$

Its abstraction $\hat{s} : \alpha(\hat{S})$ is given by the mapping:

$$n_1 \mapsto \{(\overset{x_1}{1}, \overset{x_2}{1}), (\overset{x_1}{1}, \overset{x_2}{2}), (\overset{x_1}{2}, \overset{x_2}{1}), (\overset{x_1}{2}, \overset{x_2}{2})\},\ n_2 \mapsto \{(\overset{x_2}{1}, \overset{x_3}{2}), (\overset{x_2}{2}, \overset{x_3}{3}), (\overset{x_2}{2}, \overset{x_3}{4})\}.$$

We note that $\gamma(\hat{s})$ is the set: $\{(\overset{x_1}{1}, \overset{x_2}{1}, \overset{x_3}{2}), (\overset{x_1}{1}, \overset{x_2}{2}, \overset{x_3}{3}), (\overset{x_1}{1}, \overset{x_2}{2}, \overset{x_3}{4}), (\overset{x_1}{2}, \overset{x_2}{1}, \overset{x_3}{2}), (\overset{x_1}{2}, \overset{x_2}{2}, \overset{x_3}{3}), (\overset{x_1}{2}, \overset{x_2}{2}, \overset{x_3}{4})\}$. Thus $\hat{S} \subseteq \gamma(\hat{s})$. Notice that $(\overset{x_1}{2}, \overset{x_2}{2}, \overset{x_3}{3})$ and $(\overset{x_1}{1}, \overset{x_2}{2}, \overset{x_3}{4})$ are part of $\gamma(\hat{s})$ but not the original set $\hat{S}$. Similarly, consider the abstract element s_1: $n_1 \mapsto \{(\overset{x_1}{1}, \overset{x_2}{1}), (\overset{x_1}{1}, \overset{x_2}{2})\},\ n_2 \mapsto \{(\overset{x_2}{1}, \overset{x_3}{3})\}$. We note that $\gamma(s_1) : \{(\overset{x_1}{1}, \overset{x_2}{1}, \overset{x_3}{3})\}$ and therefore $\alpha(\gamma(s_1))$ yields the abstract element $s_2 \sqsubseteq s_1$: $n_1 \mapsto \{(\overset{x_1}{1}, \overset{x_2}{1})\},\ n_2 \mapsto \{(\overset{x_2}{1}, \overset{x_3}{3})\}$.

3.2 Canonical Elements and Message Passing

In the tree decomposition, various nodes share information about the subsets of vertices associated with each node. Since the subsets have elements in common, it is possible that a node n_1 has information about a variable x_2 that is also present in some other node n_2 of the tree. We will now see how to take an abstract element s and refine each $s(n)$ by exchanging information between nodes in a systematic manner.

For each edge $(n_1, n_2) \in C$ of the tree, define the set of variables in common as $\mathrm{CV}(n_1, n_2)$: $\textsc{verts}(n_1) \cap \textsc{verts}(n_2)$ and $\mathrm{CV}_X(n_1, n_2)$: $\textsc{verts}_X(n_1) \cap \textsc{verts}_X(n_2)$.

Definition 10 (Canonical Elements). *An abstract element s is said to be* canonical *if and only if for each edge $(n_1, n_2) \in C$ in the tree:*

$$\mathsf{proj}(s(n_1), \mathrm{CV}_X(n_1, n_2)) = \mathsf{proj}(s(n_2), \mathrm{CV}_X(n_1, n_2))\,.$$

In other words, if we took the common variables $\textsc{verts}_X(n_1) \cap \textsc{verts}_X(n_2)$, the set $s(n_1)$ projected along these common variables is equal to the projection of $s(n_2)$ along the common variables.

Example 6. Consider the abstract element s_1 from Example 5: $n_1 \mapsto \{(\overset{x_1}{1}, \overset{x_2}{1}), (\overset{x_1}{1}, \overset{x_2}{2})\}$, $n_2 \mapsto \{(\overset{x_2}{1}, \overset{x_3}{3})\}$. $\mathsf{proj}(s_1(n_1), \mathrm{CV}(n_1, n_2))$ is the set $\{\overset{x_2}{1}, \overset{x_2}{2}\}$ whereas $\mathsf{proj}(s_1(n_2), \mathrm{CV}(n_1, n_2))$ is simply $\{\overset{x_2}{1}\}$. Therefore, s_1 fails to be canonical.

The key theorem of tree decomposition is that a canonical element in the abstract domain can be seen as the projection of a concrete set S along $\textsc{verts}_X(n)$ for each node n of the tree. To prove that we will first establish a useful property of a canonical element s.

Lemma 3. *For every canonical element $s \in \mathbb{A}$, node $n \in N$ and element $\mathbf{x}_n \in s(n)$, we have that $\mathsf{ext}(\mathbf{x}_n) \cap \gamma(s) \neq \emptyset$.*

Stated another way, the lemma claims that for any canonical s, any $\mathbf{x}_n \in s(n)$ can be extended to form some element of $\gamma(s)$. A proof is provided in the extended version.

Theorem 2. *An element s is canonical (Definition 10) if and only if $s = \alpha(S)$ for some concrete set S.*

Ideally, in abstract interpretation, we would like to work with abstract domain objects that satisfy $s = \alpha(\gamma(s))$. One way to ensure that is to take any given domain element s_0 and simply calculate out $\alpha(\gamma(s_0))$ by applying the maps. However, $\gamma(s_0)$ in our domain takes lower dimensional projections and reconstructs a set in the full states pace. It may thus be too expensive to compute. Fortunately, canonical objects satisfy the equality $s = \alpha(\gamma(s))$. Therefore, given any object $s \in \mathbb{A}$ that is not necessarily canonical, we would like to make it canonical: I.e, we seek an object $\hat{s}$ such that $\gamma(\hat{s}) = \gamma(s)$, but $\hat{s}$ is canonical. As

mentioned earlier, directly computing $\hat{s} = \alpha(\gamma(s))$ can be prohibitively expensive, depending on the domain. We now describe a *message passing* approach.

First, we convert the tree T to a rooted tree by designating an arbitrary node $r \in N$ as the root of the tree.

Message Passing along Edges: Let (n_1, n_2) be an edge of the tree and s be an abstract element. A message from n_1 to n_2 is defined as the set $\mathsf{msg}(s, n_1 \to n_2) : \mathsf{proj}(s(n_1), \mathrm{CV}(n_1, n_2))$. In other words, we project the set $s(n_1)$ along the dimensions that are common to (n_1, n_2).

Once a node n_2 receives $M : \mathsf{msg}(s, n_1 \to n_2)$, it processes the message by updating $s(n_2)$ as $s(n_2) := s(n_2) \cap \mathsf{ext}_{\mathrm{VERTS}(n_2)}(M)$. In other words, it intersects the message (extended to the dimensions in n_2) with the current set that is associated with n_2.

Example 7. Consider a tree decomposition with three nodes $\{n_1, n_2, n_3\}$ and the edges (n_1, n_2) and (n_2, n_3). Let $\textsc{verts}(n_1) : \{x_1, x_2\}$, $\textsc{verts}(n_2) : \{x_2, x_4\}$ and $\textsc{verts}(n_3) : \{x_2, x_3\}$. Let D be the domain $\{1, 2, 3, 4\}^4$. Consider the abstract element s:

$$n_1 \mapsto \{(\overset{x_1}{1}, \overset{x_2}{2}), (\overset{x_1}{3}, \overset{x_2}{3}), (\overset{x_1}{1}, \overset{x_2}{4})\},\ n_2 \mapsto \{(\overset{x_2}{1}, \overset{x_4}{1}), (\overset{x_2}{2}, \overset{x_4}{2}), (\overset{x_2}{3}, \overset{x_4}{3}), (\overset{x_2}{4}, \overset{x_4}{4})\},\ n_3 \mapsto \{(\overset{x_2}{4}, \overset{x_3}{4}), (\overset{x_2}{2}, \overset{x_3}{3})\}.$$

A message $\mathsf{msg}(s, n_1 \to n_2)$ is given by the set $\mathsf{proj}(s(n_1), \{x_2\}) : \{\overset{x_2}{2}, \overset{x_2}{3}, \overset{x_2}{4}\}$. This results in the new abstract object s' wherein the element $(\overset{x_2}{1}, \overset{x_4}{1})$ is removed from $s(n_2)$:

$$n_1 \mapsto \{(\overset{x_1}{1}, \overset{x_2}{2}), (\overset{x_1}{3}, \overset{x_2}{3}), (\overset{x_1}{1}, \overset{x_2}{4})\},\ n_2 \mapsto \{\cancel{(\overset{x_2}{1}, \overset{x_4}{1})}, (\overset{x_2}{2}, \overset{x_4}{2}), (\overset{x_2}{3}, \overset{x_4}{3}), (\overset{x_2}{4}, \overset{x_4}{4})\},\ n_3 \mapsto \{(\overset{x_2}{4}, \overset{x_3}{4}), (\overset{x_2}{2}, \overset{x_3}{3})\}.$$

Upwards Message Passing: The upwards message passing works from leaves up to the root of the tree according to the following two rules:

1. First, each leaf of the tree n passes a message to its parent n_p. The parent node n_p intersects its current value $s(n_p)$ with the message to update its current set.
2. After a node has received (and processed) a message from all its children, it passes a message up to its parent, if one exists.

The upwards message passing terminates at the root since it does not have a parent to send a message to.

Example 8. Going back to Example 7, we designate n_2 as the root and the upwards pass sends the messages $\mathsf{msg}(s, n_1 \to n_2)$ and $\mathsf{msg}(s, n_3 \to n_2)$. This results in the following updated element:

$$n_1 \mapsto \{(\overset{x_1}{1}, \overset{x_2}{2}), (\overset{x_1}{3}, \overset{x_2}{3}), (\overset{x_1}{1}, \overset{x_2}{4})\},\ n_2 \mapsto \{\cancel{(\overset{x_2}{1}, \overset{x_4}{1})}, (\overset{x_2}{2}, \overset{x_4}{2}), \cancel{(\overset{x_2}{3}, \overset{x_4}{3})}, (\overset{x_2}{4}, \overset{x_4}{4})\},\ n_3 \mapsto \{(\overset{x_2}{4}, \overset{x_3}{4}), (\overset{x_2}{2}, \overset{x_3}{3})\}.$$

Downwards Message Passing: The downwards message passing works from the root down to the leaves.

1. To initialize, the root sends a message to all its children.
2. After a node has received (and processed) a message from its parent, it sends a message to all its children.

The overall procedure to make a given abstract object s canonical is as follows: (a) perform an upwards message passing phase and (b) perform a downwards message passing phase.

Example 9. Going back to Example 8, the downward message passing phase sends messages from $n_2 \to n_1$ and $n_2 \to n_3$. The resulting element $\hat{s}$ is

$$n_1 \mapsto \{(\overset{x_1}{1},\overset{x_2}{2}), (\cancel{\overset{x_1}{3},\overset{x_2}{3}}), (\overset{x_1}{1},\overset{x_2}{4})\},\ n_2 \mapsto \{(\cancel{\overset{x_2}{1},\overset{x_4}{1}}), (\overset{x_2}{2},\overset{x_4}{2}), (\cancel{\overset{x_2}{3},\overset{x_4}{3}}), (\overset{x_2}{4},\overset{x_4}{4})\},\ n_3 \mapsto \{(\overset{x_2}{4},\overset{x_4}{4}), (\overset{x_2}{2},\overset{x_4}{3})\}\,.$$

On the other hand, it is important to perform message passing upwards first and then downwards second. Reversing this does not yield a canonical element. For instance going back to Example 7, if we first performed a downwards pass from n_2, the result is unchanged:

$$n_1 \mapsto \{(\overset{x_1}{1},\overset{x_2}{2}), (\overset{x_1}{3},\overset{x_2}{3}), (\overset{x_1}{1},\overset{x_2}{4})\},\ n_2 \mapsto \{(\overset{x_2}{1},\overset{x_4}{1}), (\overset{x_2}{2},\overset{x_4}{2}), (\overset{x_2}{3},\overset{x_4}{3}), (\overset{x_2}{4},\overset{x_4}{4})\},\ n_3 \mapsto \{(\overset{x_2}{4},\overset{x_4}{4}), (\overset{x_2}{2},\overset{x_4}{3})\}\,.$$

Performing an upwards pass now yields the element s_2:

$$n_1 \mapsto \{(\overset{x_1}{1},\overset{x_2}{2}), (\overset{x_1}{3},\overset{x_2}{3}), (\overset{x_1}{1},\overset{x_2}{4})\},\ n_2 \mapsto \{(\cancel{\overset{x_2}{1},\overset{x_4}{1}}), (\overset{x_2}{2},\overset{x_4}{2}), (\cancel{\overset{x_2}{3},\overset{x_4}{3}}), (\overset{x_2}{4},\overset{x_4}{4})\},\ n_3 \mapsto \{(\overset{x_2}{4},\overset{x_4}{4}), (\overset{x_2}{2},\overset{x_4}{3})\}\,.$$

However this is not canonical, since the element $(\overset{x_1}{3},\overset{x_2}{3})$ in $s_2(n_1)$ violates the requirement over the edge (n_1, n_2).

Let $\hat{s}$ be the resulting abstract object after the message passing procedure finishes.

Theorem 3. *The result of message passing $\hat{s}$ is a canonical object, and it satisfies $\gamma(\hat{s}) = \gamma(s)$.*

Proof (Sketch). First, we note that whenever a message is passed for an abstract value s from node m to n along an edge (m, n) resulting in a new abstract value s': **(P1)** $\gamma(s') = \gamma(s)$; and **(P2)** the projection of $s'(n)$ along the dimensions $\mathrm{CV}(m, n)$ is now contained in that of $s'(m)$ along $\mathrm{CV}(m, n)$. Furthermore, property **(P2)** remains unchanged regardless of any future messages that are passed along the tree edges.

Next, it is shown that after each upwards pass, when a message is passed, property **(P2)** (stated above) holds for each node m and its parent node n since a message is passed from m to n. During the downwards pass, property **(P2)** holds for each node n and its child node m in the tree. Combining the two, we note that for each edge (m, n) in the tree, we have property **(P2)** in either direction guaranteeing that $\mathsf{proj}(s^*(m), \mathrm{CV}(m, n)) = \mathsf{proj}(s^*(n), \mathrm{CV}(m, n))$, for the final result s^*, or in other words that s^* is canonical.

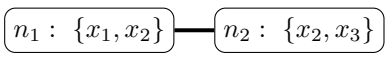
3.3 Decomposable Sets and Post-conditions

We have already noted that for any concrete set over $S \subseteq D$, the process of abstracting it by projecting into nodes of a tree T, and re-concretizing it is "lossy": I.e, $S \subseteq \gamma(\alpha(S))$. In this section, we study "tree decomposable" concrete sets S for which $\gamma(\alpha(S)) = S$. Ideally, we would like to prove that if a set S is tree decomposable then so is the set $\mathsf{post}(S, \Pi)$ of next states. However, we will disprove this by showing a counterexample. Nevertheless, we will present an analysis of why this fact fails and suggest approaches that can "manage" this loss in precision.

Definition 11 (Decomposable Sets). *We say that a set S is tree decomposable given a tree T iff $\gamma(\alpha(S)) = S$.*

This is in fact a "global" definition of decomposability. In fact, a nice "local" definition can be provided that is reminiscent of the notion of conditional independence in graphical models. We will defer this discussion to an extended version of this paper due to space limitations.

Example 10. Consider set $S : \{(\overset{x_1}{1}, \overset{x_2}{2}, \overset{x_3}{1}), (\overset{x_1}{2}, \overset{x_2}{2}, \overset{x_2}{2})\}$ and tree T below:



We wish to check if S is T-decomposable. We have $s : \alpha(S)$ as

$$s(n_1) : \mathsf{proj}(S, n_1) : \{(\overset{x_1}{1}, \overset{x_2}{2}), (\overset{x_1}{2}, \overset{x_2}{2})\}\ s(n_2) : \mathsf{proj}(S, n_2)\{(\overset{x_2}{2}, \overset{x_3}{1}), (\overset{x_2}{2}, \overset{x_3}{2})\}\,.$$

Now, $\gamma(s)$:$\{(\overset{x_1}{1}, \overset{x_2}{2}, \overset{x_3}{1}), (\overset{x_1}{1}, \overset{x_2}{2}, \overset{x_2}{2}), (\overset{x_1}{2}, \overset{x_2}{2}, \overset{x_3}{1}), (\overset{x_2}{2}, \overset{x_2}{2}, \overset{x_2}{2})\,.\}$. We note that the set S is not tree decomposable. On the other hand, one can verify that the set S_1:$\{(\overset{x_1}{1}, \overset{x_2}{2}, \overset{x_3}{2}), (\overset{x_1}{2}, \overset{x_2}{2}, \overset{x_2}{2})\}$ is tree decomposable.

The following lemma will be quite useful.

Lemma 4. *Let S_1, S_2 be tree decomposable sets over T. Their intersection is tree decomposable.*

Let Π be a transition system over system variables in $\mathbf{x} \in D$. For a given set $S \subseteq D$, us define the post-condition $\mathsf{post}(S, \Pi)$ to be the set of states reachable in one step starting from some state in S:

$$\mathsf{post}(S, \Pi) : \{\mathbf{x}' \mid \mathbf{x} \in S,\ \mathbf{x}' = \mathsf{eval}(f, \mathbf{x})\}\,.$$

Let us also consider a transition relation R over pairs of states $(\mathbf{x}, \mathbf{x}') \in D \otimes D$:

$$R = \{(\mathbf{x}, \mathbf{x}') \mid \mathbf{x}, \mathbf{x}' \in D \text{ and } \mathbf{x}' = \mathsf{eval}(f, \mathbf{x})\}\,.$$

The relation R can be viewed as the intersection of n relations: $R : \bigcap_{x_j \in X} R_j$, wherein

$$R_j : \{(\mathbf{x}, \mathbf{x}') \mid \mathbf{x}, \mathbf{x}' \in D \text{ and } \mathbf{x}'_j = \mathsf{eval}(f_j, \mathbf{x})\}\,.$$

In other words, R_j is a component of R that models the update of the system variable x_j. Also for each $x_j \in X$, let $e_j : \mathsf{inps}(f_j) \cup x_j$ be the inputs to the update function f_j and the node x_j itself.

Given the tree T, we define the extended tree T' as having the same node set N and edge set C as T. However, $\textsc{verts}_{T'}(n) = \textsc{verts}_T(n) \cup \{x'_j \mid x_j \in \textsc{verts}_T(n)\}$. Note that T' with the labeling $\textsc{verts}_{T'}$ satisfies all the condition of a tree decomposition for a graph G save the addition of vertices x'_i in each node of the tree. We will write $\textsc{verts}'(n)$ to denote the set $\textsc{verts}_{T'}(n)$.

Lemma 5. *The transition relation R of a system Π is tree T' decomposable.*

The proof is provided in the extended version and is done by writing R as an intersection of tree decomposable relations R_j, and appealing to Lemma 4.

First, we show the negative result that the image of a tree (T) decomposable set under a tree (T') decomposable transition relation is not tree decomposable, in general.

Example 11. Let $X = \{x_1, x_2, x_3\}$ and consider again the tree decomposition from Example 10. Let S be the set $\{(\overset{x_1}{*}, \overset{x_2}{*}, \overset{x_3}{*})\}$, wherein we use the wild card character as notation that can be substituted for any element in the set $\{1, 2\}$. Therefore, we take S to be a set with 8 elements. Clearly S is tree decomposable in the tree T from Example 10.

Consider the transition relation R that will be written as the intersection of three transition relations:

$$R_1 : \{(X, X') \mid x'_1 = x_2\},\ R_2 : \{(X, X') \mid x'_2 \in \{1, 2\}\},\ R'_3 : \{(X, X') \mid x'_3 = x_2\}.$$

Clearly R is tree T' decomposable. We can now compute the post-condition of S under this relation. The reader can verify the post-condition $\hat{S} : \{(\overset{x_1}{1}, \overset{x_2}{*}, \overset{x_3}{1}), (\overset{x_1}{2}, \overset{x_2}{*}, \overset{x_3}{2})\}$. However, $\hat{S}$ is not tree decomposable. We note that $\hat{s} : \alpha(\hat{S})$ is the set $\hat{s}(n_1) : \{(\overset{x_1}{*}, \overset{x_2}{*})\}$ and $\hat{s}(n_2) : \{(\overset{x_1}{*}, \overset{x_2}{*})\}$. Therefore $\gamma(\hat{s})$ is the set $\{(\overset{x_1}{*}, \overset{x_2}{*}, \overset{x_3}{*})\}$.

As noted above, the set R is tree T' decomposable. If S is tree decomposable, we can extend S to a set $S' : \mathsf{ext}_{X'}(S)$ that is now defined over $X \cup X'$ and is also tree decomposable. As a result $S' \cap R$ is also tree decomposable. However, the postcondition of S is the set $\mathsf{proj}(S' \cap R, X')$. Thus, the key operation that failed was the projection operation involved in computing the post-condition. This suggests a possible solution to this issue albeit an expensive one: at each step, we maintain the reachable states using both current and next state variables, thus avoiding projection. In effect, the reachable states at the i^{th} step will be entire trajectories of the system expressed over variables $X_0 \cup X_1 \cup \cdots X_i$. This is clearly not practical. However, a more efficient solution is to note that some of the current state variables can be projected out without losing the tree decomposability property. Going back to Example 11, we note that we can safely project away $\{x_1, x_3\}$, while maintaining the new reachable set in terms of (x_2, x'_1, x'_2, x'_3). In this way, we may recover the lost precision back.

In conclusion, we note that tree decompositions may lose precision over post-conditions. However, the loss in precision can be avoided if carefully selected "previous state variables" are maintained as the computation proceeds. The question of how to optimally maintain this information will be investigated in the future.

4 Grid-Based Interval Analysis

We now combine the ideas to create a disjunctive interval analysis using tree decompositions. The main idea here is to apply tree decompositions not to the concrete set of states but to an abstraction of the concrete domain by grid-based intervals.

We will now describe the interval-based abstraction of sets of states dynamical system Π in order to perform over-approximate reachability analysis. Let us fix a system $\Pi : \langle \mathbf{x}, \mathbf{w}, D, W, f, X_0, U \rangle$ as defined in Definition 1. We will assume that the domain of state variables D is a hyper-rectangle given by $D : [L(x_1), U(x_1)] \times \cdots \times [L(x_n), U(x_n)]$ for $L(x_j), U(x_j) \in \mathbb{R}$ and $L(x_j) \leq U(x_j)$ for each $j = 1, \ldots, n$. In other words, each system variable x_j lies inside the interval $[L(x_j), U(x_j)]$. Likewise, we will assume that $W : \prod_{k=1}^{m} [L(w_k), U(w_k)]$ such that $L(w_k) \leq U(w_k)$ and $L(w_k), U(w_k) \in \mathbb{R}$.

We will consider a *uniform* cell decomposition wherein each dimension is divided into some natural number $M > 0$ of equal sized subintervals. The i^{th} *subinterval* of variable x_j is denoted as $\mathsf{subInt}(x_j, i)$, and is given by $[L(x_j) + i\delta_j, L(x_j) + (i+1)\delta_j]$ for $i = 0, \ldots, M-1$ and $\delta_j : \frac{(U(x_j) - L(x_j))}{M}$. Similarly, we will define $\mathsf{subInt}(w_k, i)$ for disturbance variables w_k whose domains are also divided into M subdivisions. The overall domain $D \times \mathcal{W}$ is therefore divided into M^{m+n} cells wherein each cell is indexed by a tuple of natural numbers $\mathbf{i} : \langle i_1, \ldots, i_n, i_{n+1}, \ldots, i_{n+m} \rangle$, such that $i_j \in \{0, \ldots, M-1\}$ and the cell corresponding to $\mathbf{i}$ is given by:

$$\gamma_C(\mathbf{i}) : \prod_{j=1}^{n} \mathsf{subInt}(x_j, \mathbf{i}_j) \times \prod_{k=1}^{m} \mathsf{subInt}(w_k, \mathbf{i}_{n+k}) \tag{1}$$

Definition 12 (Grid-Based Abstract Domain). *The grid based abstract domain is defined by the set* $\mathcal{C} : \mathcal{P}(\mathbf{i} \in \{0, \ldots, M\}^{m+n})$*, wherein each abstract domain element is a set of grid cells. The sets are ordered simply by set inclusion* $\subseteq$ *between sets of grid cells. The abstraction map* $\alpha_C : \mathcal{P}(D) \to \mathcal{C}$ *is defined as follows:*

$$\alpha_C(S) : \{\mathbf{i} \in \mathcal{C} \mid \gamma_C(\mathbf{i}) \cap S \neq \emptyset\}.$$

The concretization map γ_C *is defined above in* (1)*.*

Definition 13 (Interval Propagator). *An interval propagator (IP) is a higher order function that takes in the description of a function* f *with* k *real-valued inputs and* p *real valued outputs, and an interval* $I : [l_1, u_1] \times \cdots \times [l_k, u_k]$

and outputs an interval (hyperrectangle over $\mathbb{R}^p$*)* $\textsc{IntvlProp}(f, I)$ *such that the following soundness guarantees hold:*

$$(\forall \mathbf{x} \in D) \bigwedge_{j=1}^{k} \mathbf{x}_j \in [l_j, u_j] \;\Rightarrow\; \mathsf{eval}(f, \mathbf{x}) \in \textsc{IntvlProp}(f, I)\,.$$

In practice, interval arithmetic approaches have been used to build sound interval propagators [33]. However, they suffer from issues such as the *wrapping effect* that make their outputs too conservative. This can be remedied by either (a) performing a finer subdivision of the inputs (i.e, increasing M) to ensure that the intervals I being input into the IntvlProp are sufficiently small to guarantee tight error bounds; or (b) using higher order arithmetics such as affine arithmetic or Taylor polynomial arithmetic [25,32].

The interval propagator serves to define an abstract post-condition operation over sets of cells $\hat{S} \subseteq \mathcal{C}$. Given such a set, $\hat{S}$, we compute the post condition in the abstract domain. Informally, the post condition is given (a) by iterating over each cell in S; and (b) computing the possible next cells using IntvlProp. Formally, we define the abstract post operation as follows:

$$\mathsf{post}_C(\hat{S}, \Pi) : \bigcup_{\mathbf{i} \in \hat{S}} \alpha_C(\textsc{IntvlProp}(f, \gamma_C(\mathbf{i})))\,.$$

Given this machinery, an abstract T-step reachability analysis is performed in the standard manner: (a) abstract the initial state; (b) compute post condition for T steps; and (c) check for intersections of the abstract states with the abstraction of the unsafe set. We can also define and use widening operators to make the sequence of iterates converge. The grid based abstract domain can offer some guarantees with respect to the quality of the abstraction. For instance, we can easily bound the Hausdorff distance between the underlying concrete set and the abstraction as a function of the discretization sizes δ_j. However, the desirable properties come at a high computational cost since the number of cells grows exponentially in the number of system and disturbance variables.

4.1 Tree Decomposed Analysis

We now consider a tree-decomposed approach based on the concept of nodal abstractions. The key idea here is to perform the grid-based abstraction not on the full set of system and disturbance variables, but instead on individual *nodal* abstractions over a tree decomposition T.

Definition 14 (Nodal Abstractions). *A nodal abstraction* $\textsc{Nodal Abstraction}(\Pi, n)$ *corresponding to a node* $n \in N$ *is defined as follows*

1. *The set of system variables are given by* $X_n :$ $\textsc{verts}_X(n)$ *with domain given by* $D_n :$ $\mathsf{proj}(D, X_n)$.
2. *The initial states are given by* $\mathsf{proj}(X_0, X_n)$.

3. *The unsafe set is given by* $\mathsf{proj}(U, X_n)$.
4. *The set of disturbance variables are* Y_n : $\textsc{verts}_W(n)$ *with domain given by* W_n : $\mathsf{proj}(W, W_n)$.
5. *The updates are described by a* relation $R(X_n, X'_n)$ *that relate the possible current states* X_n *and next states* X'_n. *The relation is constructed as a conjunction of assertions over variables* x_i, x'_i *wherein* $x_i \in X_n$.
   (a) *If the update* $x_i := f_i(\mathbf{x}, \mathbf{w})$ *is associated with the node* n, *we add the conjunct* $x'_i = f_i(X_n, W_n)$, *noting that the proper inputs to* f_i *are contained in* $\textsc{verts}(n)$.
   (b) *Otherwise,* $x'_i \in \mathsf{proj}(D, \{x_i\})$ *that simply states that the next state value of the variable* x_i *is some value in its domain.*

Given a system Π, the nodal abstraction is a conservative abstraction, and therefore, it preserves reachability properties.

Lemma 6. *For any reachable state* $\mathbf{x}$ *of* Π *at time* t, *its projection* $\mathsf{proj}(\mathbf{x}, X_n)$ *is a reachable state of* $\textsc{NodalAbstraction}(\Pi, n)$ *at time* t.

Since each nodal abstraction involves at most $\omega+1$ variables, the abstraction at each node can involve at most $M^{\omega+1}$ cells where ω is the tree width. Also, note that a tree decomposition can be found with tree width ω that has at most $|X| + |W|$ nodes. This implies that the number of nodal abstractions can be bounded by $(|X| + |W|)$.

Let $\Pi(n)$: $\textsc{NodalAbstraction}(\Pi, n)$ be the nodal abstraction for tree node $n \in N$. For each node $n \in N$, we instantiate a grid based abstract domain for $\Pi(n)$ ranging over the variables $\textsc{verts}_X(n)$. At the i^{th} step of the reachability analysis, we maintain a map s_i each node n to a set of grid cells $s_i(n)$ defined over $\textsc{verts}(n)$.

1. Compute $\hat{s_i}(n) : \mathsf{post}_C(s_i(n), \Pi(n))$.
2. Make $\hat{s_i}$ canonical using message passing between nodes to obtain s_{i+1}.

The message passing is performed not over projections of concrete states but over cells belonging to the grid based abstract domain. Nevertheless, we can easily extend the soundness guarantees in Theorem 3 to conclude soundness of the composition.

Once again, we can stop this process after T steps or use widening to force convergence. We now remark on a few technicalities that arise due to the way the tree decomposition is constructed.

Intersections with Unsafe Sets: Checking for a non-empty intersection with the unsafe sets may require constructing concrete cells over the full dimensional space if the unsafe sets are not tree decomposable for the tree T. However in many cases, the unsafe states are specified as intervals over individual variables, which yields a tree decomposable set. In such cases, we need to intersect the abstraction at each node with the unsafe set and perform message passing to make it canonical before checking for emptiness.

Handling Guards and Invariants: We have not discussed guards and invariants. It is assumed that such guards and invariants are tree decomposable over the tree T. In this case, we can check which abstract cells have a non-empty intersection with the guard using message passing. The handling of transition systems with guards and invariants will be discussed as part of future extensions.

5 Experimental Evaluation

In this section, we describe an experimental evaluation of our approach over a set of benchmark problems. Our evaluation is based on a C++-based prototype implementation that can read in the description of a nonlinear dynamical system over a set of system and disturbance variables. The dynamics can currently include polynomials, rational functions and trigonometric functions. Our implementation uses the MPFI library to perform interval arithmetic over the grid cells [36]. We use the HTD library to compute tree decompositions [1]. The system then computes a time-bounded reachable set over the first T steps of the system's execution. Currently, we plot the results and compare the reachable set estimates against simulation data. We also compare the reachable sets computed by the tree decomposition approach against an approach without using tree decompositions. However, we note that the latter approach timed out on systems beyond 4 state variables.

Table 1 presents the results over a small set of challenging nonlinear systems benchmarks along with a comparison to two other approaches (a) the approach without tree decomposition and (b) the tool SAPO [22] which computes time bounded reachable sets for polynomial systems using the technique of parallelotope bundles described by Dreossi et al. [23]. The benchmarks range in number of system variables from 3 to 20 state variables. We describe the sources for each benchmark where appropriate. Note that the SAPO tool does not handle nonpolynomial dynamics or time varying disturbances at the time of writing.

The treewidths range from 1 for the simplest system (Example 1) to 3 for the 7-state Laub Loomis oscillator example [30]. We note that the tree decomposition was constructed within 0.01 s for all the examples. We also note that systems with as many as 20 state variables are handled by our approach whereas the monolithic approach cannot handle systems beyond 4 state variables. We now compare the results of our approach to that of the monolithic approach on the two cases where the latter approach completed.

System # 1: Consider again the system from Example 1 with 3 state variables and 1 disturbance. We have already noted a tree decomposition of tree width 1 for this example.

System # 2: In this example, we consider a system over 4 state variables $\{x, y, z, w\}$ and one disturbance variable w_1.

$$x := 0.5x + y + 0.05xy - w_1,\ y := -0.7y - 0.03x,\ z := z - 0.4y,$$
$$w := w - 0.05xw$$

Table 1. Results on benchmark examples. $|X|$: Number of state variables, $|W|$: number of disturbance variables, Tree Decomp.: reachability using tree decompositions, Monolithic: reachability analysis without tree decompositions. SAPO: number of directions ($|L|$), number of bundles ($|T|$) and running time. All timings are reported in seconds on a Macbook pro laptop running MacOS 10.14 with 16 GB RAM and 3.4 GHz Intel core i7 processor. Reachability analysis was carried out for 15 time steps.

Name	$\|X\|$	$\|W\|$	Tree Width	Tree Decomp.		Monolithic		SAPO	
				Time	# Cells	Time	# Cells	($\|L\|, \|T\|$)	Time
System # 1	3	1	1	14.4	0.22M	1047.6	7.6M	-n/a-	
System # 2	4	1	2	2.7	24K	652	3.1M	-n/a-	
SIR [23,40]	3	0	1	4.1	95K	143	2M	(3,1)	0.1
1D-Lattice-10 [39]	10	0	2	99	1.1M	TO (1.5 h)		(16,6)	679
Ebola-epidemic [14]	5	0	2	799.4	1.9M	TO (1.5 h)		(5,5)	0.02
p53-gene-reg [31]	6	0	2	135.8	98K	TO (1.5 h)		-n/a-	
Influenza-epidemic [22]	4	0	2	517.9	1.4M	TO (1.5 h)		(7,4)	0.1
Coupled-vanderpol	6	0	2	10.5	0.1M	TO (1.5 h)		(10,5)	2.5
Laub-Loomis [20,30]	7	0	3	1755.1	2.6M	TO (1.5 h)		(12,6)	1.8
Honeybee* [9,23]	6	4	3	206.1	2.1M	TO (1.5 h)		(8,4)	0.7
Phosporelay [22]	7	0	3	1566.2	7.5M	TO (1.5 h)		(10,4)	1.2
Coord. Vehicles (1)	5	1	2	150.2	0.5M	TO (1.5 h)		-n/a-	
Coord. Vehicles (2)	10	2	2	1175.2	2M	TO (1.5 h)		-n/a-	
Coord. Vehicles (4)	20	4	2	2206.7	3.9M	TO (1.5 h)		-n/a-	

The domains include $(x, y, z, w) \in [-1, 1]^4$ and divided into 16×10^8 grid cells (200 for each state variable). The disturbance $w_1 \in [-0.1, 0.1]$. The initial conditions are $x \in [0.08, 0.16], y \in [-0.16, -.05], z \in [0.12, 0, 31]$ and $w \in [-0.15, -0.1]$. We obtain a tree decomposition of width 2, wherein the nodes include $n_1 : \{x, y, w_1\}$, $n_2 : \{y, z\}$ and $n_3 : \{x, w\}$ with the edges (n_1, n_2) and (n_1, n_3).

Figure 1 compares the resulting reachable sets for the tree decomposed reachability analysis versus the monolithic approach. We note differences between the two reachable sets but the loss in precision is not significant.

Coordinated Vehicles: In this example, we study nonlinear vehicle models of vehicles executing coordinated turns. Each vehicle has states $(x_i, y_i, v_{x,i}, v_{y,i}, \omega)$, representing positions, velocities and the rate of change in the yaw angle, respectively, with a disturbance w_i. The dynamics are given by

$$x_i := x_i + 0.1v_{x,i},\ y_i := y_i + 0.1v_{y,i},\ v_{x,i} = v_{x,i} + 0.1v_{x,i}\cos(0.1\omega_i) - 0.1v_{y,i}\sin(0.1\omega_i)\omega_i = 0.5\omega_i + 0.5\omega_0 + 0.1w_i$$

The vehicles are loosely coupled with ω_i representing the turn rate of the i^{th} vehicle and ω_0 that of the "lead" vehicle. The i^{th} vehicle tries to gradually

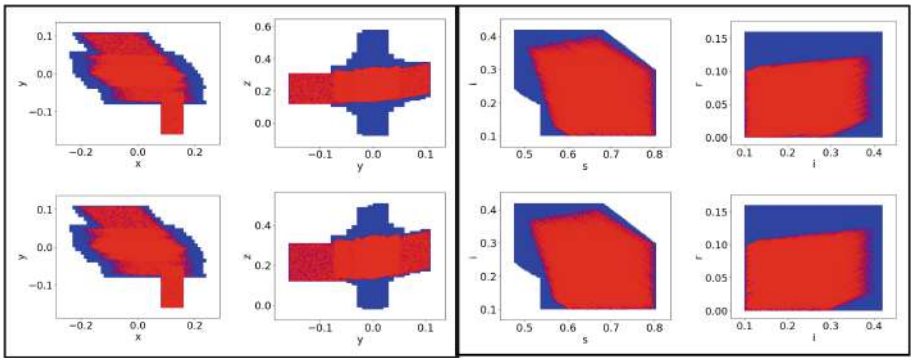


Fig. 1. Reachable set projections (shaded blue) for System# 2 (left) and the SIR model [22] (right). Top: tree decomposition approach and Bottom: monolithic approach without tree decompositions. Reachable sets are identical for the SIR model. Note the difference in range of z for the system #2. The red dots show the results of simulations. (Color figure online)

align its turn rate to that of the lead vehicle. This model represents a simple scenario of loosely coupled systems that interact using a small set of state variables. Applications including models of cardiac cells that are also loosely coupled through shared action potentials [26]. The variables x_i, y_i are set in the domain $[-15, 15]$ and subdivided into 300 parts along each dimension. Similarly, the velocities range over $[-10, 10]$ and are subdivided into 500 parts each and the yaw rate ranges over $[-0.2, 0.2]$ radians/sec and subdivided into 25 parts. The disturbance ranges over $[-0.1, 0.1]$. Table 1 reports results from models involving $1, 2$ and 4 vehicles. Since they are loosely coupled, the treewidth of these models is 2.

Laub-Loomis Model: The Laub-Loomis model is a molecular network that produces spontaneous oscillations for certain values of the model parameters. The model's description was taken from Dang et al. [20]. The system has 7 state variables each of which was subdivided into 100 cells yielding a large state space with 10^{14} cells. We note that the tree width of the graph is 3, yielding nodes with upto 4 variables in them.

Comparison with SAPO. SAPO is a state-of-the-art tool that uses polytope bundles and Bernstein polynomials to represent and propagate reachable sets for polynomial dynamical systems [22,23]. We compare our approach directly on SAPO for identical models and initial sets. Note that SAPO does not currently handle non-polynomial models or models with time-varying disturbances. Table 1 shows that SAPO is orders of magnitude faster on all the models, with the sole exception of the 1D-Lattice-10 model. Figure 2 shows the comparison of the reachable sets computed by our approach (shaded blue region) against those

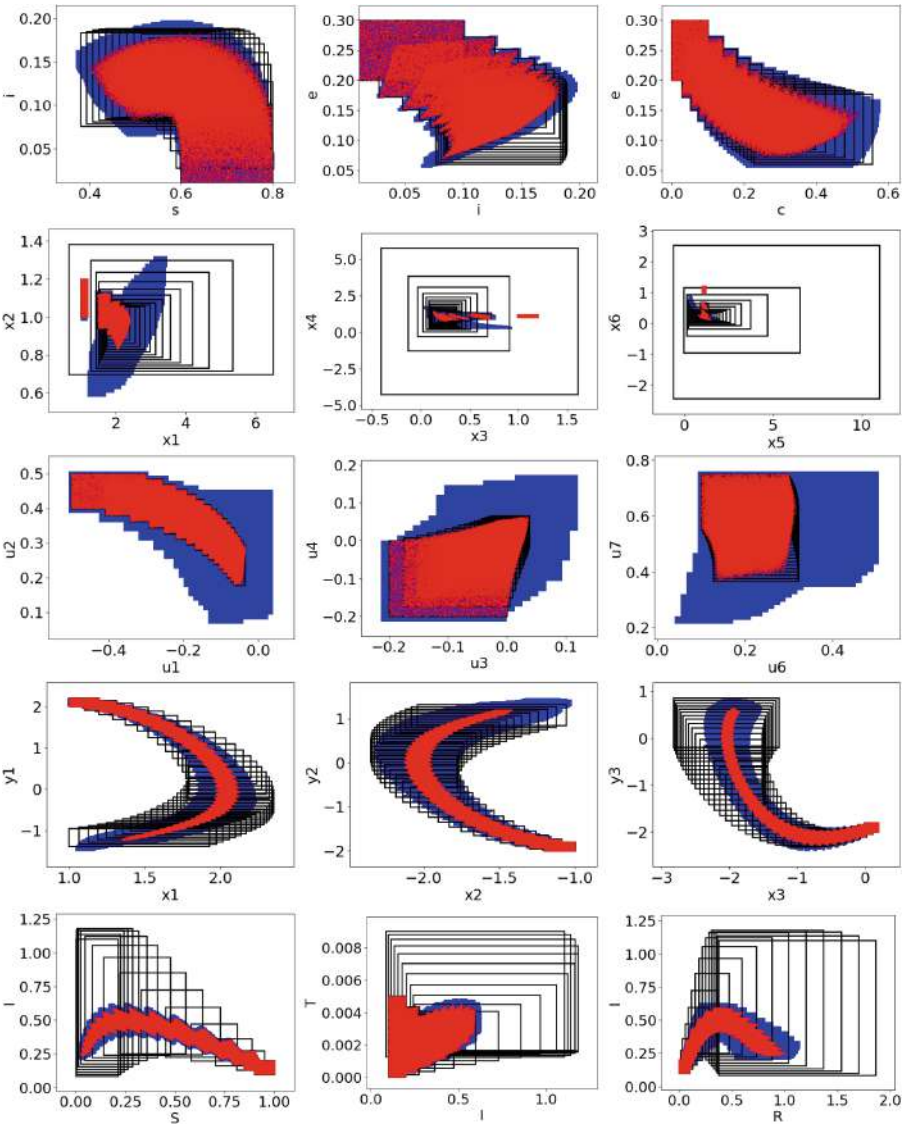


Fig. 2. Comparison of various projections of the reachable sets computed by our approach shown in blue, the reachable set computed by SAPO shown as black rectangles and states obtained through random simulation shown in red dots. Top row: ebola model, second row: phosporelay, third row: 1d-lattice-10, fourth row: vanderpol (35 steps) and bottom row: influenza model. (Color figure online)

computed by SAPO (black rectangles) for five different models. We note that for three of the models compared, neither reachable set is contained in the other. For the one dimensional lattice model, SAPO produces a better reachable set,

whereas our approach is better for the influenza model. We also note that both for our approach the precision can be improved markedly by increasing the number of subdivisions, albeit at a large computational cost that depends on the treewidth of the model. The same is true for SAPO, where the number of directions and the template sizes have a non-trivial impact on running time.

Models with Large Treewidths. We briefly report on a few models that we attempted with large treewidths. For such models, our approach of decomposing the space into cells becomes infeasible due to the curse of dimensionality.

A model of how honeybees select between different sites [9,23] has 6 variables and its tree width is 5 with a single tree node containing all state variables. However, the large treewidth is due to two terms in the model which are replaced by disturbance variables that overapproximate their value. This brings down the treewidth to 3, making it tractable for our approach. Details of this transformation are discussed in our extended version. Treewidth reduction using abstractions is an interesting topic for future work.

We originally proposed to analyze a 2D grid lattice model taken from Vleck et al [39]. However, a 2D 10×10 lattice model has a dependency hypergraph that forms a 10×10 grid with treewidth 10. Likewise, the 17-state crazyflie benchmark for SAPO [22] could not be analyzed by our approach since its treewidth is too large.

6 Conclusions

We have shown how tree decompositions can define an abstract domain that projects concrete sets along the various subsets of state variables. We showed how message passing can be used to exchange information between these subsets. We analyze the completeness of our approach and show that the abstraction is lossy due to the projection operation. We show that for small tree width models, a gridding-based analysis of nonlinear system can be used whereas such approaches are too expensive when applied in a monolithic fashion. For the future, we plan to study tree decompositions for abstract domains such as disjunctions of polyhedra, parallelotope bundles and Taylor models. The process of model abstraction to reduce treewidth is another interesting future possibility.

Acknowledgments. This work was supported by US NSF under award number CPS 1836900, CCF 1815983 and the US Air Force Research Laboratory (AFRL). The author acknowledges Profs. Mohamed Amin Ben Sassi and Fabio Somenzi for helpful discussions, and the anonymous reviewers for their comments.

References

1. Abseher, M., Musliu, N., Woltran, S.: htd – a free, open-source framework for (customized) tree decompositions and beyond. In: Salvagnin, D., Lombardi, M. (eds.) CPAIOR 2017. LNCS, vol. 10335, pp. 376–386. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59776-8_30

2. Adjé, A., Gaubert, S., Goubault, E.: Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 23–42. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_3
3. Blanchet, B., et al.: A static analyzer for large safety-critical software. In: Programming Language Design & Implementation, pp. 196–207. ACM Press (2003)
4. Blanchet, B., et al.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 85–108. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36377-7_5
5. Bodlaender, H.L.: Dynamic programming on graphs with bounded treewidth. In: Lepistö, T., Salomaa, A. (eds.) ICALP 1988. LNCS, vol. 317, pp. 105–118. Springer, Heidelberg (1988). https://doi.org/10.1007/3-540-19488-6_110
6. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM J. Comput. **25**(6), 1305–1317 (1996)
7. Bodlaender, H.L.: Fixed-parameter tractability of treewidth and pathwidth. In: Bodlaender, H.L., Downey, R., Fomin, F.V., Marx, D. (eds.) The Multivariate Algorithmic Revolution and Beyond. LNCS, vol. 7370, pp. 196–227. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30891-8_12
8. Bodlaender, H.L., Koster, A.M.: Treewidth computations I. Upper bounds. Inf. Comput. **208**(3), 259–275 (2010)
9. Britton, N.F., Franks, N.R., Pratt, S.C., Seeley, T.D.: Deciding on a new home: how do honeybees agree? Proc. R. Soc. Lond. Ser. B Biol. Sci. **269**(1498), 1383–1388 (2002)
10. Chatterjee, K., Ibsen-Jensen, R., Goharshady, A.K., Pavlogiannis, A.: Algorithms for algebraic path properties in concurrent systems of constant treewidth components. ACM Trans. Program. Lang. Syst. **40**(3), 1–43 (2018)
11. Chatterjee, K., Ibsen-Jensen, R., Pavlogiannis, A., Goyal, P.: Faster algorithms for algebraic path properties in recursive state machines with constant treewidth. In: Principles of Programming Languages (POPL), pp. 97–109. Association for Computing Machinery, New York (2015)
12. Chen, M., Herbert, S., Tomlin, C.: Exact and efficient Hamilton-Jacobi-based guaranteed safety analysis via system decomposition. In: IEEE International Conference on Robotics and Automation (ICRA) (2017, to appear). arXiv:1609.05248
13. Chen, X., Sankaranarayanan, S.: Decomposed reachability analysis for nonlinear systems. In: 2016 IEEE Real-Time Systems Symposium (RTSS), pp. 13–24, November 2016
14. Chowell, G., Hengartner, N., Castillo-Chavez, C., Fenimore, P., Hyman, J.: The basic reproductive number of Ebola and the effects of public health measures: the cases of Congo and Uganda. J. Theor. Biol. **229**(1), 119–126 (2004)
15. Courcelle, B.: The monadic second-order logic of graphs iii: treewidth, forbidden minors and complexity issues. Informatique Théorique **26**, 257–286 (1992)
16. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proceedings of the ISOP 1976, pp. 106–130. Dunod, Paris (1976)
17. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55844-6_142

18. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: ACM Principles of Programming Languages, pp. 238–252 (1977)
19. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among the variables of a program. In: POPL 1978, pp. 84–97, January 1978
20. Dang, T., Dreossi, T.: Falsifying oscillation properties of parametric biological models. In: Hybrid Systems Biology (HSB). EPTCS, vol. 125, pp. 53–67 (2013)
21. Delmas, D., Souyris, J.: Astrée: from research to industry. In: Nielson, H.R., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 437–451. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-2_27
22. Dreossi, T.: Sapo: reachability computation and parameter synthesis of polynomial dynamical systems. In: Hybrid Systems: Computation and Control (HSCC), pp. 29–34. ACM (2017)
23. Dreossi, T., Dang, T., Piazza, C.: Parallelotope bundles for polynomial reachability. In: Hybrid Systems: Computation and Control (HSCC), pp. 297–306. ACM (2016)
24. Ferrara, A., Pan, G., Vardi, M.Y.: Treewidth in verification: local vs. global. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 489–503. Springer, Heidelberg (2005). https://doi.org/10.1007/11591191_34
25. de Figueiredo, L.H., Stolfi, J.: Self-validated numerical methods and applications. In: Brazilian Mathematics Colloquium Monograph. IMPA, Rio de Janeiro (1997)
26. Grosu, R., et al.: From cardiac cells to genetic regulatory networks. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 396–411. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_31
27. Gulwani, S., Jojic, N.: Program verification as probabilistic inference. In: POPL, POPL 2007, pp. 277–289. Association for Computing Machinery (2007)
28. Ivančić, F., et al.: Scalable and scope-bounded software verification in VARVEL. Autom. Softw. Eng. **22**(4), 517–559 (2014). https://doi.org/10.1007/s10515-014-0164-0
29. Koller, D., Friedman, N.: Probabilistic Graphical Models. The MIT Press, Cambridge (2009)
30. Laub, M.T., Loomis, W.F.: A molecular network that produces spontaneous oscillations in excitable cells of dictyostelium. Mol. Biol. Cell **9**(12), 3521–3532 (1998)
31. Leenders, G., Tuszynski, J.A.: Stochastic and deterministic models of cellular p53 regulation. Front. Oncol. **3**, 64 (2013)
32. Makino, K., Berz, M.: Taylor models and other validated functional inclusion methods. Int. J. Pure Appl. Math. **4**(4), 379–456 (2003)
33. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. SIAM, Philadelphia (2009)
34. Nielson, F., Nielson, H.R., Hankin, C.: Algorithms. In: Nielson, F., Nielson, H.R., Hankin, C. (eds.) Principles of Program Analysis. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-662-03811-6_6
35. Obdržálek, J.: Fast Mu-Calculus model checking when tree-width is bounded. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 80–92. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_7
36. Revol, N., Rouillier, F.: Motivations for an arbitrary precision interval arithmetic and the MPFI library. Reliable Comput. **11**, 275–290 (2005). https://doi.org/10.1007/s11155-005-6891-y
37. Robertson, N., Seymour, P.: Graph minors. III. Planar tree-width. J. Comb. Theory Ser. B **36**(1), 49–64 (1984)
38. Thorup, M.: All structured programs have small tree width and good register allocation. Inf. Comput. **142**(2), 159–181 (1998)

39. Vleck, E.S.V., Mallet-Paret, J., Cahn, J.W.: Traveling wave solutions for systems of ODEs on a two-dimensional spatial lattice. SIAM J. Appl. Math. **59**, 455–493 (1998)
40. Weisstein, E.W.: SIR model, from MathWorld-A Wolfram Web Resource. https://mathworld.wolfram.com/SIRModel.html. Accessed May 2020

Fast and Guaranteed Safe Controller Synthesis for Nonlinear Vehicle Models

Chuchu Fan[1(✉)], Kristina Miller[2], and Sayan Mitra[2]

[1] Department of Computing and Mathematical Sciences, California Institute of Technology, Pasadena, USA
chuchu@caltech.edu

[2] Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Champaign, USA
{kmmille2,mitras}@illinois.edu

Abstract. We address the problem of synthesizing a controller for nonlinear systems with reach-avoid requirements. Our controller consists of a reference controller and a tracking controller which drives the actual trajectory to follow the reference trajectory. We identify a type of reference trajectory such that the tracking error between the actual trajectory of the closed-loop system and the reference trajectory can be bounded. Moreover, such a bound on the tracking error is independent of the reference trajectory. Using such bounds on the tracking error, we propose a method that can find a reference trajectory by solving a satisfiability problem over linear constraints. Our overall algorithm guarantees that the resulting controller can make sure every trajectory from the initial set of the system satisfies the given reach-avoid requirement. We also implement our technique in a tool FACTEST. We show that FACTEST can find controllers for four vehicle models (3–6 dimensional state space and 2–4 dimensional input space) across eight scenarios (with up to 22 obstacles), all with running time at the sub-second range.

1 Introduction

Design automation and safety of autonomous systems is an important research area. Controller synthesis aims to provide correct-by-construction controllers that can guarantee that the system under control meets certain requirements. Controller synthesis is a type of program synthesis problem. The synthesized program or *controller* g has to meet the given requirement R, when it is run in

The authors acknowledge support from the DARPA Assured Autonomy under contract FA8750-19-C-0089, the Air Force Office of Scientific Research under grant AFOSR FA9550-17-1-0236, and the National Science Foundation under grant NSF CCF 1918531. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 629–652, 2020.
https://doi.org/10.1007/978-3-030-53288-8_31

(closed-loop) composition with a given physical process or *plant* $\mathcal{A}$. Therefore, a synthesis algorithm has to account for the combined behavior of g and $\mathcal{A}$.

Methods for designing controllers for asymptotic requirements like stability, robustness, and tracking, predate the algorithmic synthesis approaches for programs [3,16,30]. However, these classic control design methods normally do not provide formal guarantees in terms of handling bounded-horizon requirements like safety. Typical controller programs are small, well-structured, and at core, have a succinct logic ("bang-bang" control) or mathematical operations (PID control). This might suggest that controllers could be an attractive target for algorithmic synthesis for safety, temporal logic (TL), and bounded time requirements [1,9,18,34,38].

On the other hand, *motion planning (MP),* which is an instance of the controller synthesis for robots is notoriously difficult (see [21] Chapter 6.5). A typical MP requirement is to make a robot $\mathcal{A}$ track certain waypoints while meeting some constraints. A popular paradigm in MP, called sampling-based MP, gives practical, *fully automatic,* randomized, solutions to hard problem instances by only considering the geometry of the vehicle and the free space [14,15,20,21]. However, they do not ensure that the dynamic behavior of the vehicle will actually follow the planed path without running into obstacles. Ergo, MP continues to be a central problem in robotics[1].

In this paper, we aim to achieve faster control synthesis with guarantees by exploiting a separation of concerns that exists in the problem: (A) how to drive a vehicle/plant to a *given waypoint*? and (B) Which *waypoints* to choose for achieving the ultimate goal? (A) can be solved using powerful control theoretic techniques—if not completely automatically, but at least in a principled fashion, with guarantees, for a broad class of $\mathcal{A}$'s. Given a solution for (A), we solve (B) algorithmically. A contribution of the paper is to identify characteristics of a solution of (A) that make solutions of (B) effective. Consider nonlinear control systems $\mathcal{A} : \frac{d}{dt}x = f(x,u)$ and reach-avoid requirements defined by a goal set G that the trajectories should reach, and obstacles $\mathbf{O}$ the trajectories should avoid. The above separation leads to a two step process: (A) Find a state feedback tracking controller g_{trk} that drives the actual trajectory of the closed-loop system ξ_g to follow a reference trajectory ξ_{ref}. (B) Design a reference controller g_{ref}, which consists of a reference trajectory ξ_{ref} and a reference input u_{ref}. The distance between ξ_g and ξ_{ref} is called the tracking error e. If we can somehow know beforehand the value of e without knowing ξ_{ref}, we can use such error to bloat $\mathbf{O}$ and shrink G, and then synthesize ξ_{ref} such that it is e away from the obstacles (inside the goal set). For linear systems, this was the approach used in [7], but for nonlinear systems, the tracking error e will generally change with ξ_{ref}, and the two steps get entangled.

For a general class of nonlinear vehicles (such as cars, drones, and underwater vehicles), the tracking controller g_{trk} is always designed to minimize the tracking

[1] In the most recent International Conference on Robotics and Automation, among the 3,512 submissions "Path and motion planning" was the second most popular key phrase.

error. The convergence of the error can be proved by a Lyapunov function for certain types of ξ_{ref}. We show how, under reasonable assumptions, we can use Lyapunov functions to bound the value of the tracking error *even when the waypoints changes* (Lemma 2). This error bound is independent of ξ_{ref} so long as ξ_{ref} satisfies the assumptions. For step (B) we introduce a SAT-based trajectory planning methods to find such ξ_{ref} and u_{ref} by solving a satisfiability (SAT) problem over quantifier free linear real arithmetic (Theorem 1). Moreover, the number of constraints in the SMT problem scales linearly to the increase of number of obstacles (and not with the vehicle model). Thus, our methods can scale to complex requirements and high dimensional systems.

Putting it all together, our final synthesis algorithm (Algorithm 2) guarantees that any trajectory following the synthesized reference trajectory will satisfy the reach-avoid requirements. The resulting tool FACTEST is tested with four nonlinear vehicle models and on eight different scenarios, taken from MP literature, which cover a wide range of 2D and 3D environments. Experiment results show that our tool scales very well: it can find the small covers $\{\Theta_j\}_j$ and the corresponding reference trajectories and control inputs satisfying the reach-avoid requirements most often in less than a second, even with up to 22 obstacles. We have also compared our SAT-based trajectory planner to a standard RRT planner, and the results show that our SAT-based method resoundingly outperforms RRT. To summarize, our main contributions are:

1. A method (Algorithm 2) for controller synthesis separating tracking controller g_{trk} and search for reference controller g_{ref}.
2. Sufficient conditions for tracking controller error performance that makes the decomposition work (Lemma 2 and Lemma 3).
3. An SMT-based effective method for synthesizing reference controller g_{ref}.
4. The FACTEST implementation of the above and its evaluation showing very encouraging results in terms of finding controllers that make any trajectories of the closed-loop system satisfy reach-avoid requirements (Sect. 6).

Related Works. *Model Predictive Control (MPC).* MPC [4,25,45,49] has to solve a constrained, discrete-time, optimal control problem. MPC for controller synthesis typically requires model reduction for casting the optimization problem as an LP [4], QP [2,36], MILP [33,34,45]. However, when the plant model is nonlinear [8,22], it may be hard to balance speed and complex requirements as the optimization problem become nonconvex and nonlinear.

Discrete Abstractions. Discrete, finite-state, abstraction of the control system is computed, and then a discrete controller is synthesized by solving a two-player game [10,17,24,42,47]. CoSyMA [28], Pessoa [37], LTLMop [18,46], Tulip [9,48], and SCOTS [38] are based on these approaches. The discretization step often leads to a severe state space explosion for higher dimensional models.

Safe Motion Planning. The idea of bounding the tracking error through precomputation has been used in several techniques: FastTrack [11] uses Hamilton-Jacobi reachability analysis to produce a "safety bubble" around planed paths.

Reachability based trajectory design for dynamical environments (RTD) [44] computes an offline forward reachable sets to guarantee that the robot is not-at-fault in any collision. In [40], a technique based on convex optimization is used to compute tracking error bounds. Another technique [23,43] uses motion primitives expanded by safety funnels, which defines similar ideas of safety tubes.

Sampling Based Planning. Probabilistic Road Maps (PRM) [15], Rapidly-exploring Random Trees (RRT) [19], and fast marching tree (FMT) [12] are widely used in actual robotic platforms. They can generate feasible trajectories through known or partially known environments. Compared with the deterministic guarantees provided by our proposed method, these methods come with stochastic guarantees. Also, they are not designed to be robust to model uncertainty or disturbances. MoveIT [5] is a tool designed to implement and benchmark various motion planners on robots. The motion planners in MoveIT are from the open motion planning library (OMPL) [41], which implements motion planners abstractly.

Controlled Lyapunov Function (CLF). CLF have been used to guarantee that the overall closed-loop controlled system satisfies a reach-while-stay specification [35]. Instead of asking for a CLF for the overall closed-loop system, our method only needs a Lyapunov function for the tracking error, which is a weaker local requirement. CLF is often a difficult requirement to meet for nonlinear vehicle models.

2 Preliminaries and Problem Statement

Let us denote real numbers by $\mathbb{R}$, non-negative real numbers by $\mathbb{R}_{\geq 0}$, and natural numbers by $\mathbb{N}$. The n-dimensional *Euclidean space* is $\mathbb{R}^n$. For a vector $x \in \mathbb{R}^n$, $x^{(i)}$ is the i^{th} entry of x and $\|x\|_2$ is the 2-norm of x. For any matrix $A \in \mathbb{R}^{n \times m}$, $A^\intercal$ is its *transpose*; $A^{(i)}$ is the i^{th} row of A. Given a $r \geq 0$, an *r-ball* around $x \in \mathbb{R}^n$ is defined as $B_r(x) = \{x' \in \mathbb{R}^n \mid ||x' - x||_2 \leq r\}$. We call r the radius of the ball. Given a matrix $H \in \mathbb{R}^{r \times n}$ and a vector $b \in \mathbb{R}^r$, an (H, b)-*polytope* is denoted by $Poly(H, b) = \{x \in \mathbb{R}^n \mid Hx \leq b\}$. Each row of the inequality $H^{(i)}x \leq b^{(i)}$ defines a *halfspace*. We also call $H^{(i)}x = b^{(i)}$ the *surface* of the polytope. Let $\mathsf{dP}(H) = r$ denotes the number of rows in H. Given a set $S \subseteq \mathbb{R}^n$, the radius of S is defined as $\sup_{x,y \in S} \|x - y\|_2/2$.

State Space and Workspace. The state space of control systems will be a subspace $\mathcal{X} \subseteq \mathbb{R}^n$. The *workspace* is a subspace $\mathcal{W} \subseteq \mathbb{R}^d$, for $d \in \{2, 3\}$, which is the physical space in which the robots have to avoid obstacles and reach goals. Given a state vector $x \in \mathcal{X}$, its projection to $\mathcal{W}$ is denoted by $x \downarrow p$. That is, $x \downarrow p = [p_x, p_y]^\intercal \in \mathbb{R}^2$ for ground vehicles on the plane and $x \downarrow p = [p_x, p_y, p_z]^\intercal \in \mathbb{R}^3$ for aerial and underwater vehicles. When x is clear from context we will write $x \downarrow p$ as simply p. The vector x may include other variables like velocity, heading, pitch, etc., but p only has the position in Cartesian coordinates. We assume that the goal set $G := Poly(H_G, b_G)$ and the unsafe set $\mathbf{O}$ (obstacles) are specified by polytopes in $\mathcal{W}$; $\mathbf{O} = \cup O_i$, where $O_i := Poly(H_{O,i}, b_{O,i})$ for each obstacle i.

Trajectories and Reach-Avoid Requirements. A *trajectory* ξ over $\mathcal{X}$ of duration T is a function $\xi : [0, T] \to \mathcal{X}$, that maps each time t in the time *domain* $[0, T]$ to a point $\xi(t) \in \mathcal{X}$. The *time bound or duration* of ξ is denoted by $\xi.\mathsf{ltime} = T$. The projection of a trajectory $\xi : [0, T] \to \mathcal{X}$ to $\mathcal{W}$ is written as $\xi \downarrow p : [0, T] \to \mathcal{W}$ and defined as $(\xi \downarrow p)(t) = \xi(t) \downarrow p$. We say that a trajectory $\xi(t)$ *satisfies a reach-avoid requirement given by unsafe set* $\mathbf{O}$ *and goal set* G if $\forall t \in [0, \xi.\mathsf{ltime}], \xi(t) \downarrow p \notin \mathbf{O}$ and $\xi(\xi.\mathsf{ltime}) \downarrow p \in G$. See Fig. 1 for an example.

Given a trajectory $\xi : [0, T] \to \mathcal{X}$ and a time $t > 0$, the *time shift* of ξ is a function $(\xi + t) : [t, t + T] \to \mathcal{X}$ defined as $\forall t' \in [t, t + T], (\xi + t)(t') = \xi(t' - t)$. Strictly speaking, for $t > 0, \xi + t$ is not a trajectory. The *concatenation* of two trajectories $\xi_1 \frown \xi_2$ is a new trajectory in which ξ_1 is followed by ξ_2. That is, for each $t \in [0, \xi_1.\mathsf{ltime}+\xi_2.\mathsf{ltime}]$, $(\xi_1 \frown \xi_2)(t) = \xi_1(t)$ when $t \leq \xi_1.\mathsf{ltime}$, and equals $\xi_2(t - \xi_1.\mathsf{ltime})$ when $t > \xi_1.\mathsf{ltime}$. Trajectories are closed under concatenation, and many trajectories can be concatenated in the same way.

2.1 Nonlinear Control System

Definition 1. *An* (n, m)-dimensional control system $\mathcal{A}$ *is a* 4*-tuple* $\langle \mathcal{X}, \Theta, \mathbf{U}, f \rangle$ *where (i)* $\mathcal{X} \subseteq \mathbb{R}^n$ *is the state space, (ii)* $\Theta \subseteq \mathcal{X}$ *is the initial set, (iii)* $\mathbf{U} \subseteq \mathbb{R}^m$ *is the input space, and (iv)* $f : \mathcal{X} \times \mathbf{U} \to \mathcal{X}$ *is the* dynamic function *that is Lipschitz continuous with respect to the first argument.*

A control system with no inputs ($m = 0$) is called a *closed* system.

Let us fix a time duration $T > 0$. An *input trajectory* $u : [0, T] \to \mathbf{U}$, is a continuous trajectory over the input space $\mathbf{U}$. We denote the set of all possible input trajectories to be $\mathcal{U}$. Given an input signal $u \in \mathcal{U}$ and an initial state $x_0 \in \Theta$, a *solution* of $\mathcal{A}$ is a continuous trajectory $\xi_u : [0, T] \to \mathcal{X}$ that satisfies (i) $\xi_u(0) = x_0$ and (ii) for any $t \in [0, T]$, the time derivative of ξ_u at t satisfies the differential equation:

$$\frac{d}{dt}\xi_u(t) = f(\xi_u(t), u(t)). \tag{1}$$

For any $x_0 \in \Theta, u \in \mathcal{U}$, ξ_u is a state trajectory and we call such a pair (ξ_u, u) a state-input trajectory pair.

A *reference state trajectory* (or *reference trajectory* for brevity) is a trajectory over $\mathcal{X}$ that the control system tries to follow. We denote reference trajectories by ξ_{ref}. Similarly, a *reference input trajectory* (or *reference input*) is a trajectory over $\mathbf{U}$ and we denote them as u_{ref}. Note these ξ_{ref} and u_{ref} are not necessarily solutions of (1). Figure 1 shows reference and actual solution trajectories.

We call a reference trajectory ξ_{ref} and a reference input u_{ref} together as a reference controller g_{ref}. Given g_{ref}, a *tracking controller* g_{trk} is a function that is used to compute the inputs for $\mathcal{A}$ so that in the resulting closed system, the state trajectories try to follow ξ_{ref}.

Definition 2. *Given an* (n, m)*-dynamical system* $\mathcal{A}$*, a reference trajectory* ξ_{ref}*, and a reference input* u_{ref}*, a* tracking controller *for the triple* $\langle \mathcal{A}, \xi_{ref}, u_{ref} \rangle$ *is a (state feedback) function* $g_{trk} : \mathcal{X} \times \mathcal{X} \times \mathbf{U} \to \mathbf{U}$.

At any time t, the tracking controller g_{trk} takes in a current state of the system x, a reference trajectory state $\xi_{\mathsf{ref}}(t)$, and a reference input $u_{\mathsf{ref}}(t)$, and gives an input $g_{\mathsf{trk}}(x, \xi_{\mathsf{ref}}(t), u_{\mathsf{ref}}(t)) \in \mathbf{U}$ for $\mathcal{A}$. The controller g for $\mathcal{A}$ is determined by both the reference controller g_{ref} and the tracking controller g_{trk}. The resulting trajectory ξ_g of the closed control system ($\mathcal{A}$ closed with g_{ref} and g_{trk}) satisfies:

$$\frac{d}{dt}\xi_g(t) = f\left(\xi_g(t), g_{\mathsf{trk}}\left(\xi_g(t), \xi_{\mathsf{ref}}(t), u_{\mathsf{ref}}(t)\right)\right), \forall\, t \in [0,T]\backslash D, \tag{2}$$

where D is the set of points in time where the second or third argument of g_{trk} is discontinuous[2].

2.2 Controller Synthesis Problem

Definition 3. *Given a (n,m)-dimensional nonlinear system $\mathcal{A} = \langle \mathcal{X}, \Theta, \mathbf{U}, f\rangle$, its workspace $\mathcal{W}$, goal set $G \subseteq \mathcal{W}$ and the unsafe set $\mathbf{O} \subseteq \mathcal{W}$, we are required to find (a) a tracking controller g_{trk}, (b) a partition $\{\Theta_j\}_j$ of Θ, and (c) for each partition Θ_j, a reference controller $g_{j,\mathsf{ref}}$, which consists of a state trajectory $\xi_{j,\mathsf{ref}}$ and an input trajectory $u_{j,\mathsf{ref}}$, such that $\forall x_0 \in \Theta_j$, the unique trajectory ξ_g of the closed system as in Eq. (2) starting from x_0 reaches G and avoids $\mathbf{O}$.*

Again, $\xi_{j,\mathsf{ref}}$ and $u_{j,\mathsf{ref}}$ in $g_{j,\mathsf{ref}}$ are not required to be a state-input pair, but, for each initial state $x_0 \in \Theta_j$, the closed loop trajectory ξ_g following ξ_{ref} *is* a valid state trajectory with corresponding input u generated by g_{trk} and $g_{j,\mathsf{ref}}$. In this paper, we will decompose the controller synthesis problem: Part (a) will be delivered by design engineers with knowledge of vehicle dynamics, and parts (b) and (c) will be automatically synthesized by our algorithm. The latter being the main contribution of the paper.

Example 1. Consider a ground vehicle moving on a 2D workspace $\mathcal{W} \subseteq \mathbb{R}^2$ as shown in Fig. 1.

This scenario is called Zigzag and it is adopted from [32]. The red polytopes are obstacles. The blue and green polytopes are the initial set Θ and the goal set G. There are also obstacles (not shown in the figure) defining the boundaries of the entire workspace. The black line is a projection of a reference trajectory to the workspace: $\xi_{\mathsf{ref}}(t) \downarrow p$. This would not be a feasible state trajectory for a ground vehicle that cannot make sharp turns. The purple dashed curve is a

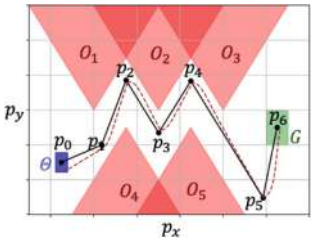


Fig. 1. Zigzag scenario for a controller synthesis problem. The initial set is blue, the goal set is green, and the unsafe sets are red. A valid reference trajectory is shown in black and a feasible trajectory is shown in purple. (Color figure online)

[2] ξ_g is a standard solution of ODE with piece-wise continuous right hand side.

real feasible state trajectory of the system starting from Θ with a tracking controller g_{trk}, where g_{trk} will be introduced in Example 2.

Consider the standard nonlinear bicycle model of a car [31]. The control system has 3 state variables: the position p_x, p_y, and the heading direction θ. Its motion is controlled by two inputs: linear velocity v and rotational velocity ω. The car's dynamics are given by:

$$\frac{d}{dt}p_x = v\cos(\theta),\ \frac{d}{dt}p_y = v\sin(\theta),\ \frac{d}{dt}\theta = \omega. \tag{3}$$

3 Constructing Reference Trajectories from Waypoints

If $\xi_{\mathsf{ref}}(t) \downarrow p$ is a PWL (PWL) curve in the workspace $\mathcal{W}$, we call $\xi_{\mathsf{ref}}(t)$ a PWL reference trajectory. In $\mathcal{W}$, a PWL curve can be determined by the endpoints of each line segment. We call such endpoints the *waypoints* of the PWL reference trajectory. In Fig. 1, the black points $p_0, \cdots, p_6$ are waypoints of $p(t) = \xi_{\mathsf{ref}}(t) \downarrow p$.

Consider any vehicle on the plane[3] with state variables p_x, p_y, θ, v (x-position, y-position, heading direction, linear velocity) and input variables a, ω (acceleration and angular velocity). Once the waypoints $\{p_i\}_{i=0}^k$ are fixed, and if we enforce constant speed $\bar{v}$ (i.e., $\xi_{\mathsf{ref}}(t) \downarrow v = \bar{v}$ for all $t \in [0, \xi_{\mathsf{ref}}.\mathsf{ltime}]$), then $\xi_{\mathsf{ref}}(t)$ can be uniquely defined by $\{p_i\}_{i=0}^k$ and $\bar{v}$ using Algorithm 1. The semantics of ξ_{ref} and u_{ref} returned by `Waypoints_to_Traj` is that the reference trajectory requires the vehicle to move at a constant speed $\bar{v}$ along the lines connecting the waypoints $\{p_i\}_{i=0}^k$. In Example 1, $\xi_{\mathsf{ref}}(t), u_{\mathsf{ref}}(t)$ can also be constructed using `Waypoints_to_Traj` moving v to input variables and dropping a.

We notice that if $k = 1$, $\xi_{\mathsf{ref}}(t), u_{\mathsf{ref}}(t)$ returned by Algorithm 1 is a valid state-input trajectory pair. However, if $k > 1$, $\xi_{\mathsf{ref}}(t), u_{\mathsf{ref}}(t)$ returned by Algorithm 1 is usually not a valid state-input trajectory pair. This is because $\theta_{\mathsf{ref}}(t)$ is discontinuous at the waypoints and no bounded inputs $u_{\mathsf{ref}}(t)$ can drive the vehicle to achieve such $\theta_{\mathsf{ref}}(t)$. Therefore, when $k > 1$, $\xi_{\mathsf{ref}}(t)$ is a PWL reference trajectory with no $u_{\mathsf{ref}}(t)$ such that $\xi_{\mathsf{ref}}, u_{\mathsf{ref}}$ are solutions of (1).

Algorithm 1: `Waypoints_to_Traj`$(\{p_i\}_{i=0}^k, \bar{v})$

input : $\{p_i\}_{i=0}^k, \bar{v}$

1 $\forall t \in [0, \sum_{i=1}^k \frac{\|p_j - p_{j-1}\|_2}{\bar{v}}], v_{\mathsf{ref}}(t) = \bar{v},\ a_{\mathsf{ref}}(t) = 0, \omega_{\mathsf{ref}}(t) = 0;$

2 $\forall i \geq 1, \forall t \in \left[\sum_{j=1}^{i-1} \frac{\|p_j - p_{j-1}\|_2}{\bar{v}}, \sum_{j=1}^{i} \frac{\|p_j - p_{j-1}\|_2}{\bar{v}}\right),$
$p_{\mathsf{ref}}(t) = p_{i-1} + \bar{v}t - \sum_{j=1}^{i-1} \|p_j - p_{j-1}\|_2,$
$\theta_{\mathsf{ref}}(t) = \mathrm{mod}(\mathrm{atan2}((p_{y,i} - p_{y,i-1}), (p_{x,i} - p_{x,i-1}), 2\pi);$

3 $\xi_{\mathsf{ref}}(t) = [p_{\mathsf{ref}}(t), \theta_{\mathsf{ref}}(t), v_{\mathsf{ref}}(t)];$

4 $u_{\mathsf{ref}}(t) = [a_{\mathsf{ref}}(t), \omega_{\mathsf{ref}}(t)];$

5 **return** $\xi_{\mathsf{ref}}(t), u_{\mathsf{ref}}(t)$;

[3] A similar construction works for vehicles in 3D workspaces with additional variables.

Proposition 1. *Given a sequence of waypoints* $\{p_i\}_{i=0}^k$ *and a constant speed* $\bar{v}$, $\xi_{ref}(t), u_{ref}(t)$ *produced by* `Waypoints_to_Traj`$(\{p_i\}_{i=0}^k, \bar{v})$ *satisfy:*

- $p_{ref}(t) = \xi_{ref}(t) \downarrow p$ *is a piece-wise continuous function connecting* $\{p_i\}_{i=0}^k$.
- *At time* $t_i = \sum_{j=1}^{i} \|p_j - p_{j-1}\|_2/\bar{v}$, $p_{ref}(t_i) = p_i$. *We call* $\{t_i\})_{i=1}^k$ *the concatenation time.*
- $\xi_{ref}(t) = \xi_{ref,1}(t) \frown \cdots \frown \xi_{ref,k}(t)$ *and* $u_{ref}(t) = u_{ref,1}(t) \frown \cdots \frown u_{ref,k}(t)$, *where* $(\xi_{ref,i}, u_{ref,i})$ *are state-input trajectory pairs returned by the function* `Waypoints_to_Traj`$(\{p_{i-1}, p_i\}, \bar{v})$.

Outline of Synthesis Approach. In this Section, we present an Algorithm `Waypoints_to_Traj` for constructing reference trajectories from arbitrary sequence of waypoints. In Sect. 4, we precisely characterize the type of vehicle tracking controller our method requires from designers. On our tool's webpage [27], we show with several extra examples that indeed developing such controllers is non-trivial, far from automatic, yet bread and butter of control engineers. In Sect. 5, we present the main synthesis algorithm, which uses the tracking error bounds from the previous section, to construct waypoints, for each initial state, which when passed through `Waypoints_to_Traj` provide the solutions to the synthesis problem.

4 Bounding the Error of a Tracking Controller

4.1 Tracking Error and Lyapunov Functions

Given a reference controller g_{ref}, a tracking controller g_{trk}, and an initial state $x_0 \in \Theta$, the resulting trajectory ξ_g of the closed control system ($\mathcal{A}$ closed with g_{ref} and g_{trk}) is a state trajectory that starts from x_0 and follows the ODE (2). In this setting, we define the *tracking error* at time t to be a continuous function:

$$e : \mathcal{X} \times \mathcal{X} \to \mathbb{R}^n.$$

When $\xi_g(t)$ and $\xi_{\mathsf{ref}}(t)$ are fixed, we also write $e(t) = e(\xi_g(t), \xi_{\mathsf{ref}}(t))$ which makes it a function of time. One thing to remark here is that if $\xi_{\mathsf{ref}}(t)$ is discontinuous, then $e(t)$ is also discontinuous. In this case, the derivative of $e(t)$ cannot be defined at the points of discontinuity. To start with, let us assume that $g_{\mathsf{ref}} = (\xi_{\mathsf{ref}}, u_{\mathsf{ref}})$ is a valid state-input pair so ξ_{ref} is a continuous state trajectory. Later we will see that the analysis can be extended to cases when ξ_{ref} is discontinuous but a concatenation of continuous state trajectories.

When $(\xi_{\mathsf{ref}}, u_{\mathsf{ref}})$ is a valid state-input pair and $e(t)$ satisfy an differential equation $\frac{d}{dt}e(t) = f_e(e(t))$, we use Lyapunov functions, which is a classic technique for proving stability of an equilibrium of an ODE, to bound the tracking error $e(t)$. The Lie derivative $\frac{\partial V}{\partial e} f_e(e)$ below captures the rate of change of the function V along the trajectories of $e(t)$.

Definition 4 (Lyapunov functions [16]). *Fix a state-input reference trajectory pair (ξ_{ref}, u_{ref}), assume that the dynamics of the tracking error e for a closed control system $\mathcal{A}$ with g_{ref} and g_{trk} can be rewritten as $\frac{d}{dt}e(t) = f_e(e(t))$, where $f_e(0) = 0$. A continuously differentiable function $V : \mathbb{R}^n \to \mathbb{R}$ satisfying (i) $V(0) = 0$, (ii) $\forall e \in \mathbb{R}^n, V(e) \geq 0$, and (iii) $\forall e \in \mathbb{R}^n, \frac{\partial V}{\partial e} f_e(e) \leq 0$, is called a Lyapunov function for the tracking error.*

Example 2. For the car of Example 1, with a continuous reference trajectory $\xi_{\mathsf{ref}}(t) = [x_{\mathsf{ref}}(t), y_{\mathsf{ref}}(t), \theta_{\mathsf{ref}}(t)]^\intercal$, we define the tracking error in a coordinate frame fixed to the car [13]:

$$\begin{pmatrix} e_x(t) \\ e_y(t) \\ e_\theta(t) \end{pmatrix} = \begin{pmatrix} \cos(\theta(t)) & \sin(\theta(t)) & 0 \\ -\sin(\theta(t)) & \cos(\theta(t)) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{\mathsf{ref}}(t) - p_x(t) \\ y_{\mathsf{ref}}(t) - p_y(t) \\ \theta_{\mathsf{ref}}(t) - \theta(t) \end{pmatrix}. \tag{4}$$

With the reference controller function g defined as:

$$\begin{aligned} v(t) &= v_{\mathsf{ref}}(t)\cos(e_\theta(t)) + k_1 e_x(t), \\ \omega(t) &= \omega_{\mathsf{ref}}(t) + v_{\mathsf{ref}}(t)(k_2 e_y(t) + k_3 \sin(e_\theta(t))), \end{aligned} \tag{5}$$

it has been shown in [13] when $k_1, k_2, k_3 > 0$, $\frac{d}{dt}\omega_{\mathsf{ref}}(t) = 0$, and $\frac{d}{dt}v_{\mathsf{ref}}(t) = 0$,

$$V([e_x, e_y, e_\theta]^\intercal) = \frac{1}{2}(e_x^2 + e_y^2) + \frac{1 - \cos(e_\theta)}{k_2} \tag{6}$$

is a Lyapunov function with negative semi-definite time derivative $\frac{\partial V}{\partial x} f_e = -k_1 e_x^2 - \frac{v_{\mathsf{ref}} k_3 \sin^2(e_\theta)}{k_2}$.

4.2 Bounding Tracking Error Using Lyapunov Functions: Part 1

Consider a given closed control system, $\mathcal{A}$ with g_{ref} and g_{trk}, in this section, we will derive upper bounds on the tracking error e. Later in Sect. 5, we will develop techniques that take the tracking error into consideration for computing reference trajectories ξ_{ref}.

To begin with, we consider state-input reference trajectory pairs $(\xi_{\mathsf{ref}}, u_{\mathsf{ref}})$ where u_{ref} is continuous, and therefore, ξ_{ref} and ξ_g are differentiable. Let us assume that the tracking error dynamics ($\frac{d}{dt}e(t) = f_e(e(t))$) has a Lyapunov function $V(e(t))$. The following is a standard result that follows from the theory of Lyapunov functions for dynamical systems.

Lemma 1. *Consider any state-input trajectory pair (ξ_{ref}, u_{ref}), an initial state x_0, the corresponding trajectory ξ_g of the closed control system, and a constant $\ell > 0$. If the tracking error $e(t)$ has a Lyapunov function V, and if initially $V(e(0)) \leq \ell$, then for any $t \in [0, \xi_{ref}.\mathsf{ltime}]$, $V(e(t)) \leq \ell$.*

This lemma is proved by showing that $V(e(t)) = V(e(0)) + \int_0^t \frac{d}{dt}V(e(\tau))d\tau \leq V(e(0))$. The last inequality holds since $\frac{d}{dt}V(e(\tau)) = \frac{\partial V}{\partial e} f_e(e) \leq 0$ for any $\tau \in [0, t]$ according the definition of Lyapunov functions (Definition 4).

Lemma 1 says that if we can bound $V(e(0)) = V(e(x_0, \xi_{\mathsf{ref}}(0)))$, we can bound $V(e(\xi_g(t), \xi_{\mathsf{ref}}(t)))$ at any time t within the domain of the trajectories, regardless of the value of $\xi_{\mathsf{ref}}(t)$. This could decouple the problem of designing the tracking controller g_{trk} and synthesizing the reference controller g_{ref} as a state-input trajectory pair $(\xi_{\mathsf{ref}}, u_{\mathsf{ref}})$.

Example 3. Given two waypoints p_0, p_1 for the car in Example 1, take the returned value of `Waypoints_to_Traj`$(\{p_0, p_1\}, \bar{v})$, move v_{ref} to u_{ref} and drop a_{ref}. Then, the resulting $(\xi_{\mathsf{ref}}, u_{\mathsf{ref}})$ is a continuous and differentiable state-input reference trajectory pair. Moreover, if the robot is controlled by the tracking controller as in Eq. (5), $V(e(t)) = \frac{1}{2}(e_x(t)^2 + e_y(t)^2) + \frac{1-\cos(e_\theta(t))}{k_2}$ is a Lyapunov function for the corresponding tracking error $e(t) = [e_x(t), e_y(t), e_\theta(t)]^\intercal$.

From Eq. (4), it is easy to check that $e_x^2(t) + e_y(t)^2 = (x_{\mathsf{ref}}(t) - p_x(t))^2 + (y_{\mathsf{ref}}(t) - p_y(t))^2$ for any time t. Assume that initially the position of the vehicle satisfies $[p_x(0), p_y(0)]^\intercal \in B_\ell([x_{\mathsf{ref}}(0), y_{\mathsf{ref}}(0)]^\intercal)$. We check that $V(e(0)) = \frac{1}{2}(e_x(0)^2 + e_y(0)^2) + \frac{1-\cos(e_\theta(0))}{k_2} \leq \frac{\ell^2}{2} + \frac{2}{k_2}$.

From Lemma 1, we know that $\forall t \in [0, \xi_{\mathsf{ref}}.\mathsf{ltime}]$, $V(e(t)) \leq \frac{\ell^2}{2} + \frac{2}{k_2}$. Then we have $(x_{\mathsf{ref}}(t) - p_x(t))^2 + (y_{\mathsf{ref}}(t) - p_y(t))^2 = (e_x(t)^2 + e_y(t)^2) \leq \ell^2 + \frac{4}{k_2}$. That is, the position of the robot at time t satisfies $[p_x(t), p_y(t)]^\intercal \in B_{\sqrt{\ell^2 + \frac{4}{k_2}}}([x_{\mathsf{ref}}(t), y_{\mathsf{ref}}(t)]^\intercal)$.

4.3 Bounding Tracking Error Using Lyapunov Functions: Part 2

Next, let us consider the case where ξ_{ref} is discontinuous. Furthermore, let us assume that it is a concatenation of several continuous state trajectories $\xi_{\mathsf{ref},1} \frown \cdots \frown \xi_{\mathsf{ref},k}$. In this case, we call ξ_{ref} a piece-wise reference trajectory. If we have a sequence of $(\xi_{\mathsf{ref},i}, u_{\mathsf{ref},i})$, each is a valid state-input trajectory pair and the corresponding error $e_i(t)$ has a Lyapunov function $V_i(e_i(t))$, then we can use Lemma 1 to bound the error of $e_i(t)$ if we know the value of $e_i(0)$. However, the main challenge to glue these error bounds together is that $e(t)$ would be discontinuous with respect to the entire piece-wise $\xi_{\mathsf{ref}}(t)$.

Without loss of generality, let us assume that the Lyapunov functions $V_i(e_i(t))$ share the same format. That is, $\forall i, V_i(e_i(t)) = V(e_i(t))$. Let t_i be the concatenation time points when $\xi_{\mathsf{ref}}(t)$ (and therefore $e(t)$) is discontinuous. We know that $\lim_{t \to t_i^-} V(e(t)) \neq \lim_{t \to t_i^+} V(e(t))$ since $\lim_{t \to t_i^-} e(t) \neq \lim_{t \to t_i^+} e(t)$.

One insight we can get from Example 3 is that although $e(t)$ is discontinuous at time t_is, some of the variables influencing $e(t)$ are continuous. For example, $e_x(t)$ and $e_y(t)$ in Example 3, which represent the error of the positions, are continuous since both the actual and reference positions of the vehicle are continuous. If we can further bound the term in $V(e(t))$ that corresponds to the *other* variables, we could analyze the error bound for the entire piece-wise reference trajectory. With this in sight, let us write $e(t)$ as $[e_p(t), e_r(t)]$, where $e_p(t) = e(t) \downarrow p$ is the projection to $\mathcal{W}$ and $e_r(t)$ is the remaining components.

Let us further assume that the Lyapunov function can be written in the form of $V(e(t)) = \alpha(e_p(t)) + \beta(e_r(t))$. Indeed, on the tool's webpage [27] we show

that four commonly used vehicle models (car, robot, underwater vehicle, and hovercraft) have Lyapunov functions for the tracking error $e(t)$ of this form. If $\beta(e_r(t))$ can be further bounded, then the tracking error for the entire trajectory can be bounded using the following lemma.

Lemma 2. *Consider* $\xi_{ref} = \xi_{ref,1} \frown \cdots \frown \xi_{ref,k}$, *and* $u_{ref} = u_{ref,1} \frown \cdots \frown u_{ref,k}$ *as a piecewise reference and input with each* $(\xi_{ref,i}, u_{ref,i})$ *being a state-input trajectory pair. Suppose (1)* $V(e(t)) = \alpha(e_p(t)) + \beta(e_r(t))$ *be a Lyapunov function for the tracking error* $e(t)$ *of each piece* $(\xi_{ref,i}, u_{ref,i})$*; (2)* $e_p(t)$ *is continuous and* $\alpha(\cdot)$ *is a continuous function; (3)* $\beta(e_r(t)) \in [b_l, b_u]$, *and (4)* $V(e(0)) \leq \varepsilon_0$. *Then, the tracking error* $e(t)$ *with respect to* ξ_{ref} *and* u_{ref} *can be bounded by,*

$$V(e(t)) \leq \varepsilon_i, \forall i \geq 1, \forall t \in [t_{i-1}, t_i),$$

where $\forall\ i > 1, \varepsilon_i = \varepsilon_{i-1} - b_l + b_u$, $\varepsilon_1 = \varepsilon_0$ *being the bound on the initial tracking error, and* t_i*'s are the time points of concatenation*[4].

Proof. We prove this by induction on i. When $i = 1$, we know from Lemma 1 that if the initial tracking error is bounded by $V(e(0))$, then for any $t \in [0, t_1), V(e(t)) \leq V(e(0)) \leq \varepsilon_0 = \varepsilon_1$, so the lemma holds.

Fix any $i \geq 1$. If $V(e(t_{i-1})) \leq \varepsilon_i$, from Lemma 1 we have $\forall t \in [t_{i-1}, t_i)$, $V(e(t)) \leq \varepsilon_i$. Also, $\lim_{t \to t_i^-} V(e(t)) = \lim_{t \to t_i^-} \alpha(e_p(t)) + \beta(e_r(t)) \leq \varepsilon_i$. Since $\forall e_r(t) \in \mathbb{R}^{n-d}, \beta(e_r(t)) \in [b_l, b_u]$, we have $\lim_{t \to t_i^-} \alpha(e_p(t)) \leq \varepsilon_i - b_l$, and $\lim_{t \to t_i^-} \alpha(e_p(t)) = \lim_{t \to t_i^+} \alpha(e_p(t))$. Therefore,

$$\varepsilon_{i+1} = \lim_{t \to t_i^+} V(e(t)) = \lim_{t \to t_i^+} \alpha(e_p(t)) + \beta(e_r(t)) \leq \varepsilon_i - b_l + b_u.$$

Another observation we have on the four vehicle models used in this paper is that not only $V(e(t))$ can be written as $\alpha(e_p(t)) + \beta(e_r(t))$ with $\beta(e_r(t))$ being bounded, but also $\alpha(e_p(t))$ can be written as $\alpha(e_p(t)) = ce_p^\intercal(t)e_p(t) = c\|p(t) - p_{\mathsf{ref}}(t)\|_2^2$, where $c \in \mathbb{R}$ is a scalar constant; $p(t) = \xi_g(t) \downarrow p$ and $p_{\mathsf{ref}}(t) = \xi_{\mathsf{ref}}(t) \downarrow p$ are the actual position and reference position of the vehicle. In this case, we can further bound the position of the vehicle $p(t)$.

Lemma 3. *In addition to the assumptions of Lemma 2, if* $\alpha(e_p(t)) = ce_p^\intercal(t)e_p(t) = c\|p(t) - p_{ref}(t)\|_2^2$, *where* $c \in \mathbb{R}, p(t) = \xi_g(t) \downarrow p$ *and* $p_{ref}(t) = \xi_{ref}(t) \downarrow p$. *Then we have that at time* $t \in [t_{i-1}, t_i)$,

$$e_p^\intercal(t)e_p(t) \leq \frac{\varepsilon_i - b_l}{c},$$

where ε_i *and* b_l *are from Lemma 2, which implies that*

$$p(t) \in B_{\ell_i}(p_{ref}(t)), \text{with } \ell_i = \sqrt{\frac{\varepsilon_i - b_l}{c}}.$$

[4] $\forall t \in [t_{i-1}, t_i), \xi_{\mathsf{ref}}(t) = \xi_{\mathsf{ref},i}(t - \sum_{j=1}^{i-1} \xi_{\mathsf{ref},j}.\mathsf{ltime})$.

Note that Lemma 2 and 3 does not depend on the concrete values of ξ_{ref} and u_{ref}. The lemmas hold for any piece-wise reference trajectory ξ_{ref} and reference input u_{ref} as long as the corresponding error e has a Lyapunov function (for each piece of ξ_{ref} and u_{ref}).

Example 4. Continue Example 3.

Now let us consider the case of a sequence of waypoints $\{p_i\}_{i=0}^k$. Let $(\xi_{\mathsf{ref}}, u_{\mathsf{ref}}) = \texttt{Waypoints\_to\_Traj}(\{p_i\}_{i=0}^k, \bar{v})$. From Example 3, we know that $V(e(t)) = \frac{1}{2}(e_x(t)^2 + e_y(t)^2) + \frac{1-\cos(e_\theta(t))}{k_2}$ is a Lyapunov function for each segment of the piece-wise reference trajectory $\xi_{\mathsf{ref}}(t)$. We also know that for any value of e_θ, the term $\frac{1-\cos(e_\theta(t))}{k_2} \in [0, \frac{2}{k}]$. From Lemma 2, we have that for $t \in [t_{i-1}, t_i)$ where t_i are the concatenation time points, we have $V(e(t)) \leq V(e(0)) + \frac{2(i-1)}{k_2}$ Therefore, following Example 3, initially $V(e(0)) \leq \frac{\ell^2}{2} + \frac{2}{k_2}$. Then $\forall t \in [t_{i-1}, t_i)$, $V(e(t)) \leq \frac{\ell^2}{2} + \frac{2i}{k_2}$, and the position of the robot satisfies $[p_x(t), p_y(t)]^\intercal \in B_{\sqrt{\ell^2 + \frac{4i}{k_2}}}([x_{\mathsf{ref}}(t), y_{\mathsf{ref}}(t)]^\intercal)$.

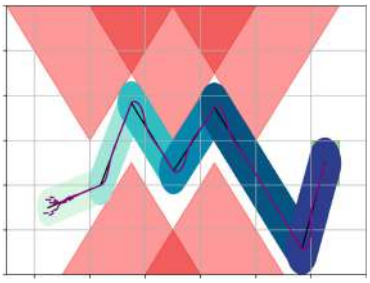


Fig. 2. Illustration of the error bounds computed from Lemma 3. The i^{th} line segment is bloated by $\sqrt{\ell^2 + \frac{4i}{k_2}}$. The closed-loop system's trajectory $p(t)$ are purple curves and they are contained by the bloated-tube. (Color figure online)

As seen in Fig. 2, we bloat the black reference trajectory $p_{\mathsf{ref}}(t) = \xi_{\mathsf{ref}}(t) \downarrow p$ by $\ell_i = \sqrt{\ell^2 + \frac{4i}{k_2}}$ for the i^{th} line segment, the bloated tube contains the real position trajectories (purple curves) $p(t)$ of the closed system.

5 Synthesizing the Reference Trajectories

In Sect. 4.3, we have seen that under certain conditions, the tracking error $e(t)$ between an actual closed-loop trajectory $\xi_g(t)$ and a piece-wise reference $\xi_{\mathsf{ref}}(t)$ can be bounded by a piece-wise constant value, which depends on the initial tracking error $e(0)$ and the number of segments in ξ_{ref}. We have also seen an example nonlinear vehicle model with PWL ξ_{ref} for which the tracking error can be bounded.

In this section, we discuss how to utilize such bound on $e(t)$ to help find a reference controller g_{ref} consisting of a reference trajectory $\xi_{\mathsf{ref}}(t)$ and a reference input $u_{\mathsf{ref}}(t)$ such that closed-loop trajectories $\xi_g(t)$ from a neighborhood of $\xi_{\mathsf{ref}}(0)$ that are trying to follow $\xi_{\mathsf{ref}}(t)$ are guaranteed to satisfy the reach-avoid requirement. The idea of finding a g_{ref} follows a classic approach in robot motion planning. The intuition is that if we know at any time $t \in [0, \xi_{\mathsf{ref}}.\mathsf{ltime}]$, $\|\xi_g(t) \downarrow p - \xi_{\mathsf{ref}}(t) \downarrow p\|_2$ will be at most ℓ, then instead of requiring $\xi_{\mathsf{ref}}(t) \downarrow p$ to be at least ℓ away from the obstacles (inside the goal region), we will bloat the obstacles (shrink the goal set) by ℓ. Then the original problem is reduced to

finding a $\xi_{\mathsf{ref}}(t)$ such that $\xi_{\mathsf{ref}}(t) \downarrow p$ can avoid the bloated obstacles and reach the shrunk goal set.

5.1 Use PWL Reference Trajectories for Vehicle Models

Finding a reference trajectory $\xi_{\mathsf{ref}}(t)$ such that (a) $\xi_{\mathsf{ref}}(t)$ satisfies the reach-avoid conditions, and (b) $\xi_{\mathsf{ref}}(t)$ and $u_{\mathsf{ref}}(t)$ are concatenations of state-input trajectory pairs $\{(\xi_{\mathsf{ref},i}, u_{\mathsf{ref},i})\}_i$ and each pair satisfies the system dynamics, is a nontrivial problem. If we were to encode the problem directly as a satisfiability or an optimization problem, the solver would have to search for over the space of continuous functions constrained by the above requirements, including the nonlinear differential constraints imposed by f. The standard tactic is to fix a reasonable template for $\xi_{\mathsf{ref}}(t), u_{\mathsf{ref}}(t)$ and search for instantiations of this template.

From Example 4, we see that if ξ_{ref} is a PWL reference trajectory constructed from waypoints in the workspace, the tracking error can be bounded using Lemma 2. A PWL reference trajectories connecting the waypoints in the workspace have the flexibility to satisfy the reach-avoid requirement. Therefore, in this section, we fix ξ_{ref} and u_{ref} to be the reference trajectory and reference input returned by the `Waypoints_to_Traj`$(\cdot,\cdot)$. In Sect. 5.2, we will see that the problem of finding such PWL $\xi_{\mathsf{ref}}(t)$ can be reduced to a satisfiability problem over quantifier-free linear real arithmetic, which can be solved effectively by off-the-shelf SMT solvers (see Sect. 6 for empirical results).

5.2 Synthesizing Waypoints for a Linear Reference Trajectory

Algorithm 1 says that $\xi_{\mathsf{ref}}(t)$ and $u_{\mathsf{ref}}(t)$ can be uniquely constructed given a sequence of waypoints $\{p_i\}_{i=0}^k$ in the workspace $\mathcal{W}$ and a constant velocity $\bar{v}$. From Proposition 1, $p_{\mathsf{ref}}(t) = \xi_{\mathsf{ref}}(t) \downarrow p$ connects the waypoints in $\mathcal{W}$. Also, let $t_i = \sum_{j=1}^{i} \|p_j - p_{j-1}\|_2/\bar{v}$ be the concatenation time, $\forall t \in [t_{i-1}, t_i)$, $p(t)$ is the line segment connecting p_{i-1} and p_i. We want to ensure that $p(t) = \xi_g(t) \downarrow p$ satisfy the reach-avoid requirements. From Lemma 3, for any $t \in [t_{i-1}, t_i)$, we can bound $\|p(t) - p_{\mathsf{ref}}(t)\|_2$ with the constant ℓ_i, then the remaining problem is to ensure that, $p_{\mathsf{ref}}(t)$ is at least ℓ_i away from the obstacles and $p_{\mathsf{ref}}(\xi_{\mathsf{ref}}.\mathsf{ltime})$ is inside the goal set with ℓ_k distance to any surface of the goal set.

Let us start with one segment $p(t)$ with $t \in [t_{i-1}, t_i)$. To enforce that $p(t)$ is ℓ_i away from a polytope obstacle, a sufficient condition is to enforce both the endpoints of the line segment to lie out at least one surface of the polytope bloated by ℓ_i.

Lemma 4. *If $p_{\mathsf{ref}}(t)$ with $t \in [t_{i-1}, t_i)$ is a line segment connecting p_{i-1} and p_i in $\mathcal{W}$. Given a polytope obstacle $O = Poly(H_O, b_O)$ and $\ell_i > 0$, if*

$$\bigvee_{s=1}^{\mathsf{dP}(H_O)} \left((H_O^{(s)} p_{i-1} > b_O^{(s)} + \|H_O^{(s)}\|_2 \ell_i) \wedge (H_O^{(s)} p_i > b_O^{(s)} + \|H_O^{(s)}\|_2 \ell_i) \right) = \mathsf{True},$$

then $\forall t \in [t_{i-1}, t_i)$, $B_{\ell_i}(p_{\mathsf{ref}}(t)) \cap O = \emptyset$.

Proof. Fix any s such that $(H_O^{(s)} p_{i-1} > b_O^{(s)} + \|H_O^{(s)}\|_2 \ell_i) \wedge (H_O^{(s)} p_i > b_O^{(s)} + \|H_O^{(s)}\|_2 \ell_i)$ holds. The set $S = \{q \in \mathbb{R}^d \mid H_O^{(s)} q > b_O^{(s)} + \|H_O^{(s)}\|_2 \ell_i\}$ defines a convex half space. Therefore, if $p_{i-1} \in S$ and $p_i \in S$, then any point on the line segment connecting p_{i-1} and p_i is in S. Therefore, for any $t \in [t_{i-1}, t_i)$, $H_O^{(s)} p_{\mathsf{ref}}(t) > b_O^{(s)} + \|H_O^{(s)}\|_2 \ell_i > b_O^{(s)}$, which means $p_{\mathsf{ref}}(t) \notin O$.

The distance between $p_{\mathsf{ref}}(t)$ and the surface $H_O^{(s)} q = b_O^{(s)}$ is $\frac{|H_O^{(s)} p_{\mathsf{ref}}(t) - b_O^{(s)}|}{\|H_O^{(s)}\|_2} > \ell_i$. Therefore, for any $p \in B_{\ell_i}(p_{\mathsf{ref}}(t))$ we have $\|p - p_{\mathsf{ref}}(t)\|_2 \leq \ell_i$ and thus $p \notin O$.

Furthermore, $\bigwedge_{s=1}^{\mathsf{dP}(H_O)} H_O^{(s)} q \leq b_O^{(s)} + \|H_O^{(s)}\|_2 \ell_i$ defines of a new polytope that we get by bloating $Poly(H_O, b_O)$ with ℓ_i. Basically, it is constructed by moving each surface of $Poly(H_O, b_O)$ along the surface's normal vector with the direction pointing outside the polytope.

Similarly, we can define the condition when $p_{\mathsf{ref}}(\xi.\mathsf{ltime}) = p_k$ is inside the goal shrunk by ℓ_k.

Lemma 5. *Given a polytope goal set $G = Poly(H_G, b_G)$ and $\ell_k > 0$, if*

$$\bigwedge_{s=1}^{\mathsf{dP}(H_G)} \left(H_G^{(s)} p_k \leq b_O^{(s)} - \|H_G^{(s)}\|_2 \ell_k \right) = \mathsf{True}, \text{ then } B_{\ell_k}(p_k) \subseteq G.$$

Putting them all together, we want to solve the following satisfiability problem to ensure that each line segment between p_{i-1} and p_i is at least ℓ_i away from all the obstacles and p_k is inside the goal set G with at least distance ℓ_k to the surfaces of G. In this way, $\xi_g(t)$ starting from a neighborhood of $\xi_{\mathsf{ref}}(0)$ can satisfy the reach-avoid requirement.

$$\begin{gathered}
\phi_{\mathsf{waypoints}}(p_{\mathsf{ref}}(0), k, \mathbf{O}, G, \{\ell_i\}_{i=1}^k) = \exists p_0, \cdots, p_k, \\
p_0 == p_{\mathsf{ref}}(0) \\
\bigwedge_{s=1}^{\mathsf{dP}(H_G)} \left(H_G^{(s)} p_k \leq b_O^{(s)} - \|H_G^{(s)}\|_2 \ell_k \right) \\
\bigwedge_{i=1}^{k} \left(\bigwedge_{Poly(H,b) \in \mathbf{O}} \left(\bigvee_{s=1}^{\mathsf{dP}(H)} \left(H^{(s)} p_{i-1} > b^{(s)} + \ell_i \|H^{(s)}\|_2 \wedge H^{(s)} p_i > b^{(s)} + \ell_i \|H^{(s)}\|_2 \right) \right) \right)
\end{gathered}$$

Notice that the constraints in $\phi_{\mathsf{waypoints}}$ are all linear over real arithmetic. Moreover, the number of constraints in $\phi_{\mathsf{waypoints}}$ is $O\left(\sum_{Poly(H,b) \in \mathbf{O}} k\mathsf{dP}(H) + \mathsf{dP}(H_G) \right)$. That is, fixing k, the number of constraints will grow linearly with the total number of surfaces in the obstacle and goal set polytopes. Fixing $\mathbf{O}$ and G, the number of constraints will grow linear with the number of line segments k.

Theorem 1. *Fix $k \geq 1$ as the number of line segments, $p_{ref}(0) \in \mathcal{W}$ as the initial position of the reference trajectory. Assume that*

(1) $\mathcal{A}$ closed with g_{ref} and g_{trk} is such that given any sequence of $k+1$ waypoints in $\mathcal{W}$ and any $\bar{v}$, the piece-wise reference ξ_{ref} (and input u_{ref}) returned by Algorithm 1 satisfy the conditions in Lemmas 2 and 3 with Lyapunov function $V(e(t))$ for the tracking error $e(t)$.
(2) For the above ξ_{ref}, fix an ε_0 such that $V(e(0)) \leq \varepsilon_0$, let $\{\ell_i\}_{i=1}^k$ be error bounds for positions constructed using Lemma 2 and Lemma 3 from ε_0.
(3) $\phi_{waypoints}(p_{ref}(0), k, \mathbf{O}, G, \{\ell_i\}_{i=1}^k)$ is satisfiable with waypoints $\{p_i\}_{i=0}^k$.

Let $\xi_{ref}(t), u_{ref}(t) =$ Waypoints_to_Trajectory $(\{p_i\}_{i=0}^k, \bar{v})$, and $p_{ref}(t) = \xi_{ref}(t) \downarrow p$. Let $\xi_g(t)$ be a trajectory of $\mathcal{A}$ closed with $g_{trk}(\cdot, \xi_{ref}, u_{ref})$ starting from $\xi_g(0)$ with $V(e(\xi_g(0), \xi_{ref}(0))) \leq \varepsilon_0$, then $\xi_g(t)$ satisfies the reach-avoid requirement.

Proof. Since $\xi_{\mathsf{ref}}(t), u_{\mathsf{ref}}(t)$ are a PWL reference trajectory and a reference input respectively constructed from the waypoints $\{p_i\}_{i=0}^k$, they satisfy Assumption (1). Moreover, $V(e(\xi_g(0), \xi_{\mathsf{ref}}(0))) \leq \varepsilon_0$ satisfies Assumption (2). Using Lemma 2 and Lemma 3, we know that for $t \in [t_{i-1}, t_i)$, $\|\xi_g(t) \downarrow p - \xi_{\mathsf{ref}}(t) \downarrow p\|_2 \leq \ell_i$.

Finally, since $\{p_i\}_{i=0}^k$ satisfy the constraints in $\phi_{\mathsf{waypoints}}$, using Lemma 4 and Lemma 5, we know that for any time $t \in [0, t_k]$, $\xi_g(t) \downarrow p \notin \mathbf{O}$ and $\xi_g(t_k) \in G$. Therefore the theorem holds.

5.3 Partitioning the Initial Set

Starting from the entire initial set Θ, fix $\xi_{\mathsf{ref}}(0) \in \Theta$ and an ε_0 such that $\forall x \in \Theta, V(e(x, \xi_{\mathsf{ref}}(0))) \leq \varepsilon_0$, then we can use Lemma 2 and Lemma 3 to construct the error bounds $\{\ell_i\}_{i=1}^k$ for positions, and next use $\{\ell_i\}_{i=1}^k$ to solve $\phi_{\mathsf{waypoints}}$ and find the waypoints and construct the reference trajectory.

However, if the initial set Θ is too large, $\{\ell_i\}_{i=1}^k$ could be too conservative so $\phi_{\mathsf{waypoints}}$ is not satisfiable. In the first two figures on the top row of Fig. 3, we could see that if we bloat the obstacle polytopes using the largest ℓ_i, then no reference trajectory is feasible. In this case, we partition the initial set Θ to several smaller covers Θ_j and repeat the above steps from each smaller cover Θ_j. In Lemma 2 and Lemma 3 we could see that the values of $\{\ell_i\}_{i=1}^k$ decrease if ε_0 decreases. Therefore, with the partition of Θ, we could possibly find a reference trajectory more and more easily. As shown in Fig. 3 bottom row, after several partitions, a reference trajectory for each Θ_j could be found.

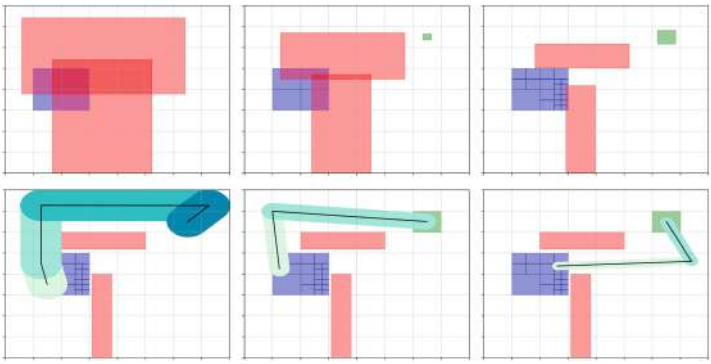


Fig. 3. Top row: each step attempting to find a reference trajectory in the space where obstacles (goal set) are bloated (shrunk) by the error bounds $\{\ell_i\}_i$. From left to right: Without partition, $\{\ell_i\}_i$ are too large so a reference trajectory cannot be found. Θ is partitioned, but $\{\ell_i\}$s for the left-top cover are still too large. With further partions, a reference trajectory could be found. Bottom row: It is shown that the bloated tubes for each cover (which contain all other trajectories from that cover) can fit between the original obstacles.

5.4 Overall Synthesis Algorithm

Taking partitioning into the overall algorithm, we have Algorithm 2 to solve the controller synthesis problem defined in Sect. 2.2. Algorithm 2 takes in as inputs (1) an (n, m)-dimensional control system $\mathcal{A}$, (2) a tracking controller g_{trk}, (3) Obstacles $\mathbf{O}$, (4) a goal set G, (5) a Lyapunov function $V(e(t))$ for the tracking error e that satisfies the conditions in Lemma 2 and Lemma 3 for any PWL reference trajectory and input, (6) the maximum number of line segments allowed $\mathsf{Seg}_{\max}$, (7) the maximum number of partitions allowed $\mathsf{Part}_{\max}$, and (8) a constant velocity $\bar{v}$. The algorithm returns a set `RefTrajs`, such that for each triple $\langle \Theta_j, \xi_{j,\mathsf{ref}}, u_{j,\mathsf{ref}} \rangle \in$ `RefTrajs`, we have $\forall x_0 \in \Theta_j$, the unique trajectory ξ_g of the closed system ($\mathcal{A}$ closed with $g_{\mathsf{trk}}(\cdot, \xi_{j,\mathsf{ref}}, u_{j,\mathsf{ref}})$) starting from x_0 satisfies the reach-avoid requirement. The algorithm also returns $\langle$`Cover`, **None**$\rangle$, which means that the algorithm fails to find controllers for the portion of the initial set in `Cover` within the maximum number of partitions $\mathsf{Part}_{\max}$.

In Algorithm 2, `Cover` is the collection of covers in Θ that the corresponding ξ_{ref} and u_{ref} have not been discovered. Initially, `Cover` only contains Θ. The **for**-loop from Line 2 will try to find a ξ_{ref} and a u_{ref} for each $\Theta \in$ `Cover` until the maximum allowed number for partitions is reached. At line 3, we fix the initial state of $\xi_{\mathsf{ref}}(0) = \xi_{\mathsf{init}}$ to be the center of the current cover Θ. Then at Line 4, we get the initial error bounds ε_0 after fixing ξ_{init}. Using ε_0 and the Lyapunov function $V(e)$, we can construct the error bounds $\{\ell_i\}_{i=1}^k$ for the positions of the vehicle using Lemma 2 and Lemma 3 at Line 5.

Algorithm 2: Controller synthesis algorithm

```
   input  : A = ⟨X, Θ, U, f⟩, g_trk, O, G, V(e(t)), Seg_max, Part_max, v̄
   initially: Cover ← {Θ}, prt ← 0, k ← 1, RefTrajs ← ∅
 1 while (Cover ≠ ∅) ∧ (prt ≤ Part_max) do
 2     for Θ ∈ Cover do
 3         ξ_init ← Center(Θ) ;
 4         ε_0 ← a such that ∀x ∈ Θ, V(e(x, ξ_init)) ≤ a ;
 5         {ℓ_i}_{i=1}^k ← GetBounds(V(e(t)), ε_0) ;
 6         while k ≤ Seg_max do
 7             if CheckSAT(ξ_init ↓ p, k, O, G, {ℓ_i}_{i=1}^k)) == SAT then
 8                 p_0, ··· , p_k ← GetValue(φ_waypoints) ;
 9                 ξ_ref, u_ref ← Waypoints_to_Traj({p_i}_{i=0}^k, v̄) ;
10                 RefTrajs ← RefTrajs ∪ ⟨Θ, ξ_ref, u_ref⟩ ;
11                 Cover ← Cover \ {Θ};
12                 k ← 1 ;
13                 Break ;
14             else
15                 k ← k + 1
16         if k > Seg_max then
17             Cover ← Cover ∪ Partition(Θ) \ {Θ} ;
18             prt ← prt + 1;
19             k ← 1 ;
20 return RefTrajs, ⟨Cover, None⟩ ;
```

If the **if** condition at Line 7 holds with $\{p_i\}_{i=0}^k$ being the waypoints that satisfy $\phi_{\mathsf{waypoints}}$, then from Theorem 1 we know that the $\xi_{\mathsf{ref}}, u_{\mathsf{ref}}$ constructed using $\{p_i\}_{i=0}^k$ at Line 9 will be such that, the unique trajectory ξ_g of the closed system ($\mathcal{A}$ closed with $g_{\mathsf{trk}}(\cdot, \xi_{\mathsf{ref}}, u_{\mathsf{ref}})$) starting from $x_0 \in \Theta$ satisfies the reach-avoid requirement. Otherwise the algorithm will increase the number of segments k in the PWL reference trajectory (Line 15). When the maximum number of line segments allowed is reached but the algorithm still could not find $\xi_{\mathsf{ref}}, u_{\mathsf{ref}}$ that can guarantee the satisfaction of reach-void requirement from the current cover Θ, we will partition the current Θ at Line 17 and add those partitions to `Cover`. At the same time, k will be reset to 1.

Theorem 2 (Soundness). *Suppose the inputs to Algorithm 2, $\mathcal{A}$, g_{trk}, $\mathbf{O}$, G, $V(e(t))$, $\bar{v}$ satisfy the conditions of Theorem 1. Let the output be* $\texttt{RefTrajs} = \{\langle \Theta_j, \xi_{j,ref}, u_{j,ref} \rangle\}_j$ *and* $\langle \texttt{Cover}, \textbf{\textit{None}} \rangle$*, then we have (1).* $\Theta \subseteq \cup \Theta_j \cup \texttt{Cover}$*, and (2). for each triple* $\langle \Theta_j, \xi_{j,ref}, u_{j,ref} \rangle$*, we have* $\forall x_0 \in \Theta_j$*, the unique trajectory* ξ_g *of the closed system ($\mathcal{A}$ closed with* $g_{trk}(\cdot, \xi_{j,ref}, u_{j,ref})$*) starting from* x_0 *satisfies the reach-avoid requirement.*

The theorem follows directly from the proof of Theorem 1.

6 Implementation and Evaluation

We have implemented our synthesis algorithm (Algorithm 2) in a prototype open source tool we call FACTEST[5] (FAst ConTrollEr SynThesis framework). Our

[5] All models and source code of FACTEST are available at [27].

implementation uses Pypoman[6], Yices 2.2 [6], SciPy[7] and NumPy[8] libraries. The inputs to FACTEST are the same as the inputs in Algorithm 2. FACTEST terminates in two ways. Either it finds a reference trajectory $\xi_{j,\mathsf{ref}}$ and reference input $u_{j,\mathsf{ref}}$ for every partition Θ_j of Θ so that Theorem 2 guarantees they solved the controller synthesis problem. Otherwise, it terminates by failing to find reference trajectories for at least one subset of Θ after partitioning Θ up to the maximum specified depth.

6.1 Benchmark Scenarios: Vehicle Models and Workspaces

We will report on evaluating FACTEST in several 2D and 3D scenarios drawn from motion planning literature (see Figs. 4). Recall, the state space $\mathcal{X}$ dimension corresponds to the vehicle model, and is separate from the dimensionality of the workspace $\mathcal{W}$. We will use four nonlinear vehicle models in these different scenarios: (a) the kinematic vehicle model (car) [31] introduced in Example 1, (b) a bijective mobile robot (robot) [13], (c) a hovering robot (hovercraft), and (d) an autonomous underwater vehicle (AUV) [29]. The dynamics and tracking controllers (g_{trk}) of the other three models are described on the FACTEST website [27]. Each of these controllers come with a Lyapunov function that meets the assumptions of Lemmas 2 and 3 so the tracking error bounds given by the lemmas $\{\ell\}_{i=1}^{k}$ can be computed.

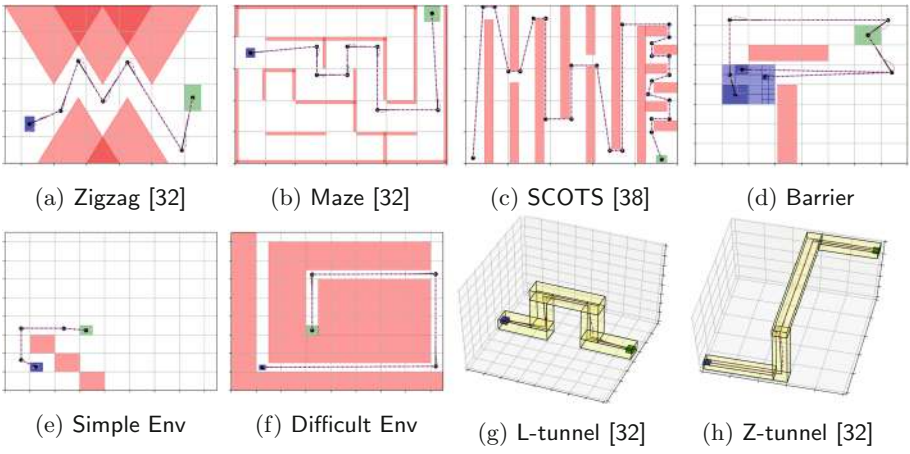


Fig. 4. 2D and 3D workspaces with initial (blue) and goal (green) sets. The scenarios run in the two-dimensional $\mathcal{W}$ use the car model. The scenarios run in the three dimensional $\mathcal{W}$ use the hovercraft model. The black lines denote ξ_{ref} and the dotted violet lines denote ξ_g. (Color figure online)

[6] https://pypi.org/project/pypoman/.
[7] https://www.scipy.org/.
[8] https://numpy.org/.

6.2 Synthesis Performance

Table 1 presents the performance of FACTEST on several synthesis problems. Several points are worth highlighting. (a) The absolute running time is at the sub-second range, even for 6-dimensional vehicle models with 4-inputs, operating in a 3D workspace. This is encouraging for online motion-control applications with dynamic obstacles. (b) The running time is not too sensitive to dimensions of $\mathcal{X}$ and $\mathbf{U}$ because the waypoints are only being generated in the lower dimensional workspace $\mathcal{W}$. Additionally, the construction of ξ_{ref} from the waypoints does not add significant time. However, since different models have different dynamics and Lypunov functions, they would have different error bounds for position. Such different bound could influence the final result. For example, the result for the Barrier scenario differs between the car and the robot. The car required 25 partitions to find a solution over all of Θ and the robot required 22. (c) Confirming what we have seen in Sect. 5.2, the runtime of the algorithm scales with the number of segments required to solve the scenario and the number of obstacles. (d) As expected and seen in Zigzag scenarios, all other things being the same, the running time and the number of partitions grow with larger initial set uncertainty.

Table 1. Synthesis performance on different scenarios (environment, vehicle). Dimension of state space $\mathcal{X}(n)$, input (m), radius of initial set Θ, number of obstacles $\mathbf{O}$, running time (in seconds).

Scenario	n, m	Radius of Θ	# $\mathbf{O}$	Time (s)	# segments per ξ_{ref}	# partitions
Zigzag, car 1	3, 2	0.200	9	0.037	6.0	1.0
Zigzag, car 2	3, 2	0.400	9	0.212	4.0	6.0
Zigzag, car 3	3, 2	0.800	9	0.915	5.0–6.0	16.0
Zigzag, robot 1	4, 2	0.200	9	0.038	6.0	1.0
Zigzag, robot 2	4, 2	0.400	9	0.227	4.0	6.0
Zigzag, robot 3	4, 2	0.800	9	0.911	5.0–6.0	16.0
Barrier car	3, 2	0.707	6	0.697	2.0–4.0	25.0
Barrier, robot	4, 2	0.707	6	0.645	2.0–4.0	22.0
Maze, car	3, 2	0.200	22	0.174	8.0	1.0
Maze, robot	4, 2	0.200	22	0.180	8.0	1.0
SCOTS, car	3, 2	0.070	19	1.541	26.0	1.0
SCOTS, robot	4, 2	0.070	19	1.623	26.0	1.0
L-tunnel, hovercraft	4, 3	0.173	10	0.060	5.0	1.0
L-tunnel, AUV	6, 4	1.732	10	0.063	5.0	1.0
Z-tunnel, hovercraft	4, 3	0.173	5	0.029	4.0	1.0
Z-tunnel, AUV	6, 4	1.732	10	0.029	4.0	1.0

Comparison with Other Motion Controller Synthesis Tools: A Challenge. Few controller synthesis tools for nonlinear models are available for direct comparisons. We had detailed discussions with the authors of FastTrack [11],

but found it difficult to plug-in new vehicle models. RTD [44] is implemented in MatLab also for specific vehicle models. Pessoa [26] and SCOTS [38] are implemented as general purpose tools. However, they are based on construction of discrete abstractions, which requires several additional user inputs. Therefore, we were only able to compare FACTEST with SCOTS and Pessoa using the scenario SCOTS. This scenario was originally built in SCOTS and is using the same car model.

The results for SCOTS and Pessoa can be found in [38]. The total runtime of SCOTS consists of the abstraction time t_{abs} and the synthesis time t_{syn}. The Pessoa tool has an abstraction time of $t_{\text{abs}} = 13509\,\text{s}$ and a synthesis time of $t_{\text{syn}} = 535\,\text{s}$, which gives a total time of $t_{\text{tot}} = 14044\,\text{s}$. The SCOTS tool has a has an abstraction time of $t_{\text{abs}} = 100\,\text{s}$ and a synthesis time of $t_{\text{syn}} = 413\,\text{s}$, which gives a total time of $t_{\text{tot}} = 513\,\text{s}$. FACTEST clearly outperforms both SCOTS and Pessoa with a total runtime of $t_{\text{tot}} = 1.541\,\text{s}$. This could be attributed to the fact that FACTEST does not have to perform any abstractions, but even by looking sole at t_{syn}, FACTEST is significantly faster. However, we do note that the inputs of FACTEST and SCOTS are different. For example, SCOTS needs a growth bound function β for the dynamics but FACTEST requires Lyapunov functions for the tracking error.

6.3 RRT vs. SAT-Plan

To demonstrate the speed of our SAT-based reference trajectory synthesis algorithm (i.e. only the **while**-loop from Line 6 to Line 15 of Algorithm 2 which we call SAT-Plan), we compare it with Rapidly-exploring Random Trees (RRT) [20]. The running time, number of line segments, and number of iterations needed to find a path were compared. RRT was run using the Python Robotics library [39], which is not necessarily an optimized implementation. SAT-Plan was run using Yices 2.2. The scenarios are displayed in Fig. 4 and the results are in Fig. 5.

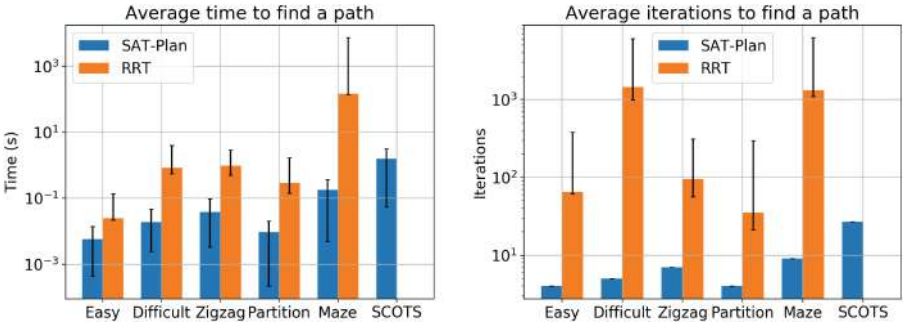


Fig. 5. Comparison of RRT and SAT-Plan. The left plot shows the runtime and the right plot shows the number of necessary iterations. Note that RRT timed out on the SCOTS scenario.

Each planner was run 100 times. The colored bars represent the average runtime and average number of iterations. The error bars represent the range of minimum and maximum. The RRT path planner was given a maximum of 5000 iterations and a path resolution of 0.01. SAT-Plan was given a maximum of 100 line segments to find a path. RRT timed out for the SCOTS scenario, unable to find a trajectory within 5000 iterations. The maze scenario timed out about 10% of the time.

Overall SAT-Plan scales in time much better as the size of the unsafe set increases. Additionally, the maximum number of iterations that RRT had to perform was far greater than the average number of line segments needed to find a safe path. This means that the maximum number of iterations that RRT must go through must be sufficiently large, or else a safe path will not be found even if one exists. SAT-Plan does not have randomness and therefore will find a reference trajectory (with k segments) in the modified space (bloated obstacles and shrunk goal) if one (with k segments) exists. Various examples of solutions found by RRT and SAT-Plan can be found on the FACTEST's website [27].

7 Conclusion and Discussion

We introduced a technique for synthesizing correct-by-construction controllers for a nonlinear vehicle models, including ground, underwater, and aerial vehicles, for reach-avoid requirements. Our tool FACTEST implementing this technique shows very encouraging performance on various vehicle models in different 2D and 3D scenarios.

There are several directions for future investigations. (1) One could explore a broader class of reference trajectories to reduce the tracking error bounds. (2) It would also be useful to extend the technique so the synthesized controller can satisfy the actuation constraints automatically. (3) Currently we require user to provide the tracking controller g_{trk} with the Lyapunov functions, it would be interesting to further automate this step.

References

1. Ames, A.D., Coogan, S., Egerstedt, M., Notomista, G., Sreenath, K., Tabuada, P.: Control barrier functions: theory and applications. In: 2019 18th European Control Conference (ECC), pp. 3420–3431. IEEE (2019)
2. Ardakani, M.M.G., Olofsson, B., Robertsson, A., Johansson, R.: Real-time trajectory generation using model predictive control. In: IEEE International Conference on Automation Science and Engineering, pp. 942–948. IEEE (2015)
3. Åström, K.J., Murray, R.M.: Feedback Systems: An Introduction for Scientists and Engineers. Princeton University Press, Princeton (2010)
4. Bemporad, A., Borrelli, F., Morari, M.: Model predictive control based on linear programming - the explicit solution. IEEE Trans. Autom. Control **47**(12), 1974–1985 (2002)
5. Chitta, S., Sucan, I., Cousins, S.: Moveit![ROS topics]. IEEE Robot. Autom. Mag. **19**(1), 18–19 (2012)

6. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_49
7. Fan, C., Mathur, U., Mitra, S., Viswanathan, M.: Controller synthesis made real: reach-avoid specifications and linear dynamics. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 347–366. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_19
8. Mendes Filho, J.M., Lucet, E., Filliat, D.: Real-time distributed receding horizon motion planning and control for mobile multi-robot dynamic systems. In: International Conference on Robotics and Automation, pp. 657–663. IEEE (2017)
9. Filippidis, I., Dathathri, S., Livingston, S.C., Ozay, N., Murray, R.M.: Control design for hybrid systems with tulip: the temporal logic planning toolbox. In: IEEE Conference on Control Applications, pp. 1030–1041 (2016)
10. Girard, A.: Controller synthesis for safety and reachability via approximate bisimulation. Automatica **48**(5), 947–953 (2012)
11. Herbert, S.L., Chen, M., Han, S.J., Bansal, S., Fisac, J.F., Tomlin, C.J.: FaSTrack: a modular framework for fast and guaranteed safe motion planning. In: 2017 IEEE 56th Annual Conference on Decision and Control (CDC), pp. 1517–1522. IEEE (2017)
12. Janson, L., Schmerling, E., Clark, A., Pavone, M.: Fast marching tree: a fast marching sampling-based method for optimal motion planning in many dimensions. Int. J. Robot. Res. **34**(7), 883–921 (2015)
13. Kanayama, Y., Kimura, Y., Miyazaki, F., Noguchi, T.: A stable tracking control method for an autonomous mobile robot. In: Proceedings of the IEEE International Conference on Robotics and Automation, pp. 384–389. IEEE (1990)
14. Karaman, S., Frazzoli, E.: Incremental sampling-based algorithms for optimal motion planning. In: Robotics Science and Systems VI, vol. 104, no. 2 (2010)
15. Kavraki, L.E., Svestka, P., Latombe, J.-C., Overmars, M.H.: Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Trans. Robot. Autom. **12**(4), 566–580 (1996)
16. Khalil, H.K., Grizzle, J.W.: Nonlinear Systems, vol. 3. Prentice Hall, Upper Saddle River (2002)
17. Kloetzer, M., Belta, C.: A fully automated framework for control of linear systems from temporal logic specifications. IEEE Trans. Autom. Control **53**(1), 287–297 (2008)
18. Kress-Gazit, H., Fainekos, G.E., Pappas, G.J.: Temporal logic based reactive mission and motion planning. IEEE Trans. Robot. **25**(6), 1370–1381 (2009)
19. Kuffner, J.J., LaValle, S.M.: RRT-connect: an efficient approach to single-query path planning. In: IEEE International Conference on Robotics and Automation, vol. 2, pp. 995–1001. IEEE (2000)
20. LaValle, S.M.: Rapidly-exploring random trees: a new tool for path planning (1998)
21. LaValle, S.M.: Planning Algorithms. Cambridge University Press, Cambridge (2006)
22. Liu, C., Lee, S., Varnhagen, S., Eric Tseng, H.: Path planning for autonomous vehicles using model predictive control. In: IEEE Intelligent Vehicles Symposium, pp. 174–179. IEEE (2017)
23. Majumdar, A., Tedrake, R.: Funnel libraries for real-time robust feedback motion planning. Int. J. Robot. Res. **36**(8), 947–982 (2017)
24. Mallik, K., Schmuck, A.-K., Soudjani, S., Majumdar, R.: Compositional synthesis of finite-state abstractions. IEEE Trans. Autom. Control **64**(6), 2629–2636 (2018)

25. Mayne, D.Q.: Model predictive control: recent developments and future promise. Automatica **50**, 2967–2986 (2014)
26. Mazo, M., Davitian, A., Tabuada, P.: PESSOA: a tool for embedded controller synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 566–569. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_49
27. Miller, K., Fan, C., Mitra, S.: Factest webpage (2020). https://kmmille.github.io/FACTEST/index.html. Accessed 13 May 2020
28. Mouelhi, S., Girard, A., Gössler, G.: CoSyMA: a tool for controller synthesis using multi-scale abstractions. In: International Conference on Hybrid Systems: Computation and Control, pp. 83–88. ACM (2013)
29. Nakamura, Y., Savant, S.: Nonlinear tracking control of autonomous underwater vehicles. In: Proceedings 1992 IEEE International Conference on Robotics and Automation, pp. A4–A9. IEEE (1992)
30. Ogata, K., Yang, Y.: Modern Control Engineering, vol. 5. Prentice Hall, Upper Saddle River (2010)
31. Paden, B., Čáp, M., Yong, S.Z., Yershov, D., Frazzoli, E.: A survey of motion planning and control techniques for self-driving urban vehicles. IEEE Trans. Intell. Veh. **1**(1), 33–55 (2016)
32. Texas A&M University Parasol MP Group, CSE Department Algorithms & applications group benchmarks
33. Raman, V., Donzé, A., Maasoumy, M., Murray, R.M., Sangiovanni-Vincentelli, A., Seshia, S.A.: Model predictive control with signal temporal logic specifications. In: 2014 IEEE 53rd Annual Conference on Decision and Control (CDC), pp. 81–87. IEEE (2014)
34. Raman, V., Donzé, A., Sadigh, D., Murray, R.M., Seshia, S.A.: Reactive synthesis from signal temporal logic specifications. In: International Conference on Hybrid Systems: Computation and Control, pp. 239–248. ACM (2015)
35. Ravanbakhsh, H., Sankaranarayanan, S.: Robust controller synthesis of switched systems using counterexample guided framework. In: 2016 International Conference on Embedded Software (EMSOFT), pp. 1–10. IEEE (2016)
36. Richter, S., Jones, C.N., Morari, M.: Computational complexity certification for real-time MPC with input constraints based on the fast gradient method. IEEE Trans. Autom. Control **57**(6), 1391–1403 (2011)
37. Roy, P., Tabuada, P., Majumdar, R.: Pessoa 2.0: a controller synthesis tool for cyber-physical systems. In: International Conference on Hybrid Systems: Computation and Control, pp. 315–316. ACM (2011)
38. Rungger, M., Zamani, M.: SCOTS: a tool for the synthesis of symbolic controllers. In: Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, pp. 99–104 (2016)
39. Sakai, A., Ingram, D., Dinius, J., Chawla, K., Raffin, A., Paques, A.: PythonRobotics: a python code collection of robotics algorithms (2018)
40. Singh, S., Majumdar, A., Slotine, J.-J., Pavone, M.: Robust online motion planning via contraction theory and convex optimization. In: 2017 IEEE International Conference on Robotics and Automation (ICRA), pp. 5883–5890. IEEE (2017)
41. Sucan, I.A., Moll, M., Kavraki, L.E.: The open motion planning library. IEEE Robot. Autom. Mag. **19**(4), 72–82 (2012)
42. Tabuada, P.: Verification and Control of Hybrid Systems - A Symbolic Approach. Springer, Heidelberg (2009). https://doi.org/10.1007/978-1-4419-0224-5
43. Tedrake, R.: LQR-trees: feedback motion planning on sparse randomized trees (2009)

44. Vaskov, S., et al.: Towards provably not-at-fault control of autonomous robots in arbitrary dynamic environments. arXiv preprint arXiv:1902.02851 (2019)
45. Vitus, M., Pradeep, V., Hoffmann, G., Waslander, S., Tomlin, C.: Tunnel-MILP: path planning with sequential convex polytopes. In: AIAA Guidance, Navigation and Control Conference and Exhibit, p. 7132 (2008)
46. Wong, K.W., Finucane, C., Kress-Gazit, H.: Provably-correct robot control with LTLMoP, OMPL and ROS. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, p. 2073 (2013)
47. Wongpiromsarn, T., Topcu, U., Murray, R.M.: Receding horizon temporal logic planning. IEEE Trans. Autom. Control **57**(11), 2817–2830 (2012)
48. Wongpiromsarn, T., Topcu, U., Ozay, N., Xu, H., Murray, R.M.: Tulip: a software toolbox for receding horizon temporal logic planning. In: International Conference on Hybrid Systems: Computation and Control, pp. 313–314. ACM (2011)
49. Zeilinger, M.N., Jones, C.N., Morari, M.: Real-time suboptimal model predictive control using a combination of explicit MPC and online optimization. IEEE Trans. Autom. Control **56**(7), 1524–1534 (2011)

SeQuaiA: A Scalable Tool for Semi-Quantitative Analysis of Chemical Reaction Networks

Milan Češka[1(✉)], Calvin Chau[2], and Jan Křetínský[2]

[1] Brno University of Technology, Brno, Czech Republic
ceskam@fit.vutbr.cz
[2] Technical University of Munich, Munich, Germany

Abstract. Chemical reaction networks (CRNs) play a fundamental role in analysis and design of biochemical systems. They induce continuous-time stochastic systems, whose analysis is a computationally intensive task. We present a tool that implements the recently proposed semi-quantitative analysis of CRN. Compared to the proposed theory, the tool implements the analysis so that it is more flexible and more precise. Further, its GUI offers a wide range of visualization procedures that facilitate the interpretation of the analysis results as well as guidance to refine the analysis. Finally, we define and implement a new notion of "mean" simulations, summarizing the typical behaviours of the system in a way directly comparable to standard simulations produced by other tools.

1 Introduction

Chemical Reaction Networks (CRNs) are a language widely used for *modelling and analysis* of biochemical systems [10] as well as for high-level programming of molecular devices [6,33]. They provide a compact formalism equivalent to Petri nets [30], vector addition systems [24] and distributed population protocols [3]. A CRN consists of a set of chemical reactions of given species, each running at a certain rate (intuitively, speed).

Example 1 (Gene expression). Our running example is the classic simple expression of a protein given by the reactions of production (p) and degradation (d) of proteins and blocking (b) the DNA, over three species: protein (P), active DNA ($\mathrm{DNA_{on}}$), and blocked DNA ($\mathrm{DNA_{off}}$):

$$\text{p: } \mathrm{D_{on}} \xrightarrow{10} \mathrm{D_{on}} + \mathrm{P} \qquad \text{d: } \mathrm{P} \xrightarrow{0.1} \emptyset \qquad \text{b: } \mathrm{D_{on}} + \mathrm{P} \xrightarrow{0.001} \mathrm{D_{off}}$$

Using mass-action kinetics (the reaction rate is multiplied by the populations of the reactants), the CRN induces a infinite population Markov chain in Fig. 1.

This work has been partially supported by the Czech Science Foundation grant GJ20-02328Y and German Research Foundation (DFG) project 383882557 *Statistical Unbounded Verification* (KR 4890/2-1).

S. K. Lahiri and C. Wang (Eds.): CAV 2020, LNCS 12224, pp. 653–666, 2020.
https://doi.org/10.1007/978-3-030-53288-8_32

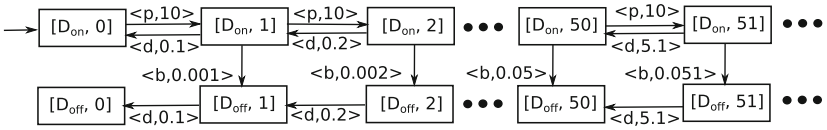


Fig. 1. The Markov chain for Gene expression, displaying the population of P. To simplify the exposition, D_{on} and D_{off} are displayed as discrete "states" of the system, but in fact the two "states" are just shorthands for 1,0 and 0,1, respectively.

In order to facilitate numerous applications in systems and synthetic biology, various techniques for simulation and formal analysis of CRNs have been proposed, e.g. [2,7,15,18,32]. We pinpoint several specifics of this setting, necessary to motivate and understand the features of the tool:

1. The analysis is notoriously difficult and **computationally expensive** due to several aspects: *state-space explosion* (exponential growth in the number of species, possibly infinite spaces due to unbounded populations as in Fig. 1, different rates for different populations, again as in Fig. 1), *stochasticity* (races between reactions), *stiffness* (rates of different magnitudes), *multimodality* (qualitatively different behaviours such as extinction of predators only, or also of preys in the predator-prey models) [17,34]. Consequently, even for small CRNs, simulations may take minutes and analyses hours.
2. We have to face **imprecise inputs**. In particular, even if all relevant reactions are known, the rates are typically not. It is then not clear what behaviours can be induced by all possible values.
3. The analysis **output need not be precise** numerically, but only qualitatively. For instance, it is important to know that initial growth is followed by extinction and what the order of magnitude of the peak population is, but not necessarily what the exact distribution at an exact time is. Unfortunately, it is hard to compute the qualitative information without the quantitative one.
4. Biologists and engineers often seek for plausible **explanations** of why the system under study features or not the discussed behaviour. In many cases, a set of system simulations/trajectories or population distributions is not sufficient and the ability to provide an accurate explanation for the temporal or steady-state behaviour is another major challenge for the existing techniques.

SeQuaiA[1] is a tool for analysis of CRN addressing these issues:

1. It features unprecedented **scalability**, analysing standard complex benchmarks within a fraction of a second.
2. It is **robust** w.r.t. concrete rates, not depending on the exact values but only on their orders of magnitude.
3. Its *semi-quantitative analysis* is **precise enough** to conclude on the qualitative behaviour of the system including rare behaviours and on rough estimates of the quantities (population sizes, times).

[1] Available at https://sequaia.model.in.tum.de.

4. It produces small abstract models (Markov chains) that are explicit, yet **interpretable**, making the behaviour more **explainable**.

It is based on the technique presented in [9], relying on two cornerstones. Firstly, it computes a system abstraction with **acceleration**, abstracting not only states and single transitions, but taking into account *segments* of paths. The resulting models are small enough to allow for a synoptic observation of the model dynamics. Secondly, it performs **semi-quantitative analysis**, focusing on the most probable behaviours and more qualitative, global descriptions, such as oscillation, rather than fully quantitative sequences of exact transient distributions. This yields explainable models and is a sufficient and computationally cheaper technique. While the basic theory is derived from [9], there are a number of new features and differences in our tool, not just the implementation:

Method: (i) The abstraction is *more precise* now that the tool can also compute numerical outputs, whereas [9] focuses on a manually feasible, and hence imprecise, abstraction. (ii) It suggests how to *refine the abstractions*, providing a knob for trading precision for computational resources.

Visualization: The GUI provides a number of ways to display the results, facilitating understanding the models, including (i) identification of strongly connected parts of 'iterations', corresponding to 'temporarily stable' behaviours, (ii) quantitative information on transient times and steady-state distributions, or (iii) visual qualitative explanations, such as semantic grouping of states or tracking correlations between populations.

Additional analysis instruments: (i) The new notion of *envelope* provides an explicit knob to consider not only the most probable, but also less probable behaviours. (ii) The novel concept of *mean simulation* yields summaries of most probable runs and an analysis output directly comparable to classic simulation-based tools.

Related Work. Since a direct analysis of the Markov chains induced by CRN does not scale well [19], deterministic approximations through fluid (mean-field) techniques can be applied [4,8] to large populations, but cannot adequately capture the stochasticity of CRNs caused by low population species. To this end, both can be combined in hybrid approaches [7,18,21], typically involving a computationally demanding numerical analysis. Reduction techniques such as [1,12] are based on approximate bisimulation [11], on aggregation according to the CRN-specific structure [13,27,35], or state truncation [20,28,29].

Despite the plethora of techniques, the practical analysis of CRNs often relies on the stochastic simulation [15] and its multi-scale improvements [5,14,17,22,31,32]. The widely used tool include the platform-independent Copasi [23], DSD [25] with a convenient web-based graphical interface, or StochPy [26] easily extensible using Python scientific libraries. In contrast, our approach (i) provides a compact explanation of the system behaviour in the form of tiny models allowing for a synoptic observation (ii) can easily reveal less probable behaviours, and iii) as shown in [9], is able to analyse standard complex benchmarks in seconds

and thus provides the unprecedented scalability compared to other numerical as well as simulation-based techniques.

2 Workflow and Key Functionality

In this section, we guide the reader through the workflow, discuss the key features of the tool and demonstrate them on examples. The GUI is structured into several tabs and panels reflecting the workflow of the tool. First, a CRN is either retrieved from a file in the Open model tab or a new one is created. Either way, the model can be changed in the Editor panel together with the analysis parameters. The process continues in the Analysis tab. The analysis follows in two steps. First, the *semi-quantitative abstraction* of the Markov chain for the CRN is generated; second, the *semi-quantitative* analysis is performed on the abstraction. The tool offers an explicit option to display the abstraction as a .dot file or to directly run both steps. After the complete analysis is executed, the Visualization panel offers a range of options to *display* the results, including various *quantitative properties*. Finally, the analysed model can be used to generate concrete runs on the Simulation tab, which we call *mean simulations* since they display the "average-case" behaviour. In the following we detail on these key elements.

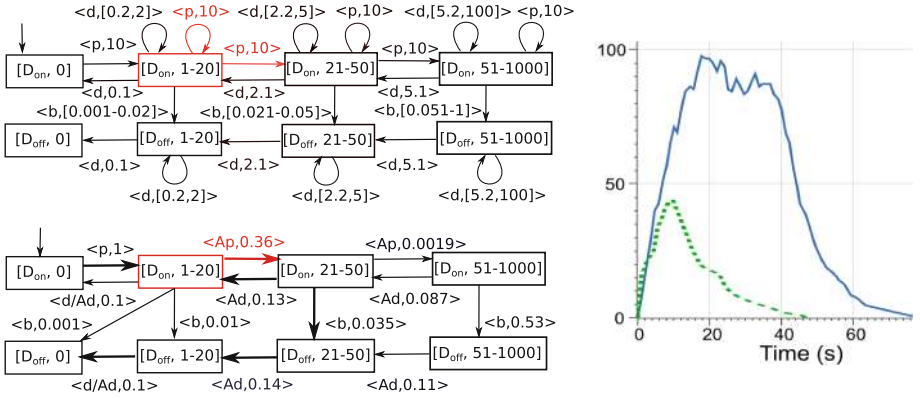


Fig. 2. Left: The abstract Markov chain for Gene expression with population discretization thresholds 20, 50 and the population bound 1000. *Top:* The classic may transition function. *Bottom:* The semi-quantitative version with accelerated transitions (denoted by prefix "A"). **Right:** The full blue line shows a typical simulation of the model (population of P), obtained using DSD tool [25]. The dotted green line corresponds to the fast variant of the model with the rate of b being 10^{-2}. (Color figure online)

2.1 Semi-quantitative Abstraction

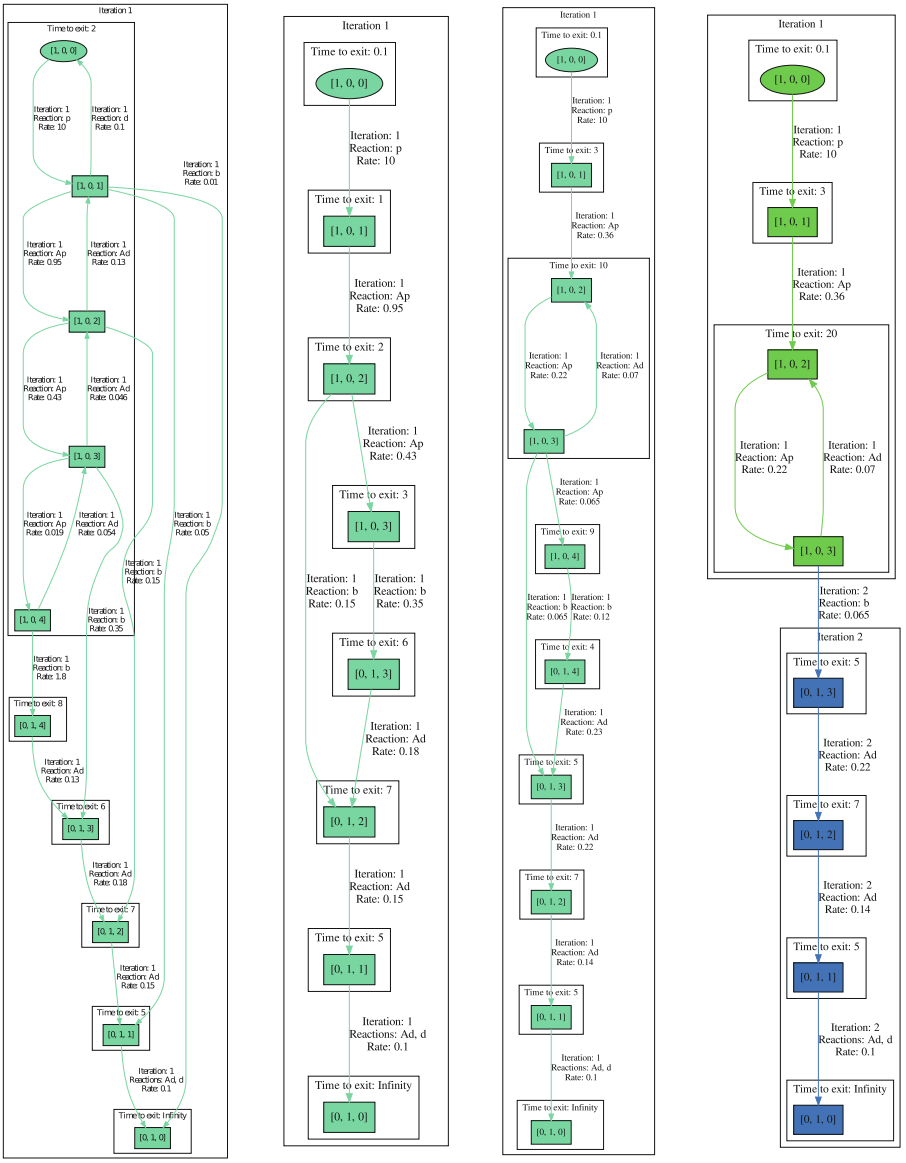
Key Idea. The abstraction of the state space is simply given by a discretization of the population for each species into finitely many intervals, see Fig. 2 (left). The classic may abstraction of the transition function results in non-deterministic self-loops as in Fig. 2 (left top) in red, which make impossible to conclude anything useful (except for some safety properties) on the behaviour once we reach such a state, even whether it is ever left at all. Instead, [9] considers sequences of transitions: in this case, sequences of prevalently growing transitions (those increasing the population) are significantly more probable than the prevalently decreasing ones. Consequently, the self-looping transitions are *accelerated* (taken multiple times) to get a "combined" transition that brings a typical representative of this population interval into a higher interval, see Fig. 2 (left bottom) also in red. Hence the new rate reflects (i) the mass-action kinetics with the typical population in the interval and (ii) the typical number of the transition repetitions before another interval is reached. These accelerated transitions are the key idea of the semi-quantitative abstraction and are denoted by a prefix A.

Tool Inputs. Technically, the tool requires, for each species, a (possible empty) list of increasing population thresholds $t_1, t_2, \ldots t_n$ and a population bound t_b. The thresholds split the concrete population to the intervals $[0,0], (0,t_1], (t_1,t_2], \ldots (t_{n-1},t_n], (t_n,\infty)$. Here 0 is taken separately to reflect enabledness of actions; the representatives, used for consequent computations, are chosen to be in the middle of the intervals and derived from t_b for the last one. (For the empty list we have only one non-zero interval $(0,\infty)$). The input numbers are supposed to reflect the monitored property of interest and the required precision, the bound t_b should give a probable upper bound on the maximal population. How to obtain and iteratively improve these is discussed in Sect. 2.5 on refinement.

Example 2. Consider Gene expression, now with a 'fast' blocking where the rate of b equals 10^{-2}. A typical simulation can be seen in Fig. 2 (right, dotted green line): the number of proteins grows until several dozen, then blocking takes place until extinction. The semi-quantitative abstraction for thresholds $10, 20, 50$ yields the model in Fig. 3(a). In contrast to classic abstractions, there are no self-loops and the abstract transitions are assigned concrete rates. One can see that the blocking can in principle take place at any population and that population can decrease also when DNA is on, i.e. in states $[1,0,\cdot]$. However, all this happens with very low probabilities and the model captures this only indirectly through the numerical labelling. This is made explicit during the semi-quantitative analysis.

2.2 Semi-quantitative Analysis

Key Idea. The aim is to prune the abstraction so that only reasonably probable behaviour is reflected, see the thick transitions in the abstraction in Fig. 2 (left bottom). To this end, we preserve in each state only the transitions with the



(a) (b) (c) (d)

Fig. 3. (a) and (b): 'Fast' Gene expression with thresholds $10, 20, 50$. (a) depicts the full abstraction and (b) depicts *envelope* $= 3$. (c)–(e): 'Slow' Gene expression with thresholds $20, 50, 80, 150$. (d) and (e) depicts the pruned abstraction with *envelope* $= 3$ and 1, respectively.

highest rate h or almost highest rates, i.e. with $h' > h/\mathit{envelope}$ where $\mathit{envelope} > 1$ is a parameter. Parameter values in $[1, 10]$ ensure we can only look at rates of the same order of magnitude, thus the most probable events and those with e.g. only 20% chance of happening. Higher values then allow for inspection of even less probable behaviours.

Consequently, the method can naturally handle uncertainty in the reaction rates since typically only the relative magnitudes of the rates are important, actually, only their orders of magnitude. This robustness w.r.t. the input is very beneficial for biologists as the precise rates are often not known.

Example 3. The analysis of the previous 'fast' Gene expression with $\mathit{envelope} = 3$ is depicted in Fig. 3(b). As such it shows the most probable behaviours: the fast growth until the intervals 2 and 3 (i.e. 10–20 and 20–50) and not beyond to 4 (over 50), followed by a slower decline. The computed rates induce expected times to pass through a state, matching closely those of the simulation Fig. 2 (right, dotted green line). Moreover, we see that the blocking transition from interval 2 has a lower probability than the production, is thus less probable. As such it would not even appear as a probable one, for a stricter $\mathit{envelope} = 2$.

Example 4. A more complicated behaviour arises when the blocking is slow, with rate 10^{-3} as in Sect. 1. A simulation run for this case is depicted in Fig. 2 (right, full blue line). One can observe a more balanced competition between blocking and oscillation around 70–100 proteins. Similarly, while the full abstraction (not shown here) features arbitrary oscillations (also back to no proteins at all), after analysis the pruned abstraction is faithfully modelling the initial growth, subsequent oscillation only in the range of higher populations, followed by blocking and gradual extinction of proteins, see Fig. 3(c).

Technically, the analysis relies on repeated alternation of transient and steady-state analysis. First, starting from the initial state, we follow in each state only the transitions with highest rates (most probable ones), until the set of explored state reaches a fixpoint. A part of the created graph is recurrent and forms a bottom strongly connected component (BSCC) or a collection thereof. The system temporarily settles in the steady state of this BSCC. After some time has passed, also a less probable transition happens almost surely and the "BSCC" is exited. These exit points are identified by a steady-state analysis of the BSCC, taking the magnitudes of exiting and non-exiting transition rates into account. The exit points trigger a new *iteration* of the transient and then the steady-state analysis.

Example 5. Figure 3(d) illustrates a situation with two iteration using the slow variant of the model. Decreasing *envelope* to 1 caused that the blocking reaction is explored in the second iteration – as an exit of the BSCC found in the first iteration. Before that exit happens, the "BSCC" represents a "temporary" steady state of the system.

Note on Correctness. As discussed in [9], the semi-quantitative analysis provides guarantees in the form of limit behaviour and convergence: firstly, the precision grows with the differences in the orders of magnitudes of involved rates: as their ratios tend to infinity, the error tends to zero; secondly, as the population discretization gets finer, the error in the new "accelerated" transitions is reduced, trivially being zero for complete refinement into singletons.

2.3 Visualization of Qualitative Information

A proper visualization is essential for clear presentation and easy interpretation of the results of our analysis. To this end, the tool and its GUI offer various options for visualizing the results. The basic ones, related to the graph structure, are the following. Further options, with more quantitative flavour, are discussed in the next section, followed by an example illustrating all of them.

Iterations. As the complete abstract model is typically very large and chaotic, further structuring is necessary. Therefore, the default view shows the states arranged and grouped into separate blocks, one for each iteration, additionally coloured distinctly for each iteration. Besides, we can restrict which iterations we show. This is useful to zoom in and investigate a particular part of the behaviour.

Intra-iteration SCCs (IISCCs). Additionally, the arrangement and colouring can be based on aggregating SCCs *within* each iteration (IISCCs). This helps to understand the emergence of repetitive behaviour patterns, such as oscillation or (temporary) steady state. It can be also combined with the iteration grouping.

Collapsed Views. In order to understand the system behaviour, one typically needs to have a synoptic overview of the system. For more complex systems, even the pruned abstraction could become too large and the view of the fully expanded system might not be sufficiently compact. In such cases, the aggregates discussed in the previous views, i.e., iterations and IISCCs, can be collapsed into a single nodes, hiding the complexity of the exact behaviour pattern within these areas. This allows us, for instance, to ignore the particular (temporary) oscillation or steady state in these states and to focus on more global behaviour, such as what happened before and after this behaviour and how often does it arise. In contrast to zooming in by restricting to certain iteration(s) only, the collapsed views provide a means to zoom out.

2.4 Visualization of Quantitative Information

The produced graphs are also labelled by *numerical information*. While the quantities cannot be precise due to the simplifications of the extremely scalable analysis, they match the orders of magnitudes of the observed quantities, which is often precise enough for biological purposes; for instance, the peak of protein growth happens after units vs. dozens of seconds in the fast and slow variants of Gene expression, respectively.

Transient Analysis. Firstly, each abstract transition is labelled with a rate corresponding (in the order of magnitude) to the rate of the concrete transition (or accelerated transition, i.e. a "sequence" of transitions) of a "typical" representative of the abstract state. These rates induce the expected time spent in each transient state of each iteration. Indeed, the waiting time is simply the inverse of the sum of the outgoing rates. Further, each BSCC of each iteration is labelled by an estimate of time before it is left into the next iteration. This is a key notion, which allows us to easily provide transient timing information for very stiff systems (working at different time scales). Consider the simple gene model. From Fig. 3(b) and (d) we can easily compute the expected time to the extinction (as the sum of the exit time for all SCC on the inspected path). Our analysis correctly estimates that the expected extinction time is around 24 and for the fast variant and 40 for the slow variant.

Steady State Analysis. In many biological models, the natural steady state is either extinction or unbounded explosion. Hence it does not say much about the "seemingly steady" state (the temporary steady state), i.e., behaviour that is stable for a long but finite time. Therefore, the tool provides information not only on the steady state of the whole system, but also for each iteration separately since they represent the temporary steady states discussed above. Both can be visualized as colouring of states, with higher probabilities corresponding to darker colours, immediatelly giving a synoptic view on frequent behaviours.

Correlations. Finally, correlations between population sizes can be observed as follows. The GUI can be given a set of equivalences of the form $m \sim n$ for species i, j, meaning that if a state has (abstract) population m of species i and n of j then it is regarded as satisfying the correlation in question. It is coloured accordingly and the overall colouring of the system provides further indication under which behaviour or in which phases the correlation holds.

Example 6. We demonstrate these visualization options on a more complicated gene expression model [16], widely used model for benchmarking CRN analyzers, in Fig. 4. As reported in [16,18], the behaviour oscillates between two steady states with DNA on and DNA off. Moreover, there is a correlation between high amounts of RNA present and DNA being on, and no RNA with DNA off.

The complete system and its steady state distribution is depicted in the part a) using the iteration and IISCC arrangement. This view shows immediately without seeing any details that the only interesting states are in iteration 1 including all states with a high steady-state probability (the red colouring). Therefore, in part b), we zoom in to iteration 1 and use the IISCC arrangement. In order to observe the interesting switches between the temporary steady states, we collapse the IISCCs, in the part c), and thus ignore the internal (non-interesting) behaviour of the big IISCC. Finally, in part d), we use the correlation colouring to identify states where the required correlation holds (i.e. the blue states). Comparing part c) and d) immediately reveals that the system spends the majority of the time in the states where the correlation holds.

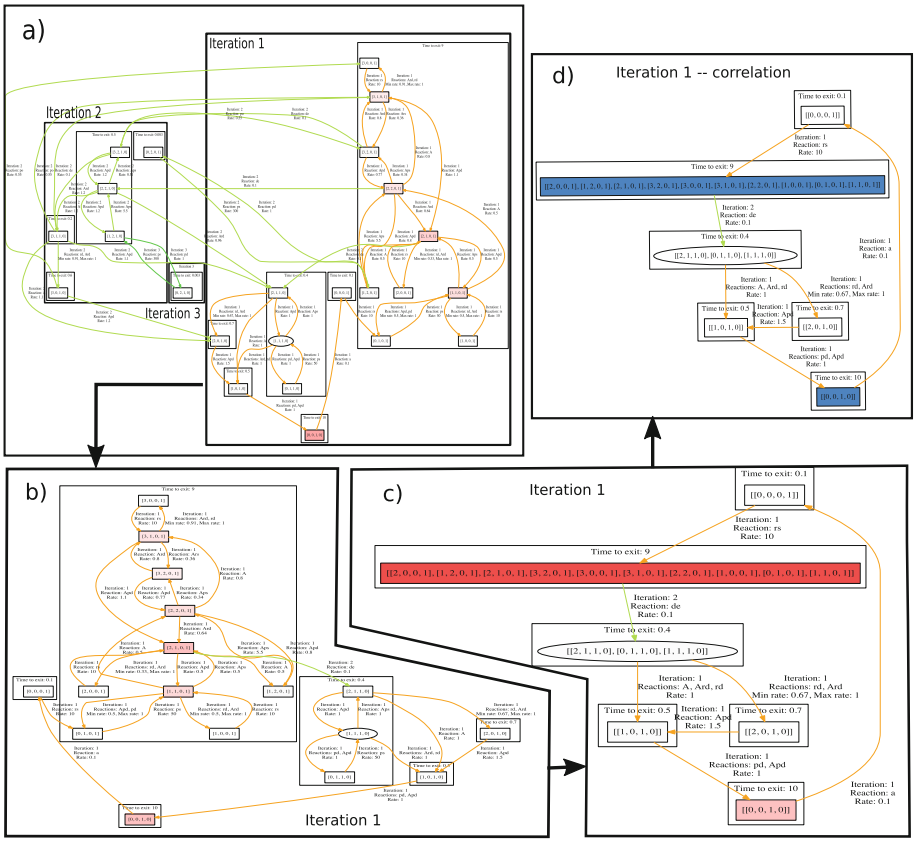


Fig. 4. A visualisation of the workflow for the extended gene expression model. (Color figure online)

2.5 Precision and Refinement

So far, we have illustrated the concepts and the functionality on models with an appropriate level of abstraction. However, it often happens that we start the investigations with a too coarse abstraction. Whenever this happens, it is important to notice this and appropriately refine the abstraction. While [9] does not discuss this issue, the tool provides support also for that.

Precision Parameters. There are several knobs for trading the size and the precision of the abstraction. They all come as input in the lower half of the Editor tab: discretization, bound, and envelope.

Example 7. Recall the initial abstraction for the Gene expression of Fig. 2 (with rate 10^{-3}). The abstraction, using thresholds $20, 50$ predicts an oscillation including low populations of P (1–20) which is not correct (recall that the P oscillates on high populations before the blocking reaction occurs). Figure 3(c) and (d)

show the abstraction and the consequent analysis and visualization for a refined model using thresholds $20, 50, 80, 150$ (instead of just $20, 50$). As already discussed, this abstraction already correctly predicts the system behaviour.

Discretization. The basic building block of each abstraction is the degree of details it preserves in the abstract states. Firstly, it determines how precisely we can observe the evolution of the population. For instance, whenever we want to detect whether a population typically grows beyond a bound or oscillates in a certain interval, such an interval should be present in the discretization. Secondly, the discretization should be fine enough so that in each state, the rates are reasonably (in orders of magnitude) precise. Fortunately, in our analysis their absolute precision is not vital. In contrast, we only need *relative* proportions of the rates to have the right *magnitude* to decide which behaviour is probable. Consequently, too rough abstraction is reflected in *"non-determinism"* when a state has two transitions under similar rate. In such a case, the probable behaviour cannot be determined. Therefore, the Visualization tab provides in the Colorization pane an option to provide suggestions for refinement, including highlighting non-deterministic states, pointing at the natural candidates for refinement. Note that we highlight only the states where the two transitions lead to mutually different SCCs so that a significant change in behaviour may occur.

Bounds. Similarly, for the single infinite interval (t_n, ∞), the tool inputs a *bound* which is a believed safe upper bound on the population of the species. Of course, it may be wrong. This is irrelevant in case when the population explodes beyond all bounds. However, whenever there are transitions from the highest level back to a lower one, its feasibility and rate are in question. Optimally, such states do not even occur in the pruned abstraction. If they do, we also highlight them using the Colorization for Refinement suggestions (in another colour).

Envelope. As too rough abstractions introduce too much non-determinism, dually, the degree of the non-determinism is determined (even defined) by the *envelope*, the factor between rates so that even the less probable option is still taken into account (and thus introduces non-determinism). Consequently, high values of envelope introduce non-determinism, making the analysis take also less important behaviour into account; in contrast, low values make the analyzed system deterministic, showing only the most probable behaviour. The choice of the envelope thus depends on whether such behaviours should also be reported.

2.6 Mean Simulations

Since our models, although abstract, have an operational semantics, we can even run simulations on them. Moreover, the accelerated transitions, as "sequences" of transitions, have a low variance in the expected time, by the law of large numbers. Hence their execution time can be chosen quite precisely in a deterministic way. Similarly, the time to leave an IIBSCC is quite deterministic. Thus we can generate simulation where the only random decisions are choices of transitions,

but the timing follows the mean time of the respective events. Moreover, runs within the pruned abstraction reflect the most important behaviours only.

Such *mean simulations*[2], which can thus be generated from our analysis, represent groups of typical runs (modulo small time shifts and order of transitions within an SCC, which are not very relevant). Therefore, a few such simulation reflect all the present behaviours (on a level of desired significant probability) and can serve to observe multi-modalities, bifurcations, rough transient timing as well as frequencies in the steady-state and temporary steady-state. To our best knowledge, such a concept has not yet been considered for simulation of stochastic systems.

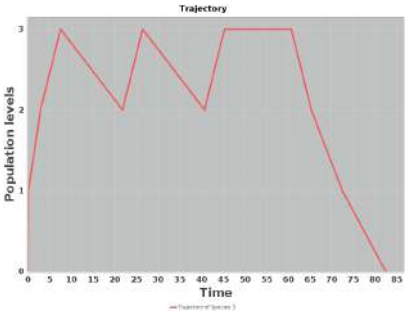


Fig. 5. Mean simulation for the slow variant of Gene expression, directly comparable to Fig. 2 (right, full line).

Example 8. Figure 5 shows an abstract simulation for our running example with discretisation thresholds $20, 50, 80, 150$. One can readily observe its validity with respect to the typical stochastic simulation in Fig. 2 (right, full blue line).

3 Conclusion

We have presented SeQuaiA, a scalable tool for robust and explainable analysis of CRNs. The analysis is precise enough as cross-validated with simulation-based results on several models widely used in the literature. One of the key contributions of the tool is the visualization, which is essential for clear presentation and easy interpretation of the results of our analysis.

References

1. Abate, A., Katoen, J.P., Lygeros, J., Prandini, M.: Approximate model checking of stochastic hybrid systems. Eur. J. Control **16**, 624–641 (2010)
2. Abate, A., Brim, L., Češka, M., Kwiatkowska, M.: Adaptive aggregation of Markov chains: quantitative analysis of chemical reaction networks. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 195–213. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_12
3. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: The computational power of population protocols. Distrib. Comput. **20**(4), 279–304 (2007)
4. Bortolussi, L., Hillston, J.: Fluid model checking. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 333–347. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32940-1_24
5. Cao, Y., Gillespie, D.T., Petzold, L.R.: The slow-scale stochastic simulation algorithm. J. Chem. Phys. **122**(1), 014116 (2005)

[2] They are not means of values from simulations since averaging oscillating values may result in no oscillation. Rather they reflect "mean patterns".

6. Cardelli, L.: Two-domain DNA strand displacement. Math. Struct. Comput. Sci. **23**(02), 247–271 (2013)
7. Cardelli, L., Kwiatkowska, M., Laurenti, L.: A stochastic hybrid approximation for chemical kinetics based on the linear noise approximation. In: Bartocci, E., Lio, P., Paoletti, N. (eds.) CMSB 2016. LNCS, vol. 9859, pp. 147–167. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45177-0_10
8. Cardelli, L., Tribastone, M., Tschaikowski, M., Vandin, A.: Maximal aggregation of polynomial dynamical systems. Proc. Natl. Acad. Sci. **114**(38), 10029–10034 (2017)
9. Češka, M., Křetínský, J.: Semi-quantitative abstraction and analysis of chemical reaction networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 475–496. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_28
10. Chellaboina, V., Bhat, S.P., Haddad, W.M., Bernstein, D.S.: Modeling and analysis of mass-action kinetics. IEEE Control Syst. Mag. **29**(4), 60–78 (2009)
11. Desharnais, J., Laviolette, F., Tracol, M.: Approximate analysis of probabilistic processes: logic, simulation and games. In: Quantitative Evaluation of SysTems (QEST), pp. 264–273. IEEE (2008)
12. D'Innocenzo, A., Abate, A., Katoen, J.P.: Robust PCTL model checking. In: Hybrid Systems: Computation and Control (HSCC), pp. 275–285. ACM (2012)
13. Ferm, L., Lötstedt, P.: Adaptive solution of the master equation in low dimensions. Appl. Numer. Math. **59**(1), 187–204 (2009)
14. Ganguly, A., Altintan, D., Koeppl, H.: Jump-diffusion approximation of stochastic reaction dynamics: error bounds and algorithms. Multiscale Model. Simul. **13**(4), 1390–1419 (2015)
15. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. J. Phys. Chem. **81**(25), 2340–2361 (1977)
16. Golding, I., Paulsson, J., Zawilski, S.M., Cox, E.C.: Real-time kinetics of gene activity in individual bacteria. Cell **123**(6), 1025–1036 (2005)
17. Goutsias, J.: Quasiequilibrium approximation of fast reaction kinetics in stochastic biochemical systems. J. Phys. Chem. **122**(18), 184102 (2005)
18. Hasenauer, J., Wolf, V., Kazeroonian, A., Theis, F.J.: Method of conditional moments (MCM) for the Chemical Master Equation. J. Math. Biol. **69**(3), 687–735 (2013). https://doi.org/10.1007/s00285-013-0711-5
19. Heath, J., Kwiatkowska, M., Norman, G., Parker, D., Tymchyshyn, O.: Probabilistic model checking of complex biological pathways. Theor. Comput. Sci. **391**(3), 239–257 (2008)
20. Henzinger, T.A., Mateescu, M., Wolf, V.: Sliding window abstraction for infinite markov chains. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 337–352. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_27
21. Henzinger, T.A., Mikeev, L., Mateescu, M., Wolf, V.: Hybrid numerical solution of the chemical master equation. In: Computational Methods in Systems Biology (CMSB), pp. 55–65. ACM (2010)
22. Hepp, B., Gupta, A., Khammash, M.: Adaptive hybrid simulations for multiscale stochastic reaction networks. J. Chem. Phys. **142**(3), 034118 (2015)
23. Hoops, S., et al.: COPASI - a complex pathway simulator. Bioinformatics **22**(24), 3067–3074 (2006). https://doi.org/10.1093/bioinformatics/btl485, http://bioinformatics.oxfordjournals.org/content/22/24/3067.abstract
24. Karp, R.M., Miller, R.E.: Parallel program schemata. J. Comput. Syst. Sci. **3**(2), 147–195 (1969)

25. Lakin, M.R., Youssef, S., Polo, F., Emmott, S., Phillips, A.: Visual DSD: a design and analysis tool for dna strand displacement systems. Bioinformatics **27**(22), 3211–3213 (2011)
26. Maarleveld, T.R., Olivier, B.G., Bruggeman, F.J.: StochPy: a comprehensive, user-friendly tool for simulating stochastic biological processes. PloS one **8**(11), e79345 (2013)
27. Madsen, C., Myers, C., Roehner, N., Winstead, C., Zhang, Z.: Utilizing stochastic model checking to analyze genetic circuits. In: Computational Intelligence in Bioinformatics and Computational Biology (CIBCB), pp. 379–386. IEEE (2012)
28. Mateescu, M., Wolf, V., Didier, F., Henzinger, T.A.: Fast adaptive uniformization of the chemical master equation. IET Syst. Biol. **4**(6), 441–452 (2010)
29. Munsky, B., Khammash, M.: The finite state projection algorithm for the solution of the chemical master equation. J. Chem. Phys. **124**, 044104 (2006)
30. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989)
31. Rao, C.V., Arkin, A.P.: Stochastic chemical kinetics and the quasi-steady-state assumption: application to the gillespie algorithm. J. Chem. Phys. **118**(11), 4999–5010 (2003)
32. Salis, H., Kaznessis, Y.: Accurate hybrid stochastic simulation of a system of coupled chemical or biochemical reactions. J. Chem. Phys. **122**(5), 054103 (2005)
33. Soloveichik, D., Seelig, G., Winfree, E.: DNA as a universal substrate for chemical kinetics. Proc. Natl. Acad. Sci. U.S.A. **107**(12), 5393–5398 (2010)
34. Van Kampen, N.G.: Stochastic Processes in Physics and Chemistry, vol. 1. Elsevier, Amsterdam (1992)
35. Zhang, J., Watson, L.T., Cao, Y.: Adaptive aggregation method for the chemical master equation. Int. J. Comput. Biol. Drug Des. **2**(2), 134–148 (2009)

Author Index