DIFTS13

Program Proceedings

International Workshop on Design and Implementation of Formal Tools and Systems

Portland, OR

October 19, 2013

Co-located with FMCAD and MEMOCODE

Preface

The second DIFTS (Design and Implementation of Formal Tools and Systems) workshop was held at Portland, Oregon on Oct 19, 2013, co-located with Formal Methods in Computer-Aided Design Conference (FMCAD), and Formal Methods and Models for Co-design (MEMOCODE). The workshop emphasized the insightful experiences in tool and system design. The goal of the workshop is to provide a forum for sharing challenges and solutions that are original with ground breaking results. The workshop provided an opportunity for discussing engineering aspects and various design decisions required to put such formal tools and systems into practical use. It took a broad view of the formal tools/systems area, and solicited contributions from hardware and software domains such as decision procedures, verification, testing, validation, diagnosis, debugging, and synthesis.

The workshop received 10 original submissions, out of which 3 were chosen under tool category, and 3 were chosen under system category. There were also three invited talks: first was given by Rance Cleaveland, Reactive Systems Inc., USA on "Approximate Formal verification using Model-based Testing", second was given by Masahiro Fujita, University of Tokyo on "Diagnosis and correction of buggy hardware/software with formal approaches", and third talk was given by Dhiraj Goswami, Synopsys Inc. on "Stimulus generation, enhancement and debug in constraint random verification."

First of all, we thank FMCAD's steering committee for their continual support. We also thank FMCAD chairs Sandip Ray and Barbara Jobstmann, and MEMOCODE chairs Marly Roncken and Fei Xie for a seamless organization. We also thank Joe Leslie-Hurd for his help in local arrangements. We thank Boğaziçi University, Turkey for hosting the DIFTS website. We sincerely thank the program committee members and sub reviewers for selecting the papers and providing candid review feedbacks to the authors. Last but not least, we thank all the authors for contributing to the workshop and to all the participants of the workshop.

Malay K. Ganai and Alper Sen
Program Chairs
DIFTS 2013

General Program Chairs

Malay Ganai	NEC Labs America, USA
Alper Sen	Bogazici University, Turkey

Program Committee

Armin Biere	Johannes Kepler University, Austria
Gianpiero Cabodi	Politecnico di Torino, Italy
Franco Fummi	University of Verona, Italy
Malay Ganai	NEC Labs America, USA
Daniel Grosse	University of Bremen, Germnay
William Hung	Synopsys Inc, USA
Daniel Kroening	Oxford University, UK
Alper Sen	Bogazici University, Turkey
Ofer Strichman	Technion - Israel Institute of Technology, Israel
Chao Wang	Virginia Tech, USA

Additional Reviewers

Balakrishnan, Gogul
Eldib, Hassan
Horn, Alexander
Ivrii, Alexander
Kusano, Markus
Le, Hoang
Schrammel, Peter
Sinz, Carsten
Sousa, Marcelo
Suelflow, Andre

Table of Contents

Approximate Formal Verification using Model-based Testing 1
Rance Cleaveland (Invited speaker)

Diagnosis and Correction of Buggy Hardware/Software with Formal Approaches 2
Masahiro Fujita (Invited speaker)

Stimulus generation, enhancement and debug in constraint random verification 3
Dhiraj Goswami (Invited speaker)

A Fast Reparameterization Procedure 4
Niklas Een and Alan Mishchenko

LEC: Learning driven data-path equivalence checking 9
Jiang Long, Robert Brayton and Michael Case

Trading-off Incrementality and Dynamic Restart of Multiple Solvers in IC3 19
Marco Palena, Gianpiero Cabodi and Alan Mishchenko

Lemmas on Demand for Lambdas 28
Mathias Preiner, Aina Niemetz and Armin Biere

CHIMP: a Tool for Assertion-Based Dynamic Verification of SystemC Models 38
Sonali Dutta, Moshe Y. Vardi and Deian Tabakov

Abstraction-Based Livelock/Deadlock Checking for Hardware Verification 46
In-Ho Moon and Kevin Harer

Approximate Formal Verification using Model-based Testing

Rance Cleaveland
University of Maryland, USA

Abstract: In model-based testing, (semi-)formal models of systems are used to drive the derivation of test cases to be applied to the system-under-test (SUT). The technology has long been a part of the traditional hardware-design workflows, and it is beginning to find application in embedded-software development processes also. In automotive and land-vehicle control-system design in particular, models in languages such as MATLAB® / Simulink® /Stateflow® are used to drive the testing of the software used to control vehicle behavior, with tools like Reactis®, developed by a team including the speaker, providing automated test-case generation support for this endeavor. This talk will discuss how test-case generation capabilities may also be used to help verify that models meet formal specifications of their behavior. The method we advocate, Instrumentation-Based Verification (IBV), involves the formalizaton of behavior specifications as models that are used to instrument the model to be verified, and the use of coverage testing of the instrumented model to search for specification violations. The presentation will discuss the foundations of IBV, the test-generation approach and other features in Reactis® that are used to support IBV, and the results of several case studies involving the use of the methods.

Diagnosis and Correction of Buggy Hardware/Software with Formal Approaches

Masahiro Fujita
University of Tokyo, Japan

Abstract: There have been intensive researches on debugging hardware as well as software. Some are very ad-hoc and based on simple heuristics, but others utilize formal methods and are mathematically modeled. In this talk, we first review various proposals on debugging from historical viewpoints, and then summarize the state-of-the-art in terms of both diagnosis and automatic correction of designs. In particular we show various approaches with SAT-based formulations of diagnosis and correction problems. We also discuss about them in relation to manufacturing test techniques. That is, if the design errors are within the pre-determined types and/or areas, there could be very efficient ways to formally verify, diagnosis and correction methods with small numbers of test vectors. In the last part of the talk, future perspectives including post-silicon issues are discussed.

Stimulus Generation, Enhancements and Debug in Constraint Random Verification

Dhiraj Goswami
Synopsys, USA

Abstract: Verification cost in an IC design team occupies 60-80% of the entire working resources and efforts. Functional verification, posed at the foremost stage of the IC design flow, determines the customers' ability to find bugs quickly and thereby their time-to-results (TTR) and cost-of-results (COR). Consequently, functional verification has been the focus of the EDA industry for the last several decades.

Constrained random simulation methodologies have become increasingly popular for functional verification of complex designs, as an alternative to directed-test based simulation. In a constrained random simulation methodology, random vectors are generated to satisfy certain operating constraints of the design. These constraints are usually specified as part of a testbench program (using industry-standard testbench languages, like SystemVerilog from Synopsys, e from Cadence, OpenVera, etc.). The testbench automation tool (TBA) is then expected to generate random solutions for specified random variables, such that all the specified constraints over these random variables are satisfied. These random solutions are then used to generate valid random stimulus for the Design Under Verification (DUV). This stimulus is simulated using industry-standard simulation tools, like VCS from Synopsys, NC-Verilog from Cadence, etc. The results of simulation are then typically examined within the testbench program to monitor functional coverage, which gives a measure of confidence on the verification quality and completeness.

In this talk we will review the challenges of stimulus and configuration generation using constraint random verification methodology. We will also explore why state-of-the-art debug solutions are important to handle complexity and improve the quality of stimulus.

A Fast Reparameterization Procedure

Niklas Een, Alan Mishchenko
{een,alanmi}@eecs.berkeley.edu

Berkeley Verification and Synthesis Research Center
EECS Department
University of California, Berkeley, USA.

Abstract. Reparameterization, also known as range preserving logic synthesis, replaces a logic cone by another logic cone, which has fewer inputs while producing the same output combinations as the original cone. It is expected that a smaller circuit leads to a shorter verification time. This paper describes an approach to reparameterization, which is faster but not as general as the previous work. The new procedure is particularly well-suited for circuits derived by localization abstraction.

1 Introduction

The use of reparameterization as a circuit transformation in the verification flow was pioneered by Baumgartner et. al. in [1]. In their work, new positions for the primary inputs (PIs) are determined by finding a minimum cut between the current PI positions and the next-state variables (flop inputs). BDDs are then used to compute the *range* (or image) on the cut. Finally, a new logic cone with the same range (but fewer PIs), is synthesized from the BDDs and grafted onto the original design in place of the current logic cone. This is a very powerful transformation, but it has potential drawbacks: (i) the BDDs may blow up and exhaust the memory, (ii) the extracted circuit may be larger than the logic it replaces, and (iii) the runtime overhead may be too high.

In contrast, the proposed approach is based on greedy local transformations, capturing only a subset of optimization opportunities. However, memory consumption is modest, runtimes are very low, and the resulting design is always smaller, or of the same size, as the original design. It is shown experimentally that the proposed method leads to sizeble reductions when applied for circuits produced by localization abstraction [5].

2 Fast Reparameterization

The fast reparameterization algorithm is based on the following observation: if a node dominates[1] a set of PIs, and those PIs are sufficient to force both a zero and a one at that node, regardless of the values given to the other PIs and state variables, then that node can be replaced by a new primary input, while the unused logic cone driving the original node can be removed. The old primary inputs dominated by the given node are also removed by this procedure.

> **Example.** Suppose a design contains inputs x_1, x_2, and a gate $\textsc{Xor}(x_1, x_2)$; and that furthermore, x_1 has no other fanouts besides this XOR-gate. Then, no matter which value x_2 takes, both a zero and a one can be forced at the output of the XOR by setting x_1 appropriately, and thus the XOR-gate can, for verification purposes, be replaced by a primary input.

The proposed method to find similar situations starts by computing all dominators of the netlist graph, then for each candidate node dominating at least one PI the following quantification problem is solved: "*for each* assignment to the non-dominated gates, *does there exist* a pair of assignments to the dominated PIs that results in a zero and a one at the candidate node". More formally, assuming that x represents non-dominated gates ("external inputs") and y_i represents dominated PIs ("internal inputs"), the following is always true for the node's function ϕ:

$$\forall x \; \exists y_0, y_1 \; . \; \neg\phi(x, y_0) \wedge \phi(x, y_1)$$

Important features of this approach are:

(i) It is circuit based, while early work on reparameterization was based on transition relations [4].

(ii) In its simplest form, the proposed restructuring replaces some internal nodes by new primary inputs and remove dangling logic.

(iii) The analysis is completely combinational: no information on the reachable state-space is used.

(iv) If the property was disproved after reparametrization, it is straight-forward to remap the resulting counter-example to depend on the original primary inputs.

It is important to realize that by analyzing and applying reductions in topological order from PIs to POs, regions amenable to reparameterization are gradually reduced to contain fewer gates and PIs. By this process, the result of repeatedly applying local transformations can lead to a substantial global reduction. In the current implementation, the above formula is evaluated by exhaustive simulation of

[1] A node n dominates another node m iff every path from m to a primary output goes through node n.

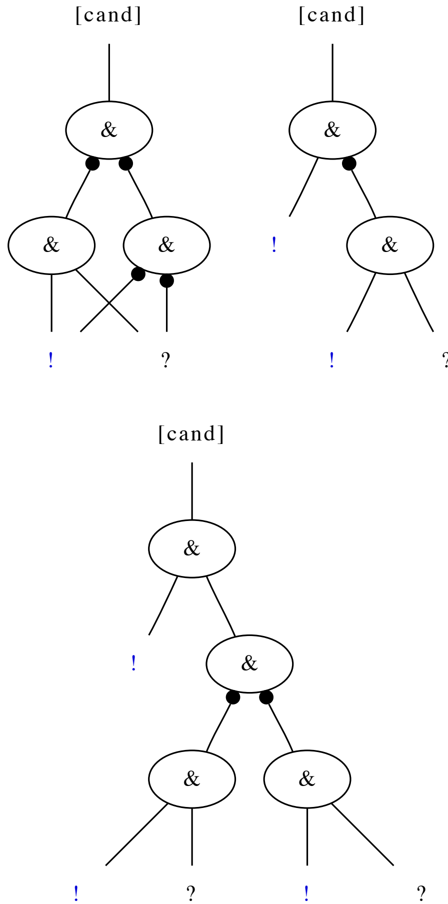


Figure 1. *Example of subgraphs that can be reduced.* Here "!" denote nodes we can control (a PI dominated by the output node); "?" denote nodes that can assume any value beyond our control. In the above examples, the output node "[cand]" can be forced to both a zero and a one by choosing the right values for the "!", regardless of the values of the "?". In such cases, the output node is replaced by a new PI.

the logic cone rooted in the given node while the cone is limited to 8 inputs. Limiting the scope to cones with 8 inputs and simulating 256 bit patterns (or eight 32-bit words) seems to be enough to saturate the reduction achievable on the benchmarks where the method is applicable.

Some typical reductions are shown in *Figure 1*. The graphs should be understood as sub-circuits of a netlist being reparameterized. The exclamation marks denote "internal" PIs dominated by the top-node, and hence under our control; and the question marks denote gates with fanouts outside the displayed logic cone, for which no assumption on their values can be made. The full algorithm is described in *Figure 2*.

Counterexample reconstruction. There are several ways that a trace on the reduced netlist can be lifted to the original netlist. For instance, the removed logic between the new PIs and the old PIs can be stored in a separate netlist. The trace on the reduced netlist can then be pro-

Fast Reparameterization

– Compute dominators. For a DAG with bounded in-degree (such as an And-Inverter-Graph), this is a linear operation in the number of nodes (see *Figure 3*).

– For each PI, add all its dominators to the set of "candidates".

– For each candidate c, in topological order from inputs to outputs:

- Compute the set D of nodes dominated by c (*Figure 4*).
- Denote the PIs in D "internal" inputs.
- Denote any node outside D, but being a direct fanin of a node inside D, an "external" input (may be any gate type).
- Simulate all possible assignments to the internal and external inputs. If *for all* external assignments *there exists* an internal assignment that gives a 0 at c, and *another* internal assignment that gives a 1 at c, then substitute c by a new primary input.

Figure 2. *Steps of the reparameterization algorithm.*

Compute Dominators

– Initialize all POs to dominate themselves

– Traverse the netlist in reverse topological order (from POs to PIs), and mark the children of each node as being dominated by the same dominator as yourself *unless* the child has already been assigned a dominator

– For already marked children, compute the "meet" of the two dominators, i.e. find the first common dominator. If there is no common dominator, mark the node as dominating itself.

Figure 3. *Review of the "finding direct dominators" algorithm for DAGs.* For a more complete treatment of this topic, see [8], or for a more precise description, the source code of "computeDominators()" in "ZZ/Netlist/StdLib.cc" of ABC-ZZ [11].

Compute Dominated Area

$area = \{w_dom\}$ – *init. set of gates to the dominator*
$count = [0, 0, \ldots, 0]$ – *count is a map "gate → integer"*
for $w \in area$:
  for $v \in$ ***faninsOf***(w):
    $count[v]$++
    if $count[v] == num_of_fanouts[v]$:
      $area = area \cup \{v\}$

Figure 4. *Find nodes used only by gate "w_dom".* This set is sometimes called *maximum fanout free cone* (MFFC). The outer for-loop over the elements of *area* is meant to visit all elements added by the inner for-loop. For this procedure, flops are treated as sinks, i.e. having no fanins.

jected onto the original PIs by rerunning the simulation used to produce the reparameterized circuit, and for each candidate pick an assignment that gives the correct value. But even simpler, one can just put the original netlist into a SAT solver and assert the values from the trace onto the appropriate variables and call the SAT solver to complete the assignment. In practice, this seems to always work well.

3 Improvements

The algorithm described in the previous section replaces internal nodes with inputs, and thus only removes logic. If we are prepared to forgo this admittedly nice property and occasionally add a bit of new logic, then nodes that are not completely controllable by their dominated inputs can still be reparameterized by the following method: for node with function $\phi(x, y)$, where x are external inputs and y are internal inputs, compute the following two functions:

$$\phi_0(x) \equiv \forall y. \neg\phi(x, y)$$
$$\phi_1(x) \equiv \forall y. \phi\ \ (x, y)$$

Using these two functions, ϕ can be resynthesized using a single input y_{new} by the expression:

$$\neg\phi_0(x) \wedge (\phi_1(x) \vee y_{new})$$

In other words, if two or more inputs are dominated by the node ϕ, a reduction in the number of inputs is guaranteed. Depending on the shape of the original logic, and how well new logic for ϕ_0 and ϕ_1 is synthesized, the number of logic gates may either increase or decrease. In our implementation, logic for ϕ_0 and ϕ_1 is created by the fast irredundant sum-of-product ("isop") proposed by Shin-ichi Minato in [10]. We greedily apply this extended method for all nodes with two or more dominated PIs, even if it leads to a blow-up in logic size. To counter such cases, fast logic synthesis can be applied after the reparameterization. Obviously, there are many ways to refine this scheme.

4 Future Work

Another possible improvement to the method is extending the reparameterization algorithm to work for multi-output cones. As an example, consider a two-output cone where the outputs can be forced to all four combinations {00, 01, 10, 11} by choosing appropriate values for dominated inputs. In such a case, the cone can be replaced by two free inputs. If some of the four combinations at the outputs are impossible under conditions expressed in terms of non-controllable signals, a logic cone can be constructed to characterize these conditions and reduce the number of PIs by adding logic similar to the case of a single-output cone.

5 Experiments

As part of the experimental evaluation, all benchmarks from the single-property track of the Hardware Modelchecking Competition 2012 were considered. Localization abstraction [5] was applied with a timeout of one hour to each benchmark and the resulting models meeting the following criteria were kept:

- At least half of the flops were removed by abstraction.
- The abstraction was accurate (no spurious counterexamples).
- At least one of the verification engines could prove the property within one hour.

The sizes of benchmarks selected in this way are listed in table *Table 1*. All those models were given to the reparameterization engine, both in *weak* mode and *strong* mode, the latter using the improvements described in section 3. The reparameterized models were also post-processed with a quick simplification method called "shrink" which is part of the ABC package [7]. The longest runtime for weak reparameterization was 16 ms, for strong reparameterization 28 ms and for the simplification phase 50 ms.[2] Reductions are listed in table *Table 2*.

For comparison, *Table 2* also include the results of running an industrial implementation of the BDD based algorithm of [1]. Because runtimes are significantly longer with this algorithm, they are given their own column. These results were given to us from IBM, and according to their statement "are not tweaked as much as they could be".

All benchmarks were given to three engines: *Property Directed Reachability* [2, 6], *BDD-based reachability* [3], and *Interpolation-based Model Checking* [9]. The complete table of results is given in *Table 3*. A slice of this table, showing only results for PDR, with and without (strong) reparameterization, is given in *Table 4* together with a scatter plot.

Analysis. Firstly, we see a speedup of 100x-1000x over previous work in the runtime of the reparameterization algorithm itself, with comparable quality of results for the application under consideration (models resulting from localization abstraction). This means the algorithm can *always* be applied without the need for careful orchestration. Secondly, we see an average speedup of 2.5x in verification times when applying reparameterization in conjunction with PDR, which is also the best overall engine on these examples. For two benchmarks, *6s121* and *6s150*, BDD reachability do substantially better than PDR, and for the latter (where runtimes are meaningful) the speedup due to reparameterization is greater than 3x. Furthermore, for BDD reachability one can see that on several occasions (*6s30* in particular), reparameterization is completely crucial for performance. Finally, interpolation based modelchecking (IMC) seems to be largely unaffected by reparameterization.

[2] Benchmarks from HWMCC'12 are quite small. For comparison: running reparameterization on a 7 million gate design from one of our industrial collaborators took 4.1 s.

Abstraction Phase

Design	#And	#PI	#FF	Depth
6s102	6,594	72	1,121 → 56	23
6s121	1,636	99	419 → 110	19
6s132	1,216	94	139 → 113	7
6s144	41,862	480	3,337 → 236	18
6s150	5,448	146	1,044 → 323	103
6s159	1,469	13	252 → 18	4
6s164	1,095	91	198 → 93	16
6s189	36,851	479	2,434 → 259	18
6s194	12,049	532	2,389 → 198	45
6s30	1,043,139	32,994	1,195 → 128	32
6s43	7,408	30	965 → 310	25
6s50	16,700	1,570	3,107 → 207	52
6s51	16,701	1,570	3,107 → 209	65
bob05	18,043	224	2,404 → 146	106
bob1u05cu	32,063	224	4,377 → 146	106

Table 1. *Sizes of original designs and abstract models.* Column #FF shows how many flip-flops were turned into unconstrained inputs by localization abstraction. The removed FFs show up as PIs in the other tables. Last column shows the BMC depth used by the abstraction engine (see [5] for more details).

Reparameterization

	No Reparam.		Weak Rep.		Strong Rep.		BDD Reparam.		
Design	**#And**	**#PI**	**#And**	**#PI**	**#And**	**#PI**	**#And**	**#PI**	**Runtime**
6s102	6,594	1,137	1,247	331	1,188	267	1,283	285	71.91 s
6s121	1,636	408	627	120	559	66	732	41	0.27 s
6s132	1,216	120	1,108	62	1,102	55	2,731	36	0.80 s
6s144	41,862	3,580	10,172	1,038	9,494	812	13,259	889	60.50 s
6s150	5,448	867	3,062	506	2,213	89	2,231	46	1.81 s
6s159	1,469	247	116	20	114	19	256	13	0.02 s
6s164	1,077	196	661	109	499	41	3,413	40	11.61 s
6s189	36,851	2,654	10,033	1,004	9,552	794	12,051	814	63.11 s
6s194	12,049	2,723	1,366	184	1,348	166	1,612	121	10.37 s
6s30	102,535	34,061	1,508	307	1,184	205	603	50	0.74 s
6s43	7,408	685	3,451	304	3,218	202	17,691	101	2.07 s
6s50	16,700	4,470	1,841	350	1,652	270	4,319	752	46.57 s
6s51	16,701	4,468	1,828	350	1,639	268	1,255	62	16.34 s
bob05	18,043	2,358	1,618	187	1,388	52	1,586	43	0.31 s
bob1u05cu	32,063	4,455	1,618	187	1,388	52	1,586	43	0.33 s

Table 2. *Effect of reparameterization on the abstracted models.* "Weak" reparameterization refers to the basic method described in section 2, "Strong" additionally includes the improvements discussed in section 3. Runtimes are omitted as the average CPU time was 4-8 ms (and the longest 28 ms). However, BDD based reparameterization is not as scalable as the method presented in this paper, and runtimes (typically between 100x-1000x longer) are listed for reference.

References

[1] J. Baumgartner and H. Mony. **Maximal Input Reduction of Sequential Netlists via Synergistic Reparameterization and Localization Strategies**. In *Proc. of CHARME*, pages 222–237, 2005.

[2] Aaron Bradley. **IC3: SAT-Based Model Checking Without Unrolling**. In *Proc. of VMCAI*, 2011.

[3] R. E. Bryant. **Graph-Based Algorithms for Boolean Function Manipulation**. In *IEEE Transactions on Computers, vol. c-35, no.8, Aug.*, 1986.

[4] Pankaj Chauhan, Edmund Clarke, and Daniel Kroening. **A SAT-Based Algorithm for Reparameterization in Symbolic Simulation**. In *Proc. of DAC*, 2004.

[5] N. Een, A. Mishchenko, and N. Amla. **A Single-Instance Incremental SAT Formulation of Proof- and Counterexample-Based Abstraction**. In *FMCAD*, 2010.

[6] Niklas Een, Alan Mishchenko, and Robert Brayton. **Efficient Implementation of Property Directed Reachability**. In *Proc. of FMCAD*, 2011.

Verification Runtimes

	PDR			BDD reach.			IMC		
	NoRep.	**Weak**	**Strong**	**NoRep.**	**Weak**	**Strong**	**NoRep.**	**Weak**	**Strong**
6s102	1.7	0.4	0.4	121.2	90.2	204.3	–	2619.2	–
6s121	64.0	12.3	5.4	0.5	0.3	0.4	10.2	5.8	36.0
6s132	7.8	8.1	7.9	2327.3	1375.5	–	201.4	384.9	267.9
6s144	10.3	12.4	9.6	–	–	–	1177.5	942.1	1413.0
6s150	–	2323.7	–	539.3	143.6	189.1	–	–	–
6s159	0.0	0.0	0.0	0.1	0.1	0.1	0.1	0.1	0.1
6s164	7.0	34.1	6.8	–	33.6	0.4	4.4	8.5	3.0
6s189	11.9	7.8	8.6	–	–	–	738.0	396.5	366.0
6s194	4.7	4.6	3.7	307.5	42.5	742.2	798.9	1042.8	1103.3
6s30	15.9	45.3	12.3	–	17.6	49.2	–	–	–
6s43	13.6	11.7	9.1	–	1191.1	1390.4	–	–	–
6s50	14.7	10.8	5.4	941.6	241.0	1680.8	1795.6	820.0	2348.9
6s51	18.7	7.1	4.1	303.8	147.6	1891.2	1806.8	954.8	1737.8
bob05	0.3	0.3	0.3	2450.5	2371.4	1253.2	30.3	26.3	25.1
bob1u05cu	0.3	0.2	0.3	–	2157.6	1088.3	30.3	24.0	24.8

Table 3. *Full table of results.* Each design after abstraction is given to three engines: Property Directed Reachability (PDR), BDD-based reachability (BDD), and Interpolation-based Model Checking (IMC). Each engine is run on three versions of the model with no/weak/strong reparameterization applied. A dash represents a timeout after one hour.

Design	NoRep.	Rep.
6s102	1.66	**0.40**
6s121	64.00	**5.43**
6s132	**7.82**	7.90
6s144	10.34	**9.56**
6s159	0.04	**0.03**
6s164	6.97	**6.83**
6s189	11.86	**8.59**
6s194	4.71	**3.72**
6s30	15.92	**12.29**
6s43	13.63	**9.15**
6s50	14.66	**5.44**
6s51	18.72	**4.08**
bob05	0.33	**0.26**
bob1u05cu	0.33	**0.26**

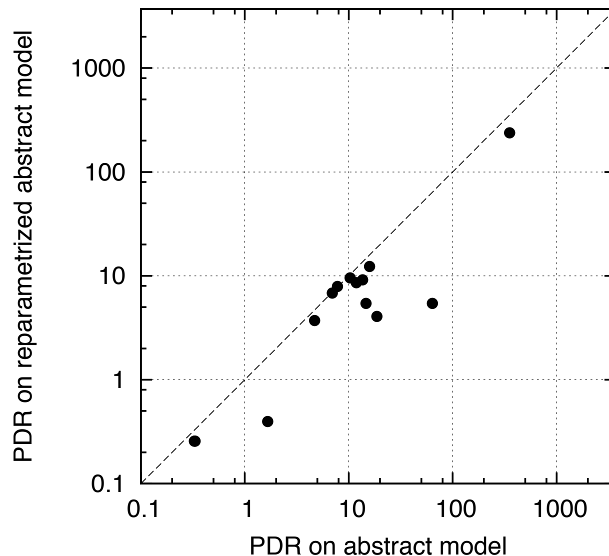


Table 4. *Runtime improvements for PDR.* Column "NoRep." shows runtimes (in seconds) for proving the property of each benchmark using PDR after abstraction, but without reparameterization; column "Rep." shows runtimes after reparameterization. The scatter plot on the right places these runtime pairs on a log-scale. The average speedup is 2.5x.

[7] Berkeley Logic Synthesis Group. **ABC: A System for Sequential Synthesis and Verification**. *http://www.eecs.berkeley.edu/˜alanmi/abc/*, v00127p.

[8] T. Lengauer and R.E. Tarjan. **A Fast Algorithm for Finding Dominators in a Flowgraph**. In *ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, July*, pages 121–141, 1979.

[9] K. L McMillan. **Interpolation and SAT-based Model Checking**. In *Proc. of CAV*, 2003.

[10] S. Minato. **Fast Generation of Irredundant Sum-Of-Products Forms from Binary Decision Diagrams**. In *Proc. of SASIMI*, 1992.

[11] Berkeley Verification and Synthesis Research Center. **ABC-ZZ: A C++ framework for verification & synthesis**. *https://bitbucket.org/niklaseen/abc-zz.*

LEC: Learning-Driven Data-path Equivalence Checking

Jiang Long*, Robert K. Brayton*, Michael Case†
*EECS Department, UC-Berkeley
{jlong, brayton}@eecs.berkeley.edu
†Calypto Design Systems
{mcase}@calypto.com

***Abstract*—**
In the LEC system, we employ a learning-driven approach for solving combinational data-path equivalence checking problems. The data-path logic is specified using Boolean and word-level operators in VHDL/Verilog. The targeted application area are C-to-RTL equivalence checking problems found in an industrial setting. These are difficult because of the algebraic transformations done on the data-path logic for highly optimized implementations. Without high level knowledge, existing techniques in bit-level equivalence checking and QF_BV SMT solving are unable to solve these problems effectively. It is crucial to reverse engineer such transformations to bring more similarity between the two sides of the logic. However, it is difficult to extract algebraic logic embedded in a cloud of Boolean and word-level arithmetic operators. To address this, LEC uses a compositional proof methodology and analysis beyond the bit and word level by incorporating algebraic reasoning through polynomial reconstruction. LEC's open architecture allows new solver techniques to be integrated progressively. It builds sub-model trees, recursively transformating the sub-problems to simplify and expose the actual bottleneck arithmetic logic. In addition to rewriting rules that normalize the arithmetic operators, LEC supports conditional rewriting, where the application of a rule is dependent on the existence of invariants in the design itself. LEC utilizes both functional and structural information of the data-path logic to recognize and reconstruct algebraic transformations. A case-study illustrates the steps used to extract the arithmetic embedded in a data-path design as a linear sum of signed integers, and shows the procedures that collaboratively led to a successful compositional proof.

I. Introduction

With the increasing popularity of high-level design methodologies there is renewed interest in data-path equivalence checking [3][13][18][20]. In such an application, a design prototype is first implemented and validated in C/C++, and then used as the golden specification. A corresponding Verilog/VHDL design is implemented either manually or automatically through high-level synthesis tool [2][4][15]. In both cases, a miter logic for equivalence checking is formed to prove the correctness of the generated RTL model by comparing it against the original C/C++ implementation.

The data-path logic targeted in this paper is specified using Verilog/VHDL. The bit and word-level operators in Verilog/VHDL have the same semantic expressiveness as SMT QF_BV theory[5]. Table I gives a one-to-one correspondence between Verilog and QF_BV unsigned operators. Signed arithmetic operators are also supported. The complexity of such an equivalence problem is NP-complete. However, on the extreme end, the complexity becomes $O(1)$ of the size of the network if the two designs are structurally the same. An NP-complete problem can be tackled by using SAT-solvers as a general procedure. To counter the capacity limitation of SAT-solving, it is crucial to reduce the complexity by identifying internal match points and by conducting transformations to bring in more structural similarity between the two sides of the miter logic.

	Verilog operators	**SMT QF_BV operators**
Boolean	$\&\&, \|\|, !, \oplus, mux$	and, or, not, xor, ite
bit-wise	$\&, \|, \sim, \oplus, mux$	$bvand, bvor, bvnot, bvxor, bvite$
arithmetic	$+, -, *, /, \%$	$bvadd, bvsub, bvmul, bvdiv, bvmod$
extract	$[]$	$extract$
concat	$\{\}$	$concat$
comparator	$<, >, \leq, \geq$	$bvugt, bvult, bvuge, bvule$
shifter	$\ll, \gg$	$bvshl, bvshr$

TABLE I
Supported operators (unsigned)

A. Motivation

The differences between the two data-path logics under equivalence checking are introduced by various arithmetic transformations for timing, area and power optimizations. These optimizations are domain specific and can be very specialized towards a particular data-path design and underlying technology. They have the following characteristics:

- The two sides of the miter logic are architecturally different and have no internal match points.
- Many expensive operators such as adders and multipliers are converted to cheaper but more complex implementations and the order of computations are changed. It is not a scalable solution to rely on SAT solving on the bit-blasted model.
- The parts of the transformed portion are embedded in a cloud of bit and word level operators. Algebraic extraction [8][26] of arithmetic logic based on structural patterns is generally too restrictive to handle real-world post-optimization data-path logic.
- Word-level rewriting uses local transformation. Without high-level information, local rewriting is not able to make the two sides of the miter logic structurally more similar.

Lacking high-level knowledge of the data-path logic, the equivalence problems can be very difficult for gate-level equiv-

alence checking and general QF_BV SMT solvers. Strategically, LEC views the bottleneck of such problems as having been introduced by high-level optimizations and employs a collaborative approach to isolate, recognize and reconstruct the high-level transformations to simplify the miter model by bringing in more structural similarities.

B. Contributions

The LEC system incorporates compositional proof strategies, uses rewriting to normalize arithmetic operators, and conducts analysis beyond bit and word level. The collaborating procedures help to expose the actual bottleneck in a proof of equivalence. The novel aspects of this system are:

1) It uses global algebraic reasoning through polynomial reconstruction. In the case-study, it uses the functional information of the design to reverse engineer the arithmetic expression as a linear sum and also uses a structural skeleton of the original data-path to achieve the equivalence proof.
2) It supports conditional rewriting and proves required invariants as pre-conditions.
3) It uses recursive transformations that target making both sides of the miter logic structurally more similar and hence more amenable to bit-level SAT sweeping.
4) It has an open architecture, allowing new solver techniques to be integrated progressively.

Through a case study, we demonstrate the steps that were used to reconstruct the arithmetic embedded in a data-path design as a linear sum of signed integers, as well as all the procedures that compositionally led to a successful equivalence proof. The experimental results demonstrate the effectiveness of these collaborating procedures.

C. Overview

The overall tool flow is described in Section II. Learning techniques and system integration are presented in Section III and IV. A case study is presented in Section V. Experimental results is presented in Section VI followed by a comparison with related work and conclusion.

II. Tool flow

LEC takes Verilog/VHDL as the input language for the data-path logic under comparison. Internally, a miter network, as in Figure 2(a), is constructed comparing combinational logic functions F and G. Figure 1 illustrates the overall tool flow.

First, the Verific RTL parser front-end[6] is used to compile input RTL into the Verific Netlist Database. VeriABC[23] processes the Verific netlist, flattens the hierarchy and produces an intermediate DAG representation in static single assignment (SSA) form, consisting of Boolean and word-level operators as shown in Table I. Except for the hierarchical information, the SSA is a close-to-verbatim representation of the original RTL description. From SSA, a bit-blasting procedure generates a corresponding bit-level network as an AIG (And-inverter graph). Word-level simulation models can be created at the SSA level. ABC[1] equivalence checking solvers are integrated as external solvers.

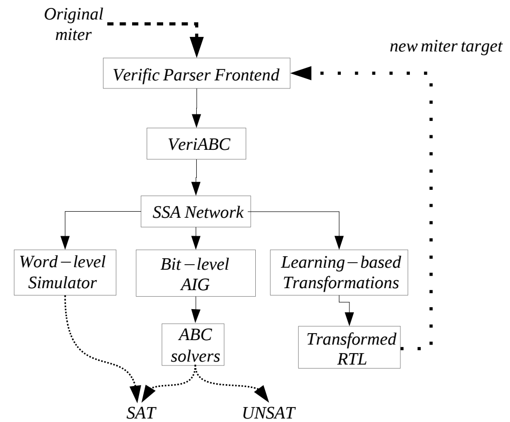


Fig. 1. Overall tool flow

LEC tries to solve the miter directly using random simulation on the word-level simulator or by ABC[1]'s equivalence checking procedure $dcec$, which is a re-implementation of $iprove$[24]. If unresolved, LEC applies transformations to the SSA and produces sub-models in Verilog miter format from which LEC can be recursively applied. The overall system integration is described in Section IV.

III. Learning techniques

In this section, we present the techniques implemented in LEC. Even though some are simple and intuitive, they are powerful when integrated together as demonstrated in the experimental results. All techniques are essential bdecause LEC may not achieve a final proof if any one is omitted. Their interactions are illustrated in the case-study in Section V.

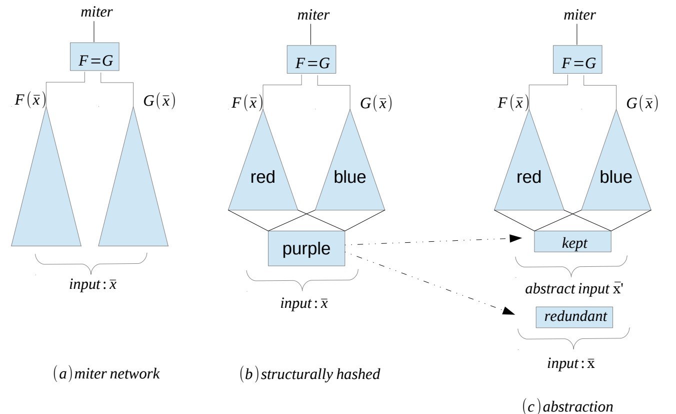


Fig. 2. Miter network

A. Structural information

An SSA netlist is a DAG of bit and word-level operators annotated with bit-width and sign information. In the tool flow, both Verific and VeriABC perform simple structural hashing at the SSA level, merging common sub-expressions. After merging, the miter logic is divided into three colored regions using cone of influence (COI) relations, as in Figure 2 (b).

- Red: if the node is in the COI of F only

- Blue: if the node is in the COI of G only
- Purple: the node is in the COI of both sides of the miter i.e. common logic

The purple region is the portion of the miter logic that has been proved equivalent already, while the red and blue regions are the unresolved ones. LEC makes progress by reducing the red/blue regions and increasing the purple region. The common logic constrains the logic for the red and blue regions, which may be abstracted (see Section III-E) to reduce redundancy and possibly expose the real bottleneck in a proof.

B. Simulation model

Two word-level simulators are generated from the SSA network. One is a native interpreted model. The other uses the open-source Verilator[29] for compiled simulation. From the SSA network, LEC automatically generates C++ code for pseudo-random input drivers and for monitoring design behavior. Verilator compiles the Verilog miter logic, links in the generated C++ code and produces a simulator as a standalone executable. Efficient and effective simulation is crucial in our current flow in capturing potential constants and potential internal equivalent points at the SSA level. Simulation is also used to reduce common logic in the abstraction computation procedure.

C. Bit-level model

As shown in Figure 1, an AIG is created from the SSA network by bit-blasting. LEC calls ABC[1]'s SAT sweeping procedure $dcec$ to perform direct solving at the bit level. Using the AIG model, the native SAT solver embedded in LEC can be used to obtain a formal proof for a particular query. Typical queries are for extracting constant nodes, proving suspected equivalent pairs of points or conducting particular learning for rewriting. Book-keeping information between the SSA nodes and the AIG nodes allows queries to be constructed at the word-level and verified at the bit-level. The result is then used to simplify the SSA network.

D. Constants and Potential Equivalent Points (PEPs)

At the word-level, candidates for constants and PEPs are identified through simulation and SAT queries are posed. Each such SAT query is constructed and checked at the bit-level. SAT-solving is configured at a low-effort level (run for a few seconds) for these types of queries. Proven constants and PEPs are used immediately to simplify the SSA network, leading to a new sub-model of less complexity. LEC then continues to process the sub-model. In the presence of unproven PEPs, LEC can choose one as the next miter target, normally the smallest in terms of the number of nodes in its COI. The proof progresses as constants and PEPs are identified and used to simplify the miter model.

E. Abstraction

As illustrated in Figure 2 (c), in computing an abstraction, LEC computes a cut in the purple region (common logic), and removes the logic between the cut and the inputs. An abstract model is formed by replacing the cut signals with free inputs $\bar{x'}$. If this abstracted miter is UNSAT, then the original miter is UNSAT. In our current implementation, LEC traverses the SSA network in topological order from the inputs. As each node is tentatively replaced with new PIs, simulation is used to validate the replacement. If successful, the node is omitted and replaced with the new PIs and the next node is processed similarly.

A successful abstraction step removes irrelevant logic and exposes a smaller unresolved region of the miter logic, allowing LEC to continue using other procedures. In addition, as seen from experimental results, the reduction of common logic can reduce significantly the amount of complexity for downstream SAT-solving, e.g. when common multipliers being removed from the miter logic. An unsuccessful abstraction when the abstract miter becomes SAT, indicates the existence of a rare event not being captured during random simulations. Often, this gives hints for selecting case-splitting candidates.

F. Rewriting

Similar to [20], word-level rewriting transforms an SSA network into a structurally different but functionally equivalent one. Through rewriting, certain equivalence checking problems can become much simpler. In our experience, a multiplier is often a source of difficulty in data-path equivalence checking. If two multipliers from opposite sides of the miter are matched exactly, LEC can simplify the miter through structural hashing and treat them as common logic. This is most effective when combined with the abstraction procedure as the common multiplier can now be totally removed.

In LEC, a few rules are hard-coded through pattern matching applied to the SSA network. The goal is to process multiplications so that they can be matched exactly. This rewriting is implementation specific; for illustration purposes, we list a few rewriting rules in Table II using Verilog notation and the semantics of the operators.

The first rule is the normalization of multiplier operands. If a multiplier uses a partial product generator and a compressor tree, switching the operands of the multiplication becomes a very hard SAT problem because at the bit level the implementation is not symmetrical. It is almost imperative to apply this rule whenever possible. The second and third rules use the distributive laws of multiplication and multiplexing. Rules 4 and 5 remove the shift operator $\gg$ when it is used with $extract$ and $concat$ because it is hard for multiplication to be restructured through the $\gg$ operator. Rule 6 distributes multiplication through the $concat$ of two bit vectors using $+$. It uses the fact that the concatenation $\{a, b[n-1:0]\}$ is equivalent to $a * 2^n + b[n-1:0]$.

The following is a more complex rule that distributes $+$ over the $extract$ operator. The right hand side is corrected with a third term, which is the carry bit from adding the lower n bits of a and b.

$$(a+b)[m:n] = \\ a[m:n] + b[m:n] + (a[n-1:0] + b[n-1:0])[n] \quad (1)$$

	Before	After
1	$a * b$	$b * a$
2	$mux(cond, d0, d1) * c$	$mux(cond, d0 * c, d1 * c)$
3	$mux(cond, d0, d1)[m : n]$	$mux(cond, d0[m : n], d1[m : n])$
4	$a[m : 0] \gg n$	{ (m-n)'b0, a[m:n] }
5	$(a[m : 0] \gg n)[m - n : 0]$	$a[m : n]$
6	$\{a, b[n - 1 : 0]\} * c$	$a * c \ll n + b[n - 1 : 0] * c$

TABLE II
REWRITING RULES

Repeatedly applying the above rules, LEC transforms the SSA network and keeps only the $*$ and $+$ operators, enhancing the possibility of multipliers to be matched. Note that the above rule (1) and Rule 4-6 in Table II are correct for unsigned operators. Currently, for signed operators, due to sign extension and the two's complement representation of the operands, we have not implemented a good set of rewriting rules.

1) Conditional rewriting: The following equation

$$(a \ll c) * b = (a * b) \ll c \tag{2}$$

reduces the bit-width of a multiplier on the left hand side to a smaller one on the right. It is correct if a, b, c are integers but incorrect in Verilog semantics, which uses modulo integer arithmetic. However, if the following is true within the miter model in modulo integer semantics

$$(\,(a \ll c) \gg c)\; == a \tag{3}$$

then equation (2) is valid. In such a situation, LEC identifies the pattern on the left hand side of (2) in the SSA network and executes a SAT query concerning (3) using the AIG model through bit-level solvers. The transformation to the left hand side of (2) is carried out only if the query is proven to be an invariant. Such a transformation may produce an exact match of $a*b$ afterwards, which can be crucial for achieving the final proof.

G. Case-split

Case-splitting on a binary signal, cofactors the original model into two sub-models. The miter is proven if both sub-models are proven, or falsified if any sub-model is falsified. Although exponential in nature, if many signals are chosen, case-splitting can simplify the underlying bit-level SAT solving significantly. For example, it is difficult to prove the following miter structure directly through bit-blasting and SAT solving at the AIG level

$$(x + y) * (x + y) == x * x + 2 * x * y + y * y \tag{4}$$

where x is a 32-bit integer and y a single binary signal. However, it can be proven easily if case-splitting is done on $y = 0$ and $y = 1$. After constant propagation, the bit-level solver can prove both sub-models easily.

The current case-splitting mechanism supports cofactoring on an input bit or input bit-vector. In verifying the test cases experienced so far, the case splits are conducted on a bit, a bit-vector equal to zero or not, or on the lsb or msb of a bit-vector equals to zero or not. A heuristic procedure can be implemented to trace back from the sel port of a mux node through its Boolean fanins and choose the candidates that have the highest controllability.

Another advantage of case-splitting is that the co-factored sub-models contain new candidates for constants and PEPs, which lead to other down-stream transformations not possible before. Case-splitting also reduces the amount of Boolean logic in the SSA network and exposes the data-path logic to high-level learning such as polynomial construction.

H. Polynomial construction

Reasoning at the word-level, rewriting rules are based on the arithmetic properties of the corresponding operators such as the commutative law of integer multiplication. However, rewriting applies only local transformations and does not have a global view. In situations when the miter logic is constructed from arithmetic optimization at the polynomial level, local rewriting is not able to bring similarity into the miter for further simplification. In such a situation, LEC tries to reconstruct the polynomial of the whole miter model to establish equivalence through arithmetic or algebraic equivalences and then use high level transformations to prove the equivalence of the original miter.

As a generic procedure, LEC follows four steps to prove a miter network $F(\bar{x}) = G(\bar{x})$ where F and G are the top level signals being compared, and $\bar{x}$ is the vector of input variables (bit-vectors):

1) Conjecture (possibly by design knowledge) about the algebraic domain of the polynomial, e.g. signed vs. unsigned integer, modulo integer arithmetic, the order of the polynomial etc. These conjectures set up the framework and semantics for polynomial reconstruction as illustrated in the case-study of Section V.
2) Determine a polynomial f and create a logic network F' such that the following can be proved formally.

$$F' \; implements \, f \tag{5}$$
$$miter \quad F' = F \tag{6}$$

How f is constructed is domain and test-case dependent. In the case-study of Section V, we use simulation patterns to probe for the coefficients of a linear function.
3) Determine a polynomial g and create a logic network G' such that the following can be proved formally.

$$G' \; implements \, g \tag{7}$$
$$miter \quad G' = G \tag{8}$$

4) Establish the following equivalence formally at the algebraic level.

$$f = g \tag{9}$$

The combination of Items 2, 3, and 4 establishes the equivalence proof of the original miter model $F = G$. In constructing F' and G', we try to make them as structurally similar to F and G as possible. Details are given in Section V.

IV. System Integration

The above learning techniques are integrated in LEC as a set of logically independent procedures. Each procedure produces one or more sub-models, illustrated as a tree in Figure 3. The root node is the current targeted Verilog miter model. It has eight children. The $simulator$ and AIG models are the ones described in Figure 1. The $simplified$ sub-model is generated by constant propagation and merging proven PEPs. The $abstraction$ and $rewrite$ sub-models are created by the abstraction and rewrite procedures in the previous section. The $case\text{-}split$ sub-model consists of a set of sub-models, corresponding to the cofactoring variables selected. In the current implementation, the user needs to input the set of signals to case-split on; eventually they will be selected by heuristics. The $linear\text{-}construction$ node has two sub-models which will be explained in detail in Section V. When PEPs are identified through simulation, a PEP node is create with the set of unproven-PEPs as sub-models.

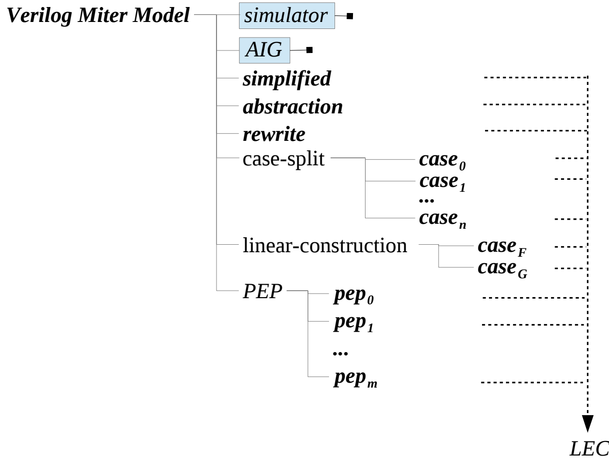


Fig. 3. Branching sub-model tree

Two nodes in the sub-model tree are terminal. One is the simulator model which can falsify the miter through random simulation. The other is the AIG model where ABC's bit-level $dcec$ procedure is applied. The rest of the leaf models (in bold font) are generated as Verilog miter models, which have the same format as the root node. LEC procedures can be applied recursively to these leaf nodes to extend the sub-model trees to simpler ones. The LEC proof process progresses by expanding the sub-model tree. A sub-model is resolved as SAT or UNSAT from its sub-models' proof results.

Since there are no logical dependencies between sibling sub-models, any branch can be chosen to continue the proof process. Sibling sub-models can be forked in parallel from a parent process. A node in the sub-model tree determines its proof result from its children. Table III gives the possible return values from the first level sub-models. SIMPLIFY is returned by a PEP node to its parent model when at least one of its sub-models, pep_i, is proven UNSAT, notifying the parent node to simplify further with the newly proved pep_i.

Depending on the logical relationships between a parent and its immediate sub-models, a node is either disjunctive or conjunctive in semantics. In Figure 3, a Verilog miter model node is disjunctive, which includes the root and all the leaf nodes in bold font. The $case\text{-}split$ and $linear\ construction$ nodes are conjunctive; a PEP node is disjunctive. The semantics, shown in the following tables, are used to resolve the proof result of the parent model from its immediate sub-models. To complete the calculus, we introduced two values: CON and BOT, where CON stands for an internal conflict indicating a potential LEC software bug and BOT is the bottom of the value lattice and acts like an uninitialized value.

Sub-model	Return
simulator	SAT
AIG	SAT/UNSAT
simplified	SAT/UNSAT
abstraction	UNSAT
rewrite	SAT/UNSAT
case-split	SAT/UNSAT
linear construction	SAT/UNSAT
PEP	SIMPLIFY

TABLE III
Sub model return value

‖	SAT	UNS	UNK	SMP	CON	BOT
SAT	SAT	CON	SAT	SAT	CON	SAT
UNS	CON	UNS	UNS	UNS	CON	UNS
UNK	SAT	UNS	UNK	SMP	CON	UNK
SMP	SAT	UNS	SMP	SMP	CON	SMP
CON	CON	CON	CON	CON	CON	CON
BOT	SAT	UNS	UNK	SMP	CON	BOT

TABLE IV
Disjunctions of models

&	SAT	UNS	UNK	SMP	CON	BOT
SAT	SAT	SAT	SAT	n/a	CON	SAT
UNS	SAT	UNS	UNK	n/a	CON	UNS
UNK	SAT	UNK	UNK	n/a	CON	UNK
SMP	n/a	n/a	n/a	n/a	n/a	n/a
CON	CON	CON	CON	n/a	CON	CON
BOT	SAT	UNS	UNK	n/a	CON	BOT

TABLE V
Conjunction of models

Tables IV and V are the truth tables for the disjunction and conjunction semantics of the return values, in which UNS, UNK, SMP stand for UNSAT, UNKNOWN, and SIMPLIFY. Assuming a bug free situation, at a disjunctive node, if either SAT or UNSAT is returned from a sub-model, this is the final proof result for the parent. In conjunction, the parent must wait until all sub-models are resolved as UNSAT before deciding that its result is UNSAT, while any SAT sub-model implies the current model is SAT. A PEP node returns SIMPLIFY to its parent if one of its sub-models, say pep_i, is proven UNSAT. In this case, the parent model can apply another round of simplification to obtain a new $simplified$ sub-model by merging the node pair in the just-proved pep_i. The proof log in Figure VI is a sample sub-model tree where only the branches that contributed to the final proof are shown. Indentation indicates the parent-child relationship. Recursively, the proof result of the top level target is evaluated as UNSAT.

```
{
  "case split": {
    "case_0": "UNSAT by AIG"
    "case_1": {
      "simplified": {
        "abstraction": {
          "case split": {
            "case_00": "UNSAT by AIG",
            "case_01": "UNSAT by AIG",
            "case_10": "UNSAT by AIG",
            "case_11": "UNSAT by AIG"
          },
        },
      },
    },
  },
}
-------------------------------------
Miter proof result: [Resolved: UNSAT]
-------------------------------------
```

Fig. 4. Illustration of proof log

Using this sub-model tree infrastructure, any new procedures discovered in the future can be plugged into the system easily. Also, the system is fully parallelizable in that siblings can be executed at the same time. The proof process can be retrieved from the expanded sub-model tree.

V. Case Study

The design in this case-study is an industrial example taken from the image processing domain. We verify specification = implementation where the "specification" is a manually-specified high-level description of the design. "Implementation" is a machine-generated and highly optimized RTL implementation of the same design using[2]. The miter logic is obtained through SLEC[3]. Therefore, the miter problem is verifying that the high-level synthesis (HLS) tool did not modify the design behavior.

This miter is sequential in nature, but here we examine a bounded model checking (BMC) problem which checks the correctness of the implementation at cycle N. This renders the problem combinational. This is industrially relevant because the sequential problem is too hard to solve in general, and even the BMC problem at cycle N becomes too difficult for industrial tools.

The original design (specification) consists of 150 lines of C++. It went through the Calypto frontend[3] and was synthesized into a word-level netlist in Verilog. The generated miter model has 1090 lines of structural Verilog code with 36 input ports: 29 of which are 7 bits wide, 2 are 9 bits, 4 are 28 bits and one is a single-bit wire. The miter is comparing two 28-bit values. We do not have knowledge about what the design does except through structural statistics: no multipliers, many adders, subtractors, comparators, shifters etc., together with Boolean logic. From a schematic produced from the Verilog, there seems to be a sorting network implemented using comparators, but we can not tell anything further.

Figure 5 illustrates the compositional proof produced by the LEC system by showing the sub-model tree created during the proof process. Indentations indicate parent and sub-model relations and are listed in the order they were created. The three numbers on the right are the node counts in the red, blue and purple regions (common logic) of the SSA network as distinguished in Figure 2(a). Only those sub-models that contributed to the final proof are shown in the figure. Others are ignored. As seen in Figure 5, the case-split procedure is

```
1.original model             :  366  332 776
2.  case-split
3.    case_0                 :  366  331 844
4.      AIG                  :  UNSAT
6.    case_1                 :  366  332 776
7.      simplified           :  344  289 675
8.        abstraction        :  344  289  29
9.          case-split
10.           case_0         :  344  289  31
11.             AIG          :  UNSAT
12.           case_1         :  344  289  31
13.             simplified   :  343  288  27
14.               PEP
15.                 pep_0    :  335  280  27
16.                   linear construction
17.                     case_F
18.                       AIG :  UNSAT
19.                     case_G
20.                       AIG :  UNSAT
21.               simplified  :  10  10  305
22.                 AIG       :  UNSAT
```

Fig. 5. Sub-model proof tree

applied twice, at lines 2 and 9. Both models have a single-bit input port, which was selected for cofactoring. ABC[1] immediately proved the first cofactored case, case_0 (3 and 10) , using the AIG model at 4 and 11. The time-out for the $dcec$ run was set to two seconds. Abstraction was applied at 8, significantly reducing the common logic from 675 to 29 SSA nodes, and effectively removing all the comparator logic. We tried abstraction on the original model without the $case\text{-}split$ procedure and it failed to produce any result. The $case\text{-}split$ at 2 removed enough Boolean logic and eliminated some corner cases such that the abstraction procedure was able to produce an abstract model successfully.

Model 15 is the smallest unproved PEP from model 13. It is proved using the linear construction procedure at 16, which we shall describe in detail in Section V-A. Model 21 is the simplified model of model 13 after merging the just-proved pep_0. After simplification, most of the logic in model 21 became common logic through structural hashing, leaving only 10 nodes in each of the blue and red regions. Model 21 was proved quickly by ABC which concludes the proof of the original miter. In this case, the linear-construction procedure was crucial in attaining the proof. However, the case-split, simplification, abstraction, and PEP models also are very important because they collaborate in removing Boolean, mux and comparator logic etc, but keeping only the part of the original miter logic which constitutes a linear function. Only at this point, can a proof by the linear construction procedure succeed.

A. Linear construction

For model 15 in Figure 5, the SSA network contains many $+, -, \ll$ and $\gg$ operators along with $extract$ and $concat$ oper-

ators, but contains no Boolean operators or muxes. The input ports consist of twenty-five 7-bit or 12-bit wide ports. The miter is comparing two 15-bit wide output ports. At this point, simplification and abstraction can not simplify the model further. Also, there are no good candidates for case-splitting. The local rewriting rules can not be applied effectively without having some global information to help converge the two sides of the miter logic. High-level information must be extracted and applied to prove this miter model.

After the linear construction procedure through LEC, the miter logic is found to be implementing the following linear sum in the signed integer domain using two's complement representation:

$$\begin{aligned}
-16*x_0+2*x_1+2*x_2+2*x_3+2*x_4+2*x_5 \\
+2*x_6+2*x_7+2*x_7+2*x_8+2*x_9 \\
+2*x_{10}+x_{11}+x_{12}+2*x_{13}+2*x_{14}+2*x_{15} \\
+2*x_{16}+2*x_{17}+2*x_{18}+2*x_{19}-2*x_{20}+2*x_{21} \\
+2*x_{22}+2*x_{23}+2*x_{24}+14
\end{aligned}$$

One side of the miter implements the above sum as a plain linear adder chain (Figure 6(a)), the other side is a highly optimized implementation using a balanced binary tree structure (Figure 6(b)) and optimization tricks, which we don't fully understand. This is a hard problem for bit-level engines because

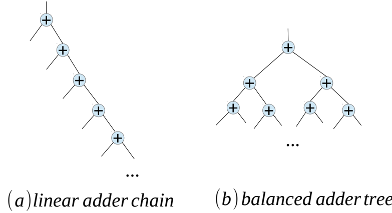


Fig. 6. Addition implementation

there are no internal match points to utilize. Therefore, LEC resorts to trying a high-level method to establish equivalence at the polynomial level. The following are the detailed steps for this specific case.

1) The conjecture: Assume the miter logic is $F(\bar{x}) = G(\bar{x})$ as in Figure 2(a). LEC conjectures the following for the arithmetic domain.

- Signed integer arithmetic. The numbers are in 2's complement representation.
- Assume $F(\bar{x})$ and $G(\bar{x})$ are implementing the linear sums f and g of the forms

$$f(\bar{x}) = \sum a_i \cdot x_i + b \quad (10)$$

$$g(\bar{x}) = \sum a'_i \cdot x_i + b' \quad (11)$$

2) Determining the coefficients of f, g and proving $f = g$ algebraically: Given the data-path logic $F(\bar{x})$ and the linear sum formula (10), it takes $n+1$ simulation patterns on the n input variables to compute the coefficients:

$$\begin{aligned}
b &= F(0,0,...,0) \\
a_0 &= F(1,0,...,0) - b \\
a_1 &= F(0,1,...,0) - b \\
&... \\
a_{n-1} &= F(0,0,...,1) - b
\end{aligned}$$

Another round of random simulation on both the logic and the polynomial can be done to increase the likelihood of the conjecture. The same is repeated for $G(\bar{x})$ to obtain $g(\bar{x})$.

In integer arithmetic, f is equal to g if and only if the coefficients match exactly for each term:

$$f = g <=> \forall\, i\ a_i = a'_i \quad and \quad b = b' \quad (12)$$

So checking of $f = g$ is trivial in this case. In other algebraic domains, domain specific reasoning may have to be applied to derive algebraic equivalence e.g. in [28].

3) Synthesizing implementations F'/G' for f (f=g), structurally similar to F/G: We want to find a Verilog implementation $F'(\bar{x})$ of f such that

1) F' implements f
2) F' is structurally similar to F

To do this, all nodes in the SSA network with arithmetic operators $+$, $-$ are marked, and edges connecting single bits are removed. A reduced graph is then created from the marked nodes in the remaining graph maintaining the input/output relations between marked nodes. This graph is a skeleton of the implementation structure of F. For each of its nodes, we annotate it with a conjectured linear sum computed in the same way as in the above steps. The root node F is annotated with f and internal nodes annotated with linear sums f_s, f_t, etc. For illustration purposes, Figure 7(a) shows such an annotated reduced graph for node w. For an arbitrary node w

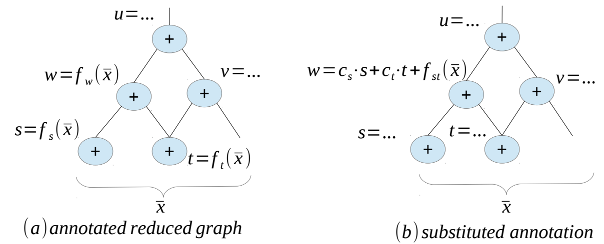


Fig. 7. Annotated reduced graph

in the reduced graph with inputs from nodes s and t, from the annotation we have the following:

$$\begin{aligned}
s &= f_s(\bar{x}) \\
t &= f_t(\bar{x}) \\
w &= f_w(\bar{x})
\end{aligned}$$

We would like to substitute f_w with variable s and t, such that w is a function of s and t in order to follow the structure of the skeleton reduced graph. Because all the functions are linear sums, we can compute, using algebraic division, two constants c_s and c_t such that the following holds:

$$w = c_s \cdot s + c_t \cdot t + f_{st}(\bar{x})$$

c_s is the quotient of f_w/f_s and $c_t = (f_w - c_s \cdot f_s)/f_t$, while f_{st} is the remainder of the previous division. The substitution is conducted in one topological traversal from the inputs to the miter output. After substitution, the annotated reduced graph is essentially a multi-level implementation of the above linear sum. Because the linear sum annotated at each node in the reduced graph is only a conjecture, it may not be exactly the same function as in the original miter logic. However, in re-implementing f using this structure, certain similarities are still captured through the construction process.

This multi-level circuit is implemented by traversing the substituted reduced graph, and creating a corresponding Verilog file for F'. It is generated by allocating a bit-vector at each internal node with its bit-width equivalent to output port of F. The same can be done for G to obtain G'. The reason why LEC goes through so much trouble to obtain separate RTL implementations for F' and G' (even though they are both implementing f), is that we need to prove $F = F'$ and $G = G'$ separately next. Without the structural similarities created through this procedure, proving equivalence with an arbitrary F' and G' would be as difficult as proving the original $F = G$. Generally, only with these extra similarities injected, can the miter model be simplified enough to allow SAT sweeping to succeed for $F = F'$ and $G = G'$.

4) Proving $F = F'$ and $G = G'$: We construct two miter models $F = F'$ and $G = G'$ as $case_F$ and $case_G$ in Figure 3, and apply LEC separately to each. By construction, each miter should be simpler than the original $F = G$ because of the increased structural similarity between the two sides of the miter. Another round of LEC might reduce this miter logic, if not prove it directly through bit-level solvers. In the present case, $F = F'$ was proven instantly because F is a simple linear adder chain, and so is F'. Proving $G = G'$ takes more time using ABCs $dcec$ because G is a highly optimized implementation of f and the multi-level implementation from the annotated reduced graph only captures part of the similarity. But, the injected similarity was sufficient enough to reduce the complexity to be within $dcec$'s capacity.

5) Proving that F'/G' implements f/g : To complete the proof, we still have the proof obligation that F' and G' implement f and g respectively. By construction from the reduced graph, the generated Verilog is a verbatim translation from the multi-level form of f. However, we need to bridge the gap between Verilog's bit-vector arithmetic vs. the integer arithmetic of the linear sum. To do so, we created SVA assertions to check that every Verilog statement captures the integer value in full without losing precision due to underflows or overflows.

$$c[n:0] = a[n-1:0] + b[n-1:0];$$
$$assert\ (a[n-1]\,\&\,b[n-1] \implies c[n])$$
$$assert\ (!a[n-1]\,\&\,!b[n-1] \implies !c[n])$$

$$c[n:0] = a[n:0]/b[m:0];$$
$$assert\ (a[n:0] == (c[n:0] * b[m:0])[n:0])$$

$$c[m:0] = \{a[n:0]\}[m:0];$$
$$assert\ a[n:0] == \{(n-m) * \{c[m-1]\}, c[m:0]\}$$

The first two sets of assertions ensure there is no overflow of signed integer add and no non-zero remainder of division. The third one ensures that extraction does not change the value in two's complement representation. The SVA checkers are formally verified separately using the VeriABC[23] flow, which in turn uses ABCs model checker.

From the above procedures, we established the following:

$$f(\bar{x}) = g(\bar{x}) \tag{13}$$
$$F'(\bar{x})\ implements\ f(\bar{x}) \tag{14}$$
$$G'(\bar{x})\ implements\ g(\bar{x}) \tag{15}$$
$$F = F' \tag{16}$$
$$G = G' \tag{17}$$

Altogether they establish the proof for $F = G$.

Combining the above procedures into a single run, LEC took about 10 minutes on an i7 processor to complete the full proof. Roughly 80% of the time is spent on compiling and running random simulation, the rest are used for SAT solving. In table VI, we also compare this run-time against Boolector[11], z3 [16] and ABC' iprove [24] solver, all run on the same server. It is clear that LEC expedites the proof significantly by using the knowledge of the linear sum formulation inside the miter logic.

Miter	Boolector	Z3	iprove	LEC
model 1	time-out	time-out	time-out	10min

TABLE VI
COMPARISON WITH OTHER SOLVES (TIME-OUT IN 24 HOURS)

In summary, polynomial reconstruction was a key technique to prove the underlying miter problem. The case-study illustrates the major steps and the proof obligations encountered during the process. The actual techniques used for different domains of the underlying arithmetic would differ. Each algebraic domain would require special purpose heuristics and automated proof procedures to guarantee the correctness of the reconstructions and transformations used in the method. The goal of the linear construction procedure was to inject increased structural similarity by using global algebraic transformations.

VI. EXPERIMENTAL RESULTS

Table VI shows the experimental results comparing Boolector[11], Z3[16] and iprove[24] using a 24-hour time-out limit on an 2.6Ghz Intel Xeon processor. These models are generated directly using SLEC[3] for checking C-to-RTL equivalence or extracted as a sub-target from PEPs. The first column is the miter design name. The second column is the number of lines of Verilog for the miter model specification. Run-time or time-out results are reported for each solver in columns 3 to 6. Although the miter models are not big in terms of lines of Verilog, they are quite challenging for Boolector, Z3 and iprove. The run-time of LEC is the total CPU time

including Verilog compilation. It was expected that iprove would not prove any of them because it works on the bit-blasted model without any high-level information that the other solvers have.

Design	Lines	Boolector	z3	iprove	LEC
mul_64_64	125	20 sec	200 sec	timeout	10 sec
d1	24	time-out	time-out	time-out	15 sec
d2	507	time-out	time-out	time-out	2 min
d3	191	time-out	time-out	time-out	15 min
d4	473	time-out	time-out	time-out	60 sec
d5_pep_0	674	time-out	9 hour	time-out	4 min

TABLE VII
BENCHMARK COMPARISON (TIMEOUT 24 HOURS)

The miter, mul_64_64, is comparing a 64x64 multiplier with an implementation using four 32x32 multipliers as the following:

$$\{a_H, a_L\} * \{b_H, b_L\} = (a_H * b_H) << 64 + \\ (a_H * b_L + a_L * b_H) << 32 + a_L * b_L$$

where a_H, a_L, b_H, b_L are the 32-bit slices of the original 64-bit bit-vectors. Both Boolector and Z3 are able to prove it. LEC proves it by first utilizing rewriting rules to transform the 64x64 multiplier into four 32x32 multipliers, matching the other four in the RHS of the miter. As they are matched exactly, they become common logic in the miter model. LEC then produces an abstraction and obtains a reduced model with all the multipliers removed: the outputs of the multipliers become free inputs in the abstract model. The abstract model is then proven instantly by ABC's dcec on the AIG model.

The miter d1, extracted from a PEP sub-model, is a demonstration of rewrite rule 6 in Table II using 32-bit multiplication. As both Boolector and Z3 fail to prove equivalence within the time-limit, they likely do not have this rewriting rule implemented.

To prove d2, LEC conducts conditional rewriting using rule (2) by first statically proving an invariant in the form of (3). After the transformation, the multipliers are matched exactly on both sides of the miter and removed in the subsequent abstract model. The final miter model is proved instantly by ABC on the bit level AIG.

The miter model d3 has part of its logic similar to mul_64_64 embedded inside. LEC proves d3 by first applying rewriting rules repeatedly until no more rewriting is possible. Then, LEC computes a reduced model through abstraction. In the reduced model, LEC conducts a case-split on a one-bit input. The case-0 AIG model is proven instantly, while case-1 is proven in about 10 minutes by ABC.

The miter d4 is proven by first conducting a case-split of two bit-vector inputs: cofactoring on whether the bit-vector equals zero or not. Three of the four cofactored cases are proven instantly. The one unresolved goes through a round of simplification and abstraction. On the then obtained sub-model, three one-bit inputs are identified and cofactored through case-split procedures. LEC prove all eight cases quickly within a few seconds.

Miter d5 is extracted from model 15 in Figure 5 which contains the purely linear sum miter described in the case-study section. For this simpler miter, Z3 is able to prove it in 9 hours while both iprove and Boolector time out. This shows that LEC's transformations through collaborating procedures successfully reduce the surrounding logic in the original model, which was preventing Z3 to prove it in 24 hours.

The above experiments demonstrate the effectiveness of LEC's collaborating procedures of simplification, rewriting, case-splitting and abstraction computations. The LEC architecture allows these procedures to be applied recursively through a sub-model tree: the model obtained by one procedure introduces new opportunities for applying other procedures in the next iteration. As exemplified in miter d4, the initial case-split gives rise to new opportunities for simplification as new constants are introduced by cofactoring. Then a new round of abstraction is able to remove enough common logic and expose three one-bit inputs as case-split candidates in the reduced model, which in turn gives rise to another case-split transformation that leads to the final proof. None of this is possible without the transformations being applied in sequence.

VII. COMPARISON WITH RELATED WORK

In bit-level equivalence-checking procedures [24][25], simulation, SAT-sweeping, AIG rewriting and internal equivalence identification are all relevant to data-path equivalence-checking. In LEC, these types of procedures are conducted at the word-level. Word-level rewriting is difficult if only a bit-level model is available. For example, with no knowledge of the boundary of a multiplier, normalizing its operands is impractical at the bit-level . Although abstraction and case-split techniques in LEC can be applied at the bit-level in theory, these are not used due to the difficulty of computing an abstraction boundary or of finding good cofactoring candidates.

SMT solving is relevant because a data-path is a subset of QF_BV theory. Methods such as [7][11][16][14][17][19], are state-of-art QF_BV solvers. These employ different implementations of word-level techniques in rewriting, abstraction, case-splitting, and simplification, and interleave Boolean and word-level reasoning via a generalized DPLL framework or through abstraction refinements of various forms. Hector[20] is closest to LEC in terms of technology and targeted application domains, and has a rich set of word-level rewriting rules along with some theorem prover [7] procedures to validate every rewriting applied. Hector also has an orchestration of a set of bit-level solvers using SAT and BDD engines to employ once the bit-level miter model is constructed. Strategically, LEC relies less on the capacity of SAT solver; instead it builds a compositional proof infrastructure and employs iterative transformations to finally obtain a proof through sub-model trees. The goal of these LEC learning procedures is to reverse engineer the embedded high-level algebraic transformations and bring more similarity between both sides of the miter model.

The techniques in [26] [31][33] also try to reconstruct an algebraic model from the underlying logic, but they employ a bottom up approach and their primitive element is a half-adder.

The method in [8] simplifies the algebraic construction by solving an integer linear programming problem. The limitation of these approaches is that they rely on the structural pattern of the underlying logic to reconstruct the algebraic model. On the other hand, the linear construction case-study in Section V-A constructs the polynomial through probing with simulation patterns. This is more general as it uses only the functional information of the data-path logic. For different domains, other techniques may well be more applicable such as the bottom-up approach. The use of vanishing polynomials and Grobner bases in [27][28] to prove equivalence between polynomials in the modulo integer domain can be utilized once a polynomial form is reconstructed in LEC. In many data-path miter models, such a polynomial in a certain domain or theory is likely embedded in other control and data-path logic. Direct application of algebraic techniques is often not practical. Thus the collaborating procedures in LEC are designed to bridge this gap and isolate such polynomials so that these high level theories can then be applied.

In conducting consistency checking between C and Verilog RTL, the work [21] focuses on how to process a C program to generate formal models. The tool relies on SMT solvers [11][16][14] as the back-end solving engines.

In terms of tool architecture, [9] [10] [22], all employ a sophisticated set of transformations to simplify the target model during verification. These are done at the bit-level. The LEC infrastructure allows future extension to take advantage of multi-core parallelization as demonstrated in [30]. [12] [32], use a dedicated data-structures to represent the proof-obligations, while LEC relies on the sub-model tree to track the compositional proof strategy used at each node.

VIII. Conclusion

In LEC, we build a system of collaborating procedures for data-path equivalence-checking problems found from an industrial setting. The strategy is to utilize Boolean level solvers, conduct the transformations at the word-level and to synthesize internal similarities by lifting the reasoning to the algebraic level . Using a real industrial case-study, we demonstrated the applicability of the sub-tree infrastructure for integrating a compositional proof methodology using LEC.

References

[1] ABC - a system for sequential synthesis and verification. Berkeley Verification and Synthesis Research Center, http://www.bvsrc.org.
[2] Calypto® Catapult Design Product. http://www.calypto.com.
[3] Calypto® SLEC. http://www.calypto.com.
[4] Forte design systems. http://www.forteds.com.
[5] smtlib. http://www.smt-lib.org.
[6] Verific Design Automation: http://www.verific.com.
[7] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Computer Aided Verification*, pages 171–177. Springer, 2011.
[8] M. A. Basith, T. Ahmad, A. Rossi, and M. Ciesielski. Algebraic approach to arithmetic design verification. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 67–71. IEEE, 2011.
[9] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 259–266. IEEE, 2007.
[10] R. Brayton and A. Mishchenko. Abc: An academic industrial-strength verification tool. In *Computer Aided Verification*, pages 24–40. Springer, 2010.
[11] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.
[12] M. L. Case, A. Mishchenko, and R. K. Brayton. Automated extraction of inductive invariants to aid model checking. In *Formal Methods in Computer Aided Design, 2007. FMCAD'07*, pages 165–172. IEEE, 2007.
[13] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur, and N. Sharma. Non-cycle-accurate sequential equivalence checking. In *Proceedings of the 46th Annual Design Automation Conference*, pages 460–465. ACM, 2009.
[14] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The mathsat5 smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.
[15] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, 2011.
[16] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
[17] B. Dutertre and L. De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2:2, 2006.
[18] M. Fujita. Equivalence checking between behavioral and rtl descriptions with virtual controllers and datapaths. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 10(4):610–626, 2005.
[19] S. Jha, R. Limaye, and S. A. Seshia. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In *Computer Aided Verification*, pages 668–674. Springer, 2009.
[20] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley. Solver technology for system-level to rtl equivalence checking. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pages 196–201. IEEE, 2009.
[21] D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.
[22] A. Kuehlmann and J. Baumgartner. Transformation-based verification using generalized retiming. In *Computer Aided Verification*, pages 104–117. Springer, 2001.
[23] J. Long, S. Ray, B. Sterin, A. Mishchenko, and R. Brayton. Enhancing abc for ltl stabilization verification of systemverilog/vhdl models. *Ganesh Gopalakrishnan University of Utah USA*, page 38, 2011.
[24] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een. Improvements to combinational equivalence checking. In *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pages 836–843, 2006.
[25] V. Paruthi and A. Kuehlmann. Equivalence checking combining a structural sat-solver, bdds, and simulation. In *Computer Design, 2000. Proceedings. 2000 International Conference on*, pages 459–464, 2000.
[26] E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, A. Dreyer, F. Seelisch, and G. Greuel. Stable: A new qf-bv smt solver for hard verification problems combining boolean reasoning with computer algebra. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.
[27] N. Shekhar, P. Kalla, and F. Enescu. Equivalence verification of polynomial datapaths using ideal membership testing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(7):1320–1330, 2007.
[28] N. Shekhar, P. Kalla, F. Enescu, and S. Gopalakrishnan. Equivalence verification of polynomial datapaths with fixed-size bit-vectors using finite ring algebra. In *Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*, pages 291–296, 2005.
[29] W. Snyder, P. Wasson, and D. Galbi. Verilator: Convert verilog code to c++/systemc, 2012.
[30] B. Sterin, N. Een, A. Mishchenko, and R. Brayton. The benefit of concurrency in model checking. IWLS, 2011.
[31] D. Stoffel and W. Kunz. Equivalence checking of arithmetic circuits on the arithmetic bit level. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(5):586–597, 2004.
[32] D. Wang and J. Levitt. Automatic assume guarantee analysis for assertion-based formal verification. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 561–566. ACM, 2005.
[33] M. Wedler, D. Stoffel, R. Brinkmann, and W. Kunz. A normalization method for arithmetic data-path verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(11):1909–1922, 2007.

Trading-off incrementality and dynamic restart of multiple solvers in IC3

G. Cabodi (*), A. Mishchenko (**), M. Palena (*)
(*) Dip. di Automatica ed Informatica
Politecnico di Torino - Torino, Italy
(**) Dept. of EECS, University of California, Berkeley, CA, USA

***Abstract*—This paper[1]addresses the problem of SAT solver performance in IC3, one of the major recent breakthroughs in Model Checking algorithms. Unlike other Bounded and Unbounded Model Checking algorithms, IC3 is characterized by numerous SAT solver queries on small sets of problem clauses. Besides algorithmic issues, the above scenario poses serious performance challenges for SAT solver configuration and tuning. As well known in other application fields, finding a good compromise between learning and overhead is key to performance. We address solver cleanup and restart heuristics, as well as clause database minimality, based on on-demand clause loading: transition relation clauses are loaded in solver based on structural dependency and phase analysis. We also compare different solutions for multiple specialized solvers, and we provide an experimental evaluation on benchmarks from the HWMCC suite. Though not finding a clear winner, the work outlines several potential improvements for a portfolio-based verification tool with multiple engines and tunings.**

I. Introduction

IC3 [1] is a SAT-based invariant verification algorithm for bit-level Unbounded Model Checking (UMC). Since its introduction, IC3 has immediately generated strong interest, and is now considered one of the major recent breakthroughs in Model Checking. IC3 proved to be impressively effective on solving industrial verification problems. Our experience with the algorithm shows that IC3 is the single invariant verification algorithm capable of solving the largest number of instances among the benchmarks of the last editions of the Hardware Model Checking Competition (HWMCC).

A. Motivations

IC3 heavily relies on SAT solvers to drive several parts of the verification algorithm: a typical run of IC3 is characterized by a huge amount of SAT queries. As stated by Bradley in [2], the queries posed by IC3 to SAT solvers differ significantly in character from those posed by other SAT-based invariant verification algorithms (such as Bounded Model Checking [3], k-induction [4] [5] or interpolation [6]). Most notably, SAT queries posed by IC3 don't involve the unrolling of the transition relation for more than one step and are thus characterized by small-sized formulas.

IC3 can be thought as composed of two different layers: at the top level, the algorithm itself drives the verification process by constantly refining a set of over-approximations to forward reachable states with new inductive clauses; at the bottom level, a SAT solving framework is exploited by the top-level algorithm to respond to queries about the system. As shown in [7], these two layers can be separated by means of a clean interface.

Performance of IC3 turns out to be both highly sensitive to the various internal behaviours of SAT solvers, and strictly dependent on the way the top-level algorithm is integrated with the underlying SAT solving framework.

The peculiar characteristics exposed by the SAT queries of IC3 can thus be exploited to improve the overall performance of the algorithm in two different manners:

1) Tuning the internal behaviours of the particular SAT solver employed to better fit IC3 needs.
2) Defining better strategies to manage the SAT solving work required by IC3.

In this paper we address this second issue, proposing and comparing different implementation strategies for handling SAT queries in IC3. The aim of this paper is to identify the most efficient way to manage SAT solving in IC3. To achieve this goal we experimentally compare a number of different implementation strategies over a selected set of benchmarks from the recent HWMCC.

The experimental work has been done by two different research groups, on two different state-of-the-art verification tools, ABC [8] and PdTRAV [9], that share similar architectural and algorithmnic choices in their implementation of IC3.

The focus of this paper is neither on the IC3 algorithm itself nor on the internal details of the SAT solving procedures employed, but rather on the implementation details of the integration between IC3 and the underlying SAT solving framework.

B. Contributions

The main contributions of this paper are:

- A characterization of SAT queries posed by IC3.
- Novel approaches to solver allocation, loading and clean up in IC3.
- An experimental evaluation of performance using two verification tools.

[1]This work was supported in part by SRC Contracts No. 2012-TJ-2328 and No. 2265.001

C. Outline

First in Section II we introduce the notation used and give some background on IC3. Then, in Section III we present a systematic characterization of the SAT solving work required by IC3. Section IV introduces the problem of handling SAT queries posed by IC3 efficiently. Both commonly used and novel approaches to the allocation, loading and cleaning up of SAT solvers in IC3 are discussed in Sections V, VI and VII respectively. Experimental data comparing these approaches are presented in Section VIII. Finally, in Section IX we draw some conclusions and give summarizing remarks.

II. Background

A. Notation

Definition 1. *A transition system is the triple $S = \langle \mathbf{M}, I, T\rangle$ where M is a set of boolean variables called state variables of the system, $I(M)$ is a boolean predicate over M representing the set of initial states of the system and $T(M, M')$ is a predicate over M, M' that represents the transition relation of the system.*

Definition 2. *A state of the system is represented by a complete assignment s to the state variables M. A set of states of the system is represented by a boolean predicate over M. Given a boolean predicate F over M, a complete assignement s such that s satisfies F (i.e. $s \models F$) represents a state contained in F and is called an F-state. Primed state variables M' are used to represent future states and, accordingly, a boolean predicate over M' represent a set of future states.*

Definition 3. *A boolean predicate F is said to be stronger than another boolean predicate G if $F \rightarrow G$, i.e. every F-state is also a G-state.*

Definition 4. *A literal is a boolean variable or the negation of a boolean variable. A disjunction of literals is called a clause while a conjunction of literals is called a cube. A formula is said to be in Conjunctive Normal Form (CNF) if it is a conjunction of clauses.*

Definition 5. *Given a transition system $S = \langle \mathbf{M}, I, T\rangle$, if an assignment s, t' satisfies the transition relation T (i.e. if $s, t' \models T$) then s is said to be a predecessor of t and t is said to be a successor of s. A sequence of states $s_0, s_1, \ldots, s_n$ is said to be a path in S if every couple of adjacent states s_i, s_{i+1}, $i \leq 0 < n$ satisfies the transition relation (i.e. if $s_i, s'_{i+1} \models T$).*

Definition 6. *Given a transition system $S = \langle \mathbf{M}, I, T\rangle$, a state s is said to be reachable in S if there exists a path $s_0, s_1, \ldots, s$, such that s_0 is an initial state (i.e. $s_0 \models I$). We denote with $R_n(S)$ the set of states that are reachable in S in at most n steps. We denote with $R(S)$ the overall set of states that are reachable in S. Note that $R(S) = \bigcup_{i \geq 0} R_i(S)$.*

Definition 7. *Given a transition system $S = \langle \mathbf{M}, I, T\rangle$ and a boolean predicate P over M (called A safety property), the invariant verification problem is the problem of determining if P holds for every reachable state in S: $\forall s \in R(S) : s \models P$. An algorithm used to solve the invariant verification problem is called an invariant verificatio algorithm.*

Definition 8. *Given a transition system $S = \langle \mathbf{M}, I, T\rangle$, a boolean predicate F over M is called an inductive invariant for S if the following two conditions hold:*

- *Base case: $I \rightarrow F$*
- *Inductive case: $F \wedge T \rightarrow F'$*

A boolean predicate F over M is called an inductive invariant for S relative to another boolean predicate G if the following two conditions hold:

- *Base case: $I \rightarrow F$*
- *Relative inductive case: $G \wedge F \wedge T \rightarrow F'$*

Lemma 1. *Given a transition system $S = \langle \mathbf{M}, I, T\rangle$, an inductive invariant F for S is an over-approximation to the set of reachable states $R(S)$.*

Definition 9. *Given a transition system $S = \langle \mathbf{M}, I, T\rangle$ and a boolean predicate P over M, an inductive strengthening of P for S is an inductive invariant F for S such that F is stronger than P.*

Lemma 2. *Given a transition system $S = \langle \mathbf{M}, I, T\rangle$ and a boolean predicate P over M, if an inductive strengthening of P can be found, then the property P holds for every reachable state of S. The invariant verification problem can be seen as the problem to find an inductive strengthening of P for S.*

B. IC3

Given a transition system $S = \langle \mathbf{M}, I, T\rangle$ and a safety property P over $\mathbf{M}$, IC3 aims to find an inductive strengthening of P for S. For this purpose, it maintains a sequence of formulas $\mathbf{F_k} = F_0, F_1, \ldots F_k$ such that, for every $0 \leq i < k$, F_i is an over-approximation of the set of states reachable in at most i steps in S. Each of these over-approximations is called a *time frame* and is represented by a set of clauses, denoted by $\text{clauses}(F_i)$. The sequence of time frames $\mathbf{F_k}$ is called *trace* and is maintained by IC3 in such a way that the following conditions hold throughout the algorithm:

(1) $F_0 = I$
(2) $F_i \rightarrow F_{i+1}$, for all $0 \leq i < k$
(3) $F_i \wedge T \rightarrow F'_{i+1}$, for all $0 \leq i < k$
(4) $F_i \rightarrow P$, for all $0 \leq i < k$

Condition (1) states that the first time frame of the trace is special and is simply assigned to the set of initial states of S. The remaining conditions, claim that for every time frame F_i but the last one: (2) every F_i-state is also a F_{i+1}-state, (3) every successor of an F_i-state is an F_{i+1}-state and (4) every F_i-state is safe. Condition (2) is maintained syntactically, enforcing the condition (2') $\text{clauses}(F_{i+1}) \subseteq \text{clauses}(F_i)$.

Lemma 3. *Let $S = \langle \mathbf{M}, I, T\rangle$ be a transition system, $\mathbf{F_k} = F_0, F_1, \ldots F_k$ a sequence of boolean formulas over $\mathbf{M}$ and let conditions (1-3) hold for $\mathbf{F_k}$. Then each F_i, with $0 \leq i < k$, is an over-approximation to the set of states reachable within i steps in S.*

Lemma 4. *Let $S = \langle \mathbf{M}, I, T \rangle$ be a transition system, P a safety property over $\mathbf{M}$, $\mathbf{F_k} = F_0, F_1, \ldots F_k$ a sequence of boolean formulas over $\mathbf{M}$ and let conditions (1-4) hold for $\mathbf{F_k}$. Then P is satisfied up to $k-1$ steps in S (i.e. there doesn't exist any counter-example to P of length less or equal than $k-1$).*

The main procedure of IC3 is described in Algorithm 1 and is composed of two nested iterations. Major iterations (lines 3-16) try to prove that P is satisfied up to k steps in S, for increasing values of k. To prove so, in minor iterations (lines 4-9), IC3 refines the trace $\mathbf{F_k}$ computed so far, by adding new relative inductive clauses to some of its time frames. The algorithm iterates until either (i) an inductive strengthening of the property is produced (line 4), or (ii) a counter-example to the property is found (line 7).

Input: $S = \langle \mathbf{M}, I, T \rangle ; P(\mathbf{M})$
Output: *SUCCESS or FAIL(σ), with σ counter-example*

1: $k \leftarrow 0$
2: $F_k \leftarrow I$
3: **repeat**
4:   **while** $\exists t : t \models F_k \wedge \neg P$ **do**
5:     $s \leftarrow$ *Extend(t)*
6:     **if** *BlockCube*(s, Q, $\boldsymbol{F_k}$) = FAIL(σ) **then**
7:       **return** FAIL(σ)
8:     **end if**
9:   **end while**
10:   $F_{k+1} \leftarrow \emptyset$
11:   $k \leftarrow k+1$
12:   $\boldsymbol{F_k} \leftarrow$ *Propagate*($\boldsymbol{F_k}$)
13:   **if** $F_i = F_{i+1}$ *for some* $0 \leq i < k$ **then**
14:     **return** SUCCESS
15:   **end if**
16: **until** forever

Algorithm 1. *IC3(S, P)*

At major iteration k, the algorithm has computed a trace $\mathbf{F_k}$ such that conditions (1-4) hold. From Lemma 4, it follows that P is satisfied up to $k-1$ steps in S. IC3 then tries to prove that P is satisfied up to k steps as well. This is done by enumerating F_k-states that violate P and then trying to block them in F_k.

Definition 10. *Blocking a state (or, more generally, a cube) s in a time frame F_k means proving s unreachable within k steps in S, and consequently refine F_k to exclude it.*

To enumerate each state of F_k that violates P (line 4), the algorithm poses SAT queries to the underlying SAT solving framework in the following form:

$$\text{SAT ?}(F_k \wedge \neg P) \qquad (Q_1)$$

If Q_1 is SAT, a bad state t in F_k can be extracted from the satisfying assignment. That state must be blocked in F_k. To increase performance of the algorithm, as suggested in [7], the bad state t generated this way is first (possibly) extended to a bad cube s. This is done by means of the *Extend*(t) procedure (line 5), not reported here, that employs ternary simulation to remove some literals from t. The resulting cube s includes t and violates the property P, it is thus a *bad cube*. The algorithm then tries to block the whole bad cube s rather than t. It is showed in [7] that extending bad states into bad cubes before blocking them dramatically improves IC3 performance.

Once a bad cube s is found, it is blocked in F_k calling the *BlockCube*(s, Q, $\boldsymbol{F_k}$) procedure (line 6). This procedure is described in Algorithm 2.

Input: *s: bad cube in F_k; Q: priority queue;* $\boldsymbol{F_k}$*: trace*
Output: *SUCCESS or FAIL(σ), with σ counter-example*

1: add a proof-obligation (s, k) to the queue Q
2: **while** Q is not empty **do**
3:   extract (s, j) with minimal j from Q
4:   **if** $j > k$ *or* $t \not\models F_j$ **then continue**;
5:   **if** $j = 0$ **then return** FAIL(σ)
6:   **if** $\exists t, v' : t, v' \models F_{j-1} \wedge T \wedge \neg s \wedge s'$ **then**
7:     $p \leftarrow$ *Extend(t)*
8:     add $(p, j-1)$ and (s, j) to Q
9:   **else**
10:     $c \leftarrow$ *Generalize*($j, s, \boldsymbol{F_k}$)
11:     $F_i \leftarrow F_i \cup c$ for $0 < i \leq j$
12:     add $(j+1, c)$ to Q
13:   **end if**
14: **end while**
15: **return** SUCCESS

Algorithm 2. *BlockCube(s, Q,* $\boldsymbol{F_k}$*)*

Otherwise, if Q_1 is UNSAT, every bad state of F_k has been blocked so far, conditions (1-4) hold for $k+1$ and IC3 can safely move to the next major iteration, trying to prove that P is satisfied up to $k+1$ steps. Before moving to the next iteration, a new empty time frame F_{k+1} is created (line 10). Initially, $\text{clauses}(F_{k+1}) = \emptyset$ and such time frame represent the entire state space, i.e. $F_{k+1} = \text{Space}(S)$. Note that $\text{Space}(S)$ is a valid over-approximation to the set of states reachable within $k+1$ steps in S. Then a phase called *propagation* takes place (line 12). The procedure *Propagate*($\boldsymbol{F_k}$) (Algorithm 4), which is discussed later, handles that phase. During propagation, IC3 tries to refine every time frame F_i, with $i < 0 \leq k$, by checking if some clauses of one time frame can be pushed forward to the following time frame. Possibly, propagation refines the outmost timeframe F_k so that $F_k \subset \text{Space}(S)$. Propagation can lead to two adjacent time frames becoming equivalent. If that happens, the algorithm has found an inductive strengthening of P S (equal to those time frames), so the property P holds for for every reachable state of S and IC3 return success (line 13).

The procedure *BlockCube*(s, Q, $\boldsymbol{F_k}$) (Algorithm 2) is responsible for refining the trace $\boldsymbol{F_k}$ in order to block a bad cube s found in F_k. To preserve condition (3), prior to blocking a cube in a certain time frame, IC3 has to recursively block its

predecessor states in the preceding time frames. To keep track of the states (or cubes) that must be blocked in certain time frames, IC3 uses the formalism of *proof-obligations*.

Definition 11. *Given a cube s and a time frame F_j, a proof-obligation is a couple (s, j) formalizing the fact that s must be blocked in F_j.*

Given a proof obligation (s, j), the cube s can either represent a set of bad states or a set of states that can all reach a bad state in one or more transitions. The number j indicates the position in the trace where s must be proved unreachable, or else the property fails.

Definition 12. *A proof obligation (s, j) is said to be discharged when s becomes blocked in F_j.*

IC3 maintains a priority queue Q of proof-obligations. During the blocking of a cube, proof-obligations (s, j) are extracted from Q and discharged for increasing values of j, ensuring that every predecessor of a bad cube s will be blocked in F_j $(j < k)$ before s will be blocked in F_k. In the *BlockCube*(s, Q, $\boldsymbol{F_k}$) procedure, first a proof-obligation encoding the fact that s must be blocked in F_k is added to Q (line 1). Then proof-obligations are iteratively extracted from the queue and discharged (lines 2-14).

Prior to discharge the proof-obligation (s, j) extracted, IC3 checks if that proof-obligation still needs to be discharged. It is in fact possible for an enqueued proof-obligation to become discharged as a result of the discharging of some previously extracted proof-obligations. To perform this check, the following SAT query is posed (line 4):

$$SAT?(F_j \wedge s) \qquad (Q_2)$$

If the result of Q_2 is SAT, then the cube s is still in F_j and (s, j) still needs to be discharged. Otherwise, s has already been blocked in F_j and the procedure can move on to the next iteration.

If the proof-obligation (s, j) still needs to be discharged, then IC3 checks if the time frame identified by j is the initial time frame (line 5). If so, the states represented by s are initial states that can reach a violation of the property P. A counter-example σ to P can be constructed going up the chain of proof-obligations that led to $(s, 0)$. In that case, the procedure terminates with failure returning the counter-example found.

To discharge a proof-obligation (s, j), i.e. to block a cube s in F_j, IC3 tries to derive a clause c such that $c \subseteq \neg s$ and c is inductive relative to F_{j-1}. This is done by posing the following SAT query (line 6):

$$SAT?(F_{j-1} \wedge \neg s \wedge T \wedge s') \qquad (Q_3)$$

If Q_3 is UNSAT (lines 10-12), then the clause $\neg s$ is inductive relative to F_{j-1} and can be used to refine F_j, ruling out s. To pursue a stronger refinement of F_j, the inductive clause found undergoes a process called *inductive generalization* (line 10) prior to being added to F_i. Inductive generalization is carried out by the procedure *Generalize*($j, s, \boldsymbol{F_k}$) (Algorithm 3), which tries to minimize the number of literals in clause $c = \neg s$ while maintaining its inductiveness relative to F_{j-1}, in order to preserve condition (2). The resulting clause is added not only to F_j, but also to every time frame F_i, $0 < i < j$ (line 11). Doing so discharges the proof-obligation (s, j). In fact, this refinement rule out s from every F_i with $0 < i \leq j$. Since the sets F_i with $i > j$ are larger than F_j, s may still be present in one of them and $(s, j+1)$ may become a new proof-obligation. To address this issue, Algorithm 2 adds $(s, j+1)$ to the priority queue (line 12).

Otherwise, if Q_3 is SAT (lines 7-8), a predecessor t of s in F_{j-1} can be extracted from the satisfying assignment. To preserve condition (3), before blocking a cube s in a time frame F_j, every predecessor of s must be blocked in F_{j-1}. So, the predecessor t is extended with ternary simulation (line 7) into the cube p, and then both proof-obligations $(p, j-1)$ and (s, j) are added to the queue (line 8).

Input: *j: time frame index; s: cube such that $\neg s$ is inductive relative to F_{j-1}; $\boldsymbol{F_k}$: trace*
Output: *c : a sub-clause of $\neg s$*

```
1:  c ← ¬s
2:  for all literals l in c do
3:      try ← the clause obtained by deleting l from c
4:      if ∄t, v′ : t, v′ ⊨ F_{j−1} ∧ T ∧ try ∧ ¬try′ then
5:          if ∄t ⊨ I ∧ ¬try then
6:              c ← try
7:          end if
8:      end if
9:  end for
10: return c
```

Algorithm 3. *Generalize*($j, s, \boldsymbol{F_k}$)

The *Generalize*($j, s, \boldsymbol{F_k}$) procedure (Algorithm 3) tries to remove literals from $\neg s$ while keeping it relatively inductive to F_{j-1}. To do so, a clause c intialized with $\neg s$ (line 1) is used to represent the current minimal inductive clause. For every literal of c, the clause try obtained by dropping that literal from c (line 3), is checked for inductiveness relative to F_{j-1} by means of the following SAT query (line 4):

$$SAT?(F_{j-1} \wedge try \wedge T \wedge \neg try') \qquad (Q_4)$$

If Q_4 is UNSAT, the iductive case for the reduced formula still holds. Since dropping literals from a relatively inductive clause can break both the inductive case and the base case, the latter must be explicilty checked too for the reduced clause try (line 5). This is done by posing the following SAT query:

$$SAT?(I \wedge \neg try) \qquad (Q_5)$$

If both the inductive case and the base case hold for the reduced clause try, the currently minimal inductive clause c is updated with try (line 6).

The *Propagate*($\boldsymbol{F_k}$) procedure (Algorithm 4) handles the propagation phase. For every clause c of each time frame F_j, with $0 \leq j < k-1$, the procedure checks if c can be pushed

Input: $\boldsymbol{F_k}$: *trace*
Output: $\boldsymbol{F_k}$: *updated trace*

1: **for** $j = 0$ to $k-1$ **do**
2:     **for all** $c \in F_j$ **do**
3:         **if** $\exists t, v' : t, v' \models F_j \wedge T \wedge c'$ **then**
4:             $F_{j+1} \leftarrow F_{j+1} \cup \{c\}$
5:         **end if**
6:     **end for**
7: **end for**
8: **return** $\boldsymbol{F_k}$

Algorithm 4. *Propagate($\boldsymbol{F_k}$)*

forward to F_{j+1} (line 3). To do so, it poses the following SAT query:

$$SAT?(F_j \wedge T \wedge c') \qquad (Q_6)$$

If the result of Q_6 is SAT, then it is safe, with respect to condition (3), to push clause c forward to F_{i+1}. Otherwise, c can't be pushed and the procedure iterates to the next clause.

C. Related works

In [2], Aaron Bradley outlined the opportunity for SAT and SMT researchers to directly address the problem of improving IC3's performance by exploiting the peculiar character of the SAT queries it poses. A description of the IC3 algorithm, specifically addressing implementation issues, is given in [7].

III. SAT SOLVING IN IC3

SAT queries posed by IC3 have some specific characteristics:

- *Small-sized formulas*: they employ no more than a single instance of the transition relation;
- *Large number of calls*: reasoning during the verification process is highly localized and takes place at relatively-small-cubes granularity;
- *Separated contexts*: each SAT query is relative to a single time frame;
- *Related calls*: subsequent calls may expose a certain correlation (for instance, inductive generalization calls take place on progressively reduced formulas).

We performed an analysis of the implementation of IC3 within the academic model checking suite PdTRAV, closely following the description of IC3 given in [7] (there called PDR: Property Directed Reachability). The experimental analysis lead us to identify six different types of SAT queries that the algorithm poses during its execution. These queries are the ones already outlined in Section II-B. The type of these queries is reported in Table I.

For each of the queries identified, we have measured the average number of calls and the average solving time. These statistics are reported in Table II. The results were collected by running PdTRAV's implementation of IC3 on the complete set of 310 single property benchmarks of the HWMCC'12, using time and memory limits of 900 s and 2 GB, respectively.

Such statistics can be summarized as follows:

Name	Query Type	Query
Q_1	Target Intersection	$SAT?(F_k \wedge \neg P)$
Q_2	Blocked Cube	$SAT?(F_i \wedge s)$
Q_3	Relative Induction	$SAT?(F_i \wedge \neg s \wedge T \wedge s')$
Q_4	Inductive Generalization	$SAT?(F_i \wedge c \wedge T \wedge \neg c')$
Q_5	Base of Induction	$SAT?(I \wedge \neg c)$
Q_6	Clause Propagation	$SAT?(F_i \wedge T \wedge \neg c')$

Table I: SAT Queries Breakdown in IC3

- SAT calls involved in inductive generalization are by far the most frequent ones. These are in fact the calls that appears at the finest granularity. In the worst case scenario, one call is posed for every literal of every inductive clause found.
- Inductive generalization and propagation checks are the most expensive queries in terms of average SAT solving time required.
- Target intersection calls are very infrequent and don't take much time to be solved.
- Blocked cube and relative induction checks are close in the number of calls and solving time.

Query	Calls		Avg Time
	[Number]	[%]	[ms]
Q_1	483	0.1	81
Q_2	27891	6.8	219
Q_3	31172	7.6	334
Q_4	142327	34.7	575
Q_5	147248	35.9	112
Q_6	61114	14.9	681

Table II: SAT queries statistics in IC3: Number of calls, percentage, and average time spent in different SAT queries during an IC3 run.

IV. HANDLE SAT SOLVING IN IC3

Subsequent SAT calls in IC3 are often applied to highly different formulas. In the general case, two subsequent calls can in fact be posed in the context of different time frames, thus involving different sets of clauses. In addition, one of them may require the use of the transition relation, while the other may not, and each query can involve the use of temporary clauses/cubes that are needed only to respond to that particular query (e.g. the candidate inductive clause used to check relative inductiveness during cube blocking or inductive generalization).

In the general case, whenever a new query is posed by IC3 to the underlying SAT solving framework, the formula currently loaded in the solver must be modified to accomodate the new query. For this reason, IC3 requires the use of SAT solvers that expose an incremental SAT interface. An incremental SAT interface for IC3 must support the following features:

- Pushing and popping clauses to or from the formula.

- Specifying literal assumptions.

Many state-of-the-art SAT solvers, like MiniSAT [10], feature an incremental interface capable of pushing new clauses into the formula and allowing literal assumptions. Removing clauses from the current formula is a more difficult task since one have to remove not only the single clause specified, but also every learned clause that has been derived from it. Although solvers such as *zchaff* [11] directly support clause removal, the majority of the state-of-the-art SAT solvers feature an interface like the one of MiniSAT, in which clause removal can only be simulated. This is done through the use of literal assumptions and the introduction of auxiliary variables known as *activation variables*, as described in [12]. In such solvers, clauses aren't actually removed from the formula but only made redundant for the purpose of determining the satisfiability of the rest of the formula. Since such clauses are not removed from the formula, they still participate in the Boolean Constraint Propagation (BCP) and, thus, degrade the overall SAT solving performance. In order to minimize this degradation, each solver employed by IC3 must be periodically cleaned up, i.e. emptied and re-loaded with only relevant clauses. In this work we assume the use of a SAT solver exposing an incremental interface similar to the one of MiniSAT.

Once an efficient incremental SAT solving tool has been chosen, any implementation of IC3 must face the problem of deciding how to integrate the top-level algorithm with the underlying SAT solving layer. Such problem can be divided into the following three sub-problems:

- *SAT solvers allocation*: decide how many different SAT solvers to employ and how to distribute workload among them.
- *SAT solvers loading*: decide which clauses must be loaded in each solver to make them capable of responding correctly to the SAT queries posed by IC3.
- *SAT solvers clean up*: decide when and how often the algorithm must clean up each solver, in order to avoid performance degradation.

V. SAT SOLVERS ALLOCATION

We assume in this work the use of multiple SAT solvers, one for each different time frame. Using multiple solvers, we observed that performance is highly related to:

- *Solver cleanup frequency*: cleaning up the solver means removal of incrementally loaded problem clauses and learned clauses
- *Clause loading strategy*: on-demand loading of transition relation clauses based on topological dependency

A. Specialized solvers

From the statistical results of reported in Table II it's easy to see that inductive generalization and clause propagation queries are by far the most expensive ones in terms of average SAT solving time. Inductive generalization queries, in addition of being expensive, are also the most frequent type of query posed.

The reason why inductive generalization calls are so expensive can be due to the fact that during the inductive generalization of a clause, at every iteration a slightly changing formula is queried for a satisfying assignment in increasingly larger subspaces. Given two subsequent queries in inductive generalization, in fact, it can be noticed that their formulas can differ only for one literal of the present state clause try and one literal of the next state cube $\neg try$. As the subspace becomes larger solving times for those queries increases. The average expensiveness of clause propagation calls can be intuitively motivated by noticing that they are posed one time for every clause of every time frame. The innermost time frames are the ones with the largest number of clauses, and thus require the largest number of propagation calls. Unfortunately, given the high number of clauses in those time frames, the CNF formulas upon which such calls act are highly constrained and usually harder to solve. So during propagation there are, in general, more hard queries than simple queries, making the average SAT solving time for those queries high.

In an attempt to reduce the burden of each time frame's SAT solver, we have experimented the use of specialized solvers for handling such queries. For every time frame, a second solver is instantiated and used to respond a particular type of query (Q_4 or Q_6). Table III summarize the results of such experiment.

VI. SOLVER LOADING

To minimize the size of the formula to be loaded into each solver, a common approach is to load, for every SAT call that queries the dynamic behavior of the system, only the necessary part of the transition relation.

It is easy to observe that every SAT call that uses the transition relation involves a constraint on the next state variables of the system in the form of a cube c'. Such queries ask if there is a state of the system satisfying some constraints on the present state, which can reach in one transition states represented by c'. Since c' is a cube, the desired state must have a successor p such that p correspond to c' for the value of every variable of c'. It's easy to see that the only present state variables that are relevant in determining if such a state exists, are those in the structural support of the next state variables of c'. It follows that the only portions of the transition relation required to answer such queries are the logic cones of the next state variables of c'.

Such loading strategy, known as *lazy loading of transition relation*, is commonly employed in various implementations of IC3, as the ones of PdTRAV and ABC. We observed in average 50% reduction in the size of the CNF formula for the transition relation using lazy loading of transition relation.

We noticed that, for these queries, the portions of the transition relation loaded can be further minimized employing a CNF encoding technique, called Plaisted-Greenbaum CNF encoding [13] (henceforth simply called PG encoding). The AIG representation of the transition relation together with the assumptions specified by the next state cube c' can be viewed as a *constrained boolean circuit* [14], i.e. a boolean circuit in which some of the outputs are constrained to assume

certain values. The Plaisted-Greenbaum encoding is a special encoding that can be applied in the translation of a constrained boolean circuit into a CNF formula.

Below we give an informal description of the PG encoding. For a complete description refer to [13] or [14].

Given an AIG representation of a circuit, a CNF encoding first subdivides that circuit into a set of functional blocks, i.e. gates or group of connected gates representing certain boolean functions, and introduces a new CNF variable a for each of these blocks. For each functional block representing a function f on the input variables $x_1, x_2, \ldots, x_n$, a set of clauses is derived translating into CNF the formula:

$$a \leftrightarrow f(x_1, x_2, \ldots, x_n) \tag{1}$$

The final CNF formula is obtained by the conjunction of these sets of clauses. Different CNF encodings differ in the way the gates are grouped together to form functional blocks and in the way these blocks are translated into clauses. The idea of PG encoding is to start from a base CNF encoding, and then use both output assumptions and topological information of the circuit to get rid of some of the derived clauses, while still producing an equi-satisfiable CNF formula. Based on the output constraints and the number of negations that can be found in every path from a gate to an output node, it can be shown that, for some gates of the circuit, an equi-satisfiable encoding can be produced by only translating one of the two sides of the bi-implication (1). The CNF formula produced by PG encoding will be a subset of the one produced by the traditional encoding.

PG encoding proves to be effective in reducing the size of loaded formulas, but it is not certain whether it is more efficient for SAT solving, since it may have worst propagation behaviour [15].

We observed a 10-20% reduction in the size of the CNF formula for the transition relation.

VII. Solvers clean up

A natural question arises regarding how frequently and at what conditions SAT solvers cleanups should be scheduled. Cleaning up a SAT solver, in fact, introduces a certain overhead. This is because:

- Relevant clauses for the current query must be reloaded.
- Relevant inferences previously derived must be derived again from those clauses.

A heuristic cleanup strategy is needed in order to achieve a trade-off between the overhead introduced and the slowdown in BCP avoided. The purpose of that heuristic is to determine when the number of irrelevant clauses (w.r.t. the current query) loaded in a solver becomes large enough to justify a cleanup. To do so, a heuristic measure representing an estimate of the number of currently loaded irrelevant clauses is compared to a certain threshold. If that measure exceeds the threshold, then the solver is cleaned up.

Clean up heuristics currently used in state-of-the-art tools, like ABC and PdTRAV, rely on loose estimates of the size of irrelevant portions of the formula loaded into each solver. These heuristics clean up each solver as soon as the computed estimate meets some, often static, threshold.

Our experiments with IC3 prove that the frequency of the cleanups play a crucial role in determining the overall verification performance. We explored the use of new cleanup heuristics based on more precise measures of the number of irrelevant clauses loaded and able to exploit correlation among different SAT calls to dynamically adjust the frequency of cleanups.

For SAT calls in IC3, notice that there are two sources of irrelevant clauses loaded in a solver:

1) Deactivated clauses loaded for previous inductive checks (Q_3 and Q_4 queries).
2) Portions of logic cones loaded for previous calls querying the dynamic behavour of the system.

Cleanup heuristics commonly used, such as the ones used in baseline versions of ABC and PdTRAV, typically take into account only the number of deactivated clauses in the solver to compute an estimate of the number of irrelevant clauses. We investigated the use of a new heuristic measure taking into account the second source of irrelevant clauses, i.e. irrelevant portions of previously loaded cones.

Every time a new query requires to load a new portion of the transition relation, to compute a measure of the number of the irrelevant transition relation's clauses, the following quantities are computed (assuming that c' is the cube constraining the next state variables for that query):

- A: the number of transition relation clauses already loaded into the solver (before loading the logic cones required by the current query);
- $S(c')$: the size (number of clauses) of the logic cones required for solving the current query;
- $L(c')$: the number of clauses in the required logic cones to be loaded into the solver (equal to the size of the required logic cone minus the number of clauses that such cone shares with previously loaded cones);

A measure of the number of irrelevant transition relation's clauses loaded for c', denoted by $u(c')$, can be computed as follows:

$$u(c') = A - (S(c') - L(c')) \tag{2}$$

Such a heuristic measure, divided by the number of clauses loaded into the solver, indicates the percentage of irrelevant clauses loaded in the solver w.r.t the current query. In Section VIII we investigate the use new cleanup strategies based on this measure. In order to take into account correlation between subsequent SAT calls, we consider such measure averaged on a time window of the last SAT calls.

VIII. Experimental Results

We have compared different cleanup and clause loading heuristics in both PdTRAV and ABC. In this section, we briefly summarize the obtained results.

A. PG encoding

A first set of experiments was done on the full set of 310 HWMCC'12 benchmarks [16], with a lower timeout, of 900 seconds, in order to evaluate the use of PG encoding. Results were controversial. A run in PdTRAV, with 900 seconds timeout, showed a reduction in the number of solved instances from 79 to 63 (3 of which previously unsolved). The percentage reduction of loaded transition relation clauses was 21.32%.

A similar run on ABC, showed a more stable comparison, from 80 to 79 solved instances (3 of which previously unsolved). So a first conclusion is that, PG encoding was not able to win over the standard encoding, suggesting it can indeed suffer of worst propagation behaviour. Nonetheless, it was very interesting to observe that the overall number of solved problems grew from 79 to 82 in PdTRAV and from 80 to 85 in ABC.

Different results between the two tools in this experimentation could be partially due to different light-weight preprocessing done by default within them. We started a more detailed comparison, in order to better understand the partially controversial results.

B. Experiments with PdTRAV

We then focused on a subset of 70 selected circuits, for which preprocessing was done off-line, and the tools were run on the same problem instances. In the following tables, the $P0$ rows always represent the default setting of PdTRAV. We report number of solver instances (out of 70) and average execution time (time limit 900 seconds). Column labeled New, when present, shows the number of instances not solved by $P0$.

Configuration	Solved [#]	New [#]	Avg Time [s]
P0 (baseline)	64		137.00
P1 (Q_4 spec.)	66	4	144.18
P2 (Q_6 spec.)	60	3	134.25

Table III: Tests on specialized solvers.

Table III shows two different implementations with specialized solvers (so two solvers per time frame): in the first one ($P1$) the second solver handles generalization calls while in the second one ($P2$) the it handles propagation calls. Overall, we see a little improvement by $P1$, with two more instances solved w.r.t $P0$, including 4 instances not solved by $P0$.

The next two tables show an evaluation of different solver cleanup heuristics. Let a be the number of activation variables in the solver, $|vars|$ the total number of variables in the solver, $|clauses|$ the total number of clauses in the solver, $u(c')$ the heuristic measure discussed in Section VII and $W(x, n)$ a sliding window containing the values of x for the last n SAT calls. The heuristics compared are the following:

- H1: $a > 300$;
- H2: $a > \frac{1}{2}|vars|$;
- H3: $avg\big(W\big(\frac{u(c')}{|clause|}, 1000\big)\big) > 0.5$
- H4: $H2 \mid\mid H3$;

Heuristic H1 is the cleanup heuristic used in the baseline versions of both PdTRAV and ABC. The static threshold of 300 activation literals was determined experimentally. Heuristic H2 cleans up each solver as soon as half of its variables are used for activation literals. It can be seen as a first simple attempt to adopt a dynamic cleanup threshold. Heuristic H3 is the first heuristic proposed to take into account the second source of irrelevant clauses described in Section VII, i.e. irrelevant portions of previously loaded cones. In H3 a solver is cleaned up as soon as the percentage of irrelevant transition relation's clauses loaded, averaged on a window of the last 1000 calls, reaches 50%. Heuristic H4 takes into account both sources of irrelevant clauses, combining H2 and H3.

Configuration	Solved [#]	New [#]	Avg Time [s]
P0 (H1)	64		137.00
P3 (H2)	60	1	122.19
P4 (H3)	44		116.28
P5 (H4)	62	3	125.94

Table IV: Tests on clean up heuristics.

Table IV shows a comparison among H1 (row $P0$), H2, H3, and H4, in rows $P3$, $P4$, and $P5$, respectively. No PG encoding, nor specialized solvers were used. Heuristic H1, that employs a static threshold, was able to solve the largest number of instances. Among dynamic threshold heuristics, both H2 and H3 take into account a single source of irrelevant loaded clauses, respectively the deactivated clauses in H2 and the unused portions of transition relation in H3. Data clearly indicates that H3 has worse performance. This suggests that deactivated clauses play a bigger role in degrading SAT solvers' performance than irrelevant transition relation's clauses do. Taking into account only the latter source of irrelevant clauses it's not sufficient. It can be noticed that heuristic H4, that takes into account both sources, outperforms both H2 and H3. This proves that considering irrelevant clauses arising from previoulsy loaded cones in addition to deactivated clauses can be beneficial. In addition Table IV shows that dynamic heuristics were able solve some instances that can't be solved by the static heuristic H1 and viceversa. In terms of overall number of solved instances, the static heuristic H1 outperformes our best dynamic heuristic H4. This can be due to the fact that the threshold parameter of H1 results from extensive experimentation while to determine the parameters of H4 (window size and percentage thresholds) a narrower experimentation has been performed.

We then focused on H4, and benchmarked it in different setups. Results are showed in table V.

Here PG encoding was used in configurations $P6$ and $P8$, single solver per time frame in $P6$, additional specialized solver for generalization in $P7$ and $P8$. We see that the specialized solver configuration appears to perform worse with PG encoding. Also, adding a specialized solver for generalization to the dynamic heuristic H4 doesn't seem to be as effective as it is when using the static heuristic H1.

Configuration	Solved [#]	New [#]	Avg Time [s]
P0 (baseline)	64		137.00
P6 (PG)	59	3	208.85
P7 (Q_4 spec)	58	1	111.59
P8 (PG+Q_4 spec)	50	1	178.56

Table V: Tests on mixed strategies for cleanup heuristic H4.

This can be due to the fact that irrelevant clauses arising from generalization calls are not taken into account to schedule the clean up of the main solver that, in turn, is cleaned up less frequently.

C. Experiments with ABC

The 70 selected circuits were also benchmarked with ABC, with same pre-processing used in PdTRAV: Table VI report in row $A0$ the default setting of ABC. Row $A1$ shows the variant with PG encoding, row $A2$ shows a run without dynamic TR clause loading. Row $A3$ finally shows a different period for solver cleanup (1000 variables instead of 300).

Configuration	Solved [#]	New [#]	Avg Time [s]
A0	64		138.66
A1	63	1	152.18
A2	63	2	158.75
A3	64		138.45

Table VI: Tests on ABC with different strategies.

Overall, results show little variance among different settings, which could suggest lesser sensitivity of ABC to different tunings. Nonetheless, a further experimentation with ABC on the full set of 310 benchmarks (with 300 seconds time limit), showed a 14% improvement in the number of solved problems (from 71 to 81), which indicate a potential improvement for a portfolio-based tool, able to concurrently exploit multiple settings.

IX. Conclusions

The paper shows a detailed analysis and characterization of SAT queries posed by IC3. We also discuss new ideas for solver allocation/loading/restarting. The experimental evaluation done on two different state-of-the-art academic tools, shows lights and shadows, as no breakthrough or clear winner emerges from the new ideas.

PG encoding showed to be less effective than expected. This is probably because the benefits introduced in terms of loaded formula size will be overwhelmed by the supposed worst propagation behaviour of that formula.

The use of specialized solvers seems to be effective when a static cleanup heuristic is used, less effective when combined with PG encoding or a dynamic heuristic.

Our experiments showed that, when a dynamic cleanup heuristic is used, IC3's performance can be improved by taking into account both deactivated clauses and irrelevant portions of previously loaded cones. Even if a parameter configuration for H4 that is able to outperform the currently used, well-rounded static heuristic H1 hasn't been found yet, we believe that a more extensive experimentation could lead to better results.

Nonetheless, the results are more interesting if we consider them from the standpoint of a portfolio-based tool, since the overall coverage (by the union of all setups) is definitely higher.

So we believe that more effort in implementation, experimentation, and detailed analysis of case sudies, needs to be done. We also deem that this work contributes to the discussion of new developments in the research related to IC3.

References

[1] A. R. Bradley, "SAT-based model checking without unrolling," in *VMCAI*, Austin, Texas, Jan. 2011, pp. 70–87.

[2] A. R. Bradley, "Understanding IC3," in *SAT*, ser. Lecture Notes in Computer Science, A. Cimatti and R. Sebastiani, Eds., vol. 7317. Springer, 2012, pp. 1–14.

[3] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking using SAT procedures instead of BDDs," in *Proc. 36th Design Automation Conf.* New Orleans, Louisiana: IEEE Computer Society, Jun. 1999, pp. 317–320.

[4] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT Solver," in *Proc. Formal Methods in Computer-Aided Design*, ser. LNCS, W. A. Hunt and S. D. Johnson, Eds., vol. 1954. Austin, Texas, USA: Springer, Nov. 2000, pp. 108–125.

[5] P. Bjesse and K. Claessen, "SAT–Based Verification without State Space Traversal," in *Proc. Formal Methods in Computer-Aided Design*, ser. LNCS, vol. 1954. Austin, TX, USA: Springer, 2000.

[6] K. L. McMillan, "Interpolation and SAT-Based Model Checking," in *Proc. Computer Aided Verification*, ser. LNCS, vol. 2725. Boulder, CO, USA: Springer, 2003, pp. 1–13.

[7] N. Eén, A. Mishchenko, and R. K. Brayton, "Efficient implementation of property directed reachability," in *FMCAD*, 2011, pp. 125–134.

[8] A. Mishchenko, "ABC: A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/∼alanmi/abc/," 2005.

[9] G. Cabodi, S. Nocco, and S. Quer, "Benchmarking a model checker for algorithmic improvements and tuning for performance," *Formal Methods in System Design*, vol. 39, no. 2, pp. 205–227, 2011.

[10] N. Eén and N. Sörensson, "The Minisat SAT Solver, http://minisat.se," Apr. 2009.

[11] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. 38th Design Automation Conf.* Las Vegas, Nevada: IEEE Computer Society, Jun. 2001.

[12] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 4, pp. 543–560, 2003.

[13] D. A. Plaisted and S. Greenbaum, "A structure-preserving clause form translation," *J. Symb. Comput.*, vol. 2, no. 3, pp. 293–304, 1986.

[14] M. Järvisalo, A. Biere, and M. Heule, "Simulating circuit-level simplifications on CNF," *J. Autom. Reasoning*, vol. 49, no. 4, pp. 583–619, 2012.

[15] N. Eén, "Practical SAT: a tutorial on applied satisfiability solving," *Slides of invited talk at FMCAD*, 2007.

[16] A. Biere and T. Jussila, "The Model Checking Competition Web Page, http://fmv.jku.at/hwmcc."

Lemmas on Demand for Lambdas

Mathias Preiner, Aina Niemetz, and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

***Abstract*—We generalize the lemmas on demand decision procedure for array logic as implemented in Boolector to handle non-recursive and non-extensional lambda terms. We focus on the implementation aspects of our new approach and discuss the involved algorithms and optimizations in more detail. Further, we show how arrays, array operations and SMT-LIB v2 macros are represented as lambda terms and lazily handled with lemmas on demand. We provide experimental results that demonstrate the effect of native lambda support within an SMT solver and give an outlook on future work.**

I. Introduction

The theory of arrays as axiomatized by McCarthy [14] enables us to reason about memory (components) in software and hardware verification, and is particularly important in the context of deciding satisfiability of first order formulas w.r.t. first order theories, also known as *Satisfiability Modulo Theories (SMT)*. However, it is restricted to array operations on single array indices and lacks support for efficiently modeling operations such as memory initialization and parallel updates (*memset* and *memcpy* in the standard C library).

In 2002, Seshia et al. [4] introduced an approach to overcome these limitations by using restricted λ-terms to model array expressions (such as *memset* and *memcpy*), ordered data structures and partially interpreted functions within the SMT solver UCLID [17]. The SMT solver UCLID employs an eager SMT solving approach and therefore eliminates all λ-terms through β-reduction, which replaces each argument variable with the corresponding argument term as a preliminary rewriting step. Other SMT solvers that employ a lazy SMT solving approach and natively support λ-terms such as CVC4 [1] or Yices [8] also treat them eagerly, similarly to UCLID, and eliminate all occurrences of λ-terms by substituting them with their instantiated function body (cf. C-style macros). Eagerly eliminating λ-terms via β-reduction, however, may result in an exponential blow-up in the size of the formula [17]. Recently, an extension of the theory of arrays was proposed [10], which uses λ-terms similarly to UCLID. This extension provides support for modeling *memset*, *memcpy* and loop summarizations. However, it does not make use of native support of λ-terms provided by an SMT solver. Instead, it reduces instances in the theory of arrays with λ-terms to a theory combination supported by solvers such as Boolector [3] (without native support for λ-terms), CVC4, STP [12], and Z3 [6].

In this paper, we generalize the decision procedure for the theory of arrays with bit vectors as introduced in [3] to lazily handle non-recursive and non-extensional λ-terms. We show how arrays, array operations and SMT-LIB v2 macros are represented in Boolector as λ-terms and introduce a lemmas on demand procedure for lazily handling λ-terms in Boolector in detail. We summarize an experimental evaluation and compare our results to solvers with SMT-LIB v2 macro support (CVC4, MathSAT [5], SONOLAR [13] and Z3) and finally, give an outlook on future work.

This work was funded by the Austrian Science Fund (FWF) under NFN Grant S11408-N23 (RiSE).

II. Preliminaries

We assume the usual notions and terminology of first order logic and are mainly interested in many-sorted languages, where bit vectors of different bit width correspond to different sorts and array sorts correspond to a mapping ($\tau_i \Rightarrow \tau_e$) from index sort τ_i to element sort τ_e. Our approach is focused primarily on the *quantifier-free* first order theories of *fixed size bit vectors*, *arrays* and *equality with uninterpreted functions*, but not restricted to the above.

We call 0-arity function symbols *constant* symbols and a, b, i, j, and e denote constants, where a and b are used for array constants, i and j for array indices, and e for an array value. For each bit vector of size n, the equality $=_n$ is interpreted as the identity relation over bit vectors of size n. We further interpret the *if-then-else* bit vector term ite_n as $ite(\top,t,e) =_n t$ and $ite(\bot,t,e) =_n e$. As a notational convention, the subscript might be omitted in the following. We identify *read* and *write* as basic operations on array elements, where $read(a,i)$ denotes the value of array a at index i, and $write(a,i,e)$

denotes the modified array a overwritten at position i with value e. The theory of arrays (without extensionality) is axiomatized by the following axioms, originally introduced by McCarthy in [14]:

$$i = j \rightarrow read(a,i) = read(a,j) \quad \text{(A1)}$$

$$i = j \rightarrow read(write(a,i,e),j) = e \quad \text{(A2)}$$

$$i \neq j \rightarrow read(write(a,i,e),j) = read(a,j) \quad \text{(A3)}$$

The *array congruence* axiom A1 asserts that accessing array a at two equal indices i and j produces the same element. The *read-over-write* Axioms A2 and A3 ensure a basic characteristic of arrays: A2 asserts that accessing a modification to an array a at the index it has most recently been updated (i), produces the value it has been updated with (e). A3 captures the case when a modification to an array a is accessed at an index other than the one it has most recently been updated at (j), which produces the unchanged value of the original array a at position j. Note that we assume that all variables a, i, j and e in axioms A1, A2 and A3 are universally quantified.

From the theory of equality with uninterpreted functions we primarily focus on the following axiom:

$$\forall \bar{x}, \bar{y}. \bigwedge_{i=1}^{n} x_i = y_i \rightarrow f(\bar{x}) = f(\bar{y}) \quad \text{(EUF)}$$

The *function congruence* axiom (EUF) asserts that a function evaluates to the same value for the same argument values.

We only consider a non-recursive λ-calculus, assuming the usual notation and terminology, including the notion of *function application*, *currying* and *β-reduction*. In general, we denote a λ-term λ_x as $\lambda x.t(x)$, where x is a variable *bound* by λ_x and $t(x)$ is a term in which x may or might not occur. We interpret $t(x)$ as defining the *scope* of bound variable x. Without loss of generality, the number of bound variables per λ-term is restricted to exactly one. Functions with more than one parameter are transformed into a chain of nested λ-terms by means of *currying* (e.g. $f(x,y) := x+y$ is rewritten as $\lambda x \,.\, \lambda y \,.\, x+y$). As a notational convention, we will use $\lambda_{\bar{x}}$ as a shorthand for $\lambda x_0 \ldots \lambda x_k \,.\, t(x_0,\ldots,x_k)$ for $k \geq 0$. We identify the *function application* as an explicit operation on λ-terms and interpret it as instantiating a bound variable (all bound variables) of a λ-term (a curried λ-chain). We interpret *β-reduction* as a form of function application, where all formal parameter variables (bound variables) are substituted with their actual parameter terms. We will use $\lambda_{\bar{x}}[x_0 \backslash a_0,\ldots,x_n \backslash a_n]$ to indicate β-reduction of a λ-term $\lambda_{\bar{x}}$, where the formal parameters $x_0,\ldots,x_n$ are substituted with the actual argument terms $a_0,\ldots,a_n$.

III. λ-TERMS IN BOOLECTOR

In contrast to λ-term handling in other SMT solvers such as e.g. UCLID or CVC4, where λ-terms are eagerly eliminated, in Boolector we provide a lazy λ-term handling with *lemmas on demand*. We generalized the lemmas on demand decision procedure for the extensional theory of arrays introduced in [3] to handle lemmas on demand for λ-terms as follows.

In order to provide a uniform handling of arrays and λ-terms within Boolector, we generalized all arrays (and array operations) to λ-terms (and operations on λ-terms) by representing *array variables* as uninterpreted functions (UF), *read* operations as function applications, and *write* and *if-then-else* operations on arrays as λ-terms. We further interpret macros (as provided by the command *define-fun* in the SMT-LIB v2 format) as (curried) λ-terms. Note that in contrast to [3], our implementation currently does not support extensionality (equality) over arrays (λ-terms).

We represent an *array* as exactly one λ-term with exactly one bound variable (parameter) and define its representation as $\lambda j \,.\, t(j)$. Given an array of sort $(\tau_i \Rightarrow \tau_e)$ and its λ-term representation $\lambda j \,.\, t(j)$, we require that bound variable j is of sort index τ_i and term $t(j)$ is of sort element τ_e. Term $t(j)$ is not required to contain j and if it does not contain j, it represents a *constant* λ-term (e.g. $\lambda j \,.\, 0$). In contrast to SMT-LIB v2 macros, it is not required to represent arrays with curried λ-chains, as arrays are accessed at one single index at a time (cf. *read* and *write* operations on arrays).

We treat *array variables* as UF with exactly one argument and represent them as f_a for array variable a.

We interpret *read* operations as function applications on either UF or λ-terms with read index i as argument and represent them as $read(a,i) \equiv f_a(i)$ and $read(\lambda j \,.\, t(j), i) \equiv (\lambda j \,.\, t(j))(i)$, respectively.

We interpret *write* operations as λ-terms modeling the result of the *write* operation on array a at index i with value e, and represent them as $write(a,i,e) \equiv \lambda j \,.\, ite(i=j, e, f_a(j))$. A function application on a λ-term λ_w representing a *write* operation yields value e if j is equal to the modified index i, and the unmodified value $f_a(j)$, otherwise. Note that applying β-reduction to a λ-term λ_w yields the same behaviour described by array axioms A2 and A3. Consider a function application on $\lambda_w(k)$, where k represents the position to be read from. If $k = i$ (A2), β-reduction yields the written value e, whereas if $k \neq i$ (A3), β-reduction returns the unmodified value of array a at position k represented by $f_a(k)$. Hence, these

axioms do not need to be explicitly checked during consistency checking. This is in essence the approach to handle arrays taken by UCLID [17].

We interpret *if-then-else* operations on arrays a and b as λ-terms, and represent them as $ite(c,a,b) \equiv \lambda j \,.\, ite(c, f_a(j), f_b(j))$. Condition c yields either function application $f_a(j)$ or $f_b(j)$, which represent the values of arrays a and b at index j, respectively.

In addition to the base array operations introduced above, λ-terms enable us to succinctly model array operations like e.g. *memcpy* and *memset* from the standard C library, which we previously were not able to efficiently express by means of *read*, *write* and *ite* operations on arrays. In particular, both *memcpy* and *memset* could only be represented by a fixed sequence of *read* and *write* operations within a constant index range, such as copying exactly 5 words etc. It was not possible to express a variable range, e.g. copying n words, where n is a symbolic (bit vector) variable.

With λ-terms however, we do not require a sequence of array operations as it usually suffices to model a parallel array operation by means of exactly one λ-term. Further, the index range does not have to be fixed and can therefore be within a variable range. This type of high level modeling turned out to be useful for applications in software model checking [10]. See also [17] for more examples. For instance, the *memset* with signature *memset* (a,i,n,e), which sets each element of array a within the range $[i, i+n[$ to value e, can be represented as $\lambda j \,.\, ite(i \leq j \wedge j < i+n, e, f_a(j))$. Note, n can be symbolic, and does not have to be a constant. In the same way, *memcpy* with signature *memcpy* (a,b,i,k,n), which copies all elements of array a within the range $[i, i+n[$ to array b, starting from index k, is represented as $\lambda j \,.\, ite(k \leq j \wedge j < k+n, f_a(i+j-k), f_b(j))$. As a special case of *memset*, we represent *array initialization* operations, where all elements of an array are initialized with some (constant or symbolic) value e, as $\lambda j \,.\, e$.

Introducing λ-terms does not only enable us to model arrays and array operations, but further provides support for arbitrary functions (macros) by means of currying, with the following restrictions: (1) functions may not be recursive and (2) arguments to functions may not be functions. The first restriction enables keeping the implementation of λ-term handling in Boolector as simple as possible, whereas the second restriction limits λ-term handling in Boolector to non-higher order functions. Relaxing these restrictions will turn the considered λ-calculus to be Turing-complete and in general render the decision problem to be undecidable. As future work it might be interesting to consider some relaxations.

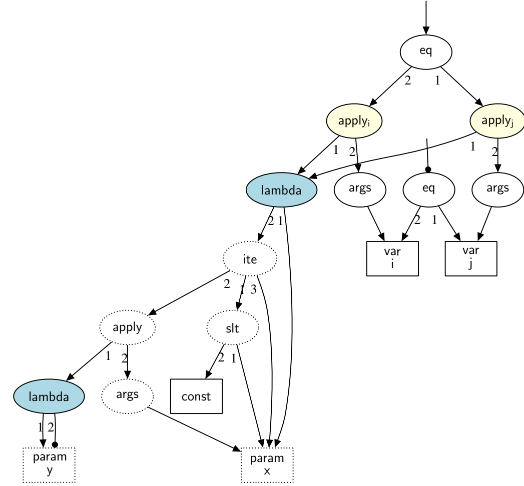


Fig. 1: DAG representation of formula ψ_1.

In contrast to treating SMT-LIB v2 macros as C-style macros, i.e., substituting every function application with the instantiated function body, in Boolector, we directly translate SMT-LIB v2 macros into λ-terms, which are then handled lazily via lemmas on demand. Formulas are represented as directed acyclic graphs (DAG) of bit vector and array expressions. Further, in this paper, we propose to treat arrays and array operations as λ-terms and operations on λ-terms, which results in an expression graph with no expressions of sort array ($\tau_i \Rightarrow \tau_e$). Instead, we introduce the following four additional expression types of sort bit vector:

- a *param* expression serves as a placeholder variable for a variable bound by a λ-term
- a *lambda* expression binds exactly one *param* expression, which may occur in a bit vector expression that represents the body of the λ-term
- an *args* expression is a list of function arguments
- an *apply* expression represents a function application that applies arguments *args* to a *lambda* expression

Example 1: Consider $\psi_1 \equiv f(i) = f(j) \wedge i \neq j$ with functions $f(x) := ite(x < 0, g(x), x)$, $g(y) := -y$ as depicted in Fig. 1. Both functions are represented as λ-terms, where function $g(y)$ returns the negation of y and is used in function $f(x)$, which computes the absolute value of x. Dotted nodes indicate parameterized expressions, i.e., expressions that depend on *param* expressions, and serve as templates that are instantiated as soon as β-reduction is applied.

In order to lazily evaluate λ-terms in Boolector we implemented two β-reduction approaches, which we will discuss in the next section in more detail.

IV. β-REDUCTION

In this section we discuss how concepts from the λ-calculus have been adapted and implemented in

our SMT solver Boolector. We focus on reduction algorithms for the non-recursive λ-calculus, which is rather atypical for the (vast) literature on λ-calculus. In the context of Boolector, we distinguish between *full* and *partial* β-reduction. They mainly differ in their application and the depth up to which λ-terms are expanded. In essence, given a function application $\lambda_{\bar{x}}(a_0,\ldots,a_n)$ *partial* β-reduction reduces only the top-most λ-term $\lambda_{\bar{x}}$, whereas *full* β-reduction reduces $\lambda_{\bar{x}}$ and every λ-term in the scope of $\lambda_{\bar{x}}$.

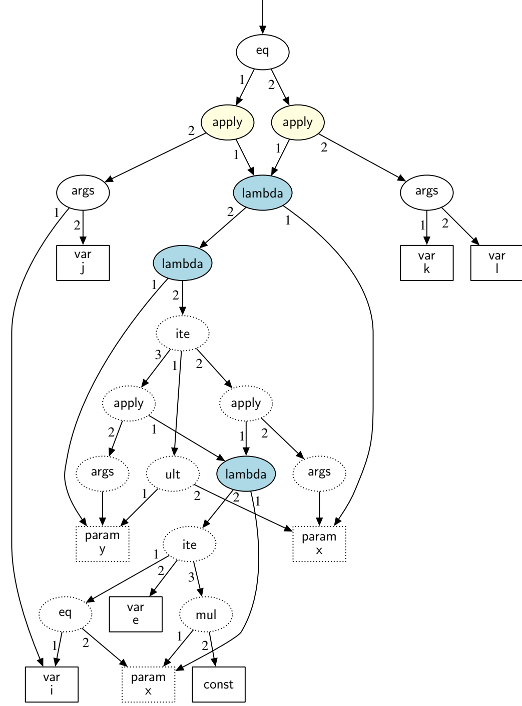
Full β-reduction of a function application on λ-term $\lambda_{\bar{x}}$ consists of a series of β-reductions, where λ-term $\lambda_{\bar{x}}$ and every λ-term $\lambda_{\bar{y}}$ within the scope of $\lambda_{\bar{x}}$ are instantiated, substituting all formal parameters with actual parameter terms. Since we do not allow partial function applications, full β-reduction guarantees to yield a term which is free of λ-terms. Given a formula with λ-terms, we usually employ full β-reduction in order to eliminate all λ-terms by substituting every function application with the term obtained by applying full β-reduction on that function application. In the worst case, full β-reduction results in an exponential blow-up. However, in practice, it is often beneficial to employ full β-reduction, since it usually leads to significant simplifications through rewriting. In Boolector, we incorporate this method as an optional rewriting step. We will use $\lambda_{\bar{x}}[x_0\backslash a_0,\ldots,x_n\backslash a_n]_{\mathbf{f}}$ as a shorthand for applying full β-reduction to $\lambda_{\bar{x}}$ with arguments $a_0,\ldots,a_n$.

Partial β-reduction of a λ-term $\lambda_{\bar{x}}$, on the other hand, essentially works in the same way as what is referred to as β-reduction in the λ-calculus. Given a function application $\lambda_{\bar{x}}(a_0,\ldots,a_n)$, partial β-reduction substitutes formal parameters $x_0,\ldots,x_n$ with the actual argument terms $a_0,\ldots,a_n$ without applying β-reduction to other λ-terms within the scope of $\lambda_{\bar{x}}$. This has the effect that λ-terms are expanded function-wise, which we require for consistency checking. In the following, we use $\lambda_{\bar{x}}[x_0\backslash a_0,\ldots,x_n\backslash a_n]_{\mathbf{p}}$ to denote the application of partial β-reduction to $\lambda_{\bar{x}}$ with arguments $a_0,\ldots,a_n$.

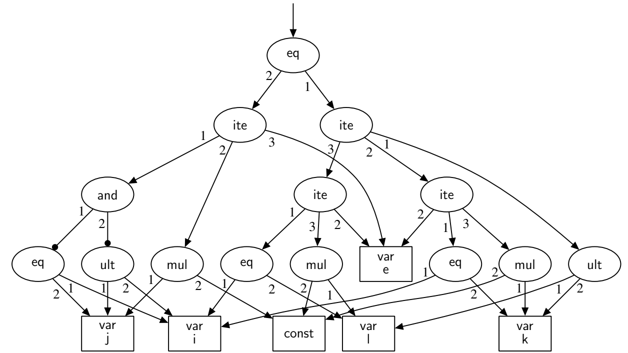
A. Full β-reduction

Given a function application $\lambda_{\bar{x}}(a_0,\ldots,a_n)$ and a DAG representation of $\lambda_{\bar{x}}$. Full β-reduction of $\lambda_{\bar{x}}$ consecutively substitutes formal parameters with actual argument terms while traversing and rebuilding the DAG in depth-first-search (DFS) post-order as follows.

1) Initially, we instantiate $\lambda_{\bar{x}}$ by assigning arguments $a_0,\ldots,a_n$ to the formal parameters $x_0,\ldots,x_n$.
2) While traversing down, for any λ-term $\lambda_{\bar{y}}$ in the scope of $\lambda_{\bar{x}}$, we need special handling for each function application $\lambda_{\bar{y}}(b_0,\ldots,b_m)$ as follows.



(a) Original formula ψ_2.



(b) Formula ψ'_2 after full β-reduction of ψ_2.

Fig. 2: Full β-reduction of formula ψ_2.

   a) Visit arguments $b_0,\ldots,b_m$ first, and obtain rebuilt arguments $b'_0,\ldots,b'_m$.
   b) Assign rebuilt arguments $b'_0,\ldots,b'_m$ to $\lambda_{\bar{y}}$ and apply β-reduction to $\lambda_{\bar{y}}(b'_0,\ldots,b'_m)$.

   This ensures a bottom-up construction of the β-reduced DAG (see step 3.), since all arguments $b'_0,\ldots,b'_m$ passed to a λ-term $\lambda_{\bar{y}}$ are β-reduced and rebuilt prior to applying β-reduction to $\lambda_{\bar{y}}$.
3) During up-traversal of the DAG we rebuild all visited expressions bottom-up and require special handling for the following expressions:
   - *param*: substitute *param* expression y_i with current instantiation b'_i
   - *apply*: substitute expression $\lambda_{\bar{y}}(b_0,\ldots,b_m)$ with $\lambda_{\bar{y}}[y_0\backslash b'_0,\ldots,y_m\backslash b'_m]_{\mathbf{f}}$

We further employ following optimizations to improve the performance of the full β-reduction algorithm.

- *Skip expressions that do not need rebuilding*
  Given an expression e within the scope of a λ-term $\lambda_{\bar{x}}$. If e is not parameterized and does not contain any λ-term, e is not dependent on arguments passed to $\lambda_{\bar{x}}$ and may therefore be skipped.
- *λ-scope caching*
  We cache rebuilt expressions in a λ-scope to prevent rebuilding parameterized expressions several times.

Example 2: Given a formula $\psi_2 \equiv f(i,j) = f(k,l)$ and two functions $g(x) := ite(x = i, e, 2 * x)$ and $f(x,y) := ite(y < x, g(x), g(y))$ as depicted in Fig. 2a. Applying full β-reduction to formula ψ_2 yields formula ψ_2' as illustrated in Fig. 2b. Function application $f(i,j)$ has been reduced to $ite(j \geq i \wedge i \neq j, 2 * j, e)$ and $f(k,l)$ to $ite(l < k, ite(k = i, e, 2 * k), ite(l = i, e, 2 * l))$.

B. Partial β-reduction

Given a function application $\lambda_{\bar{x}}(a_0, \ldots, a_n)$ and a DAG representation of $\lambda_{\bar{x}}$. The scope of a partial β-reduction $\beta_p(\lambda_{\bar{x}})$ is defined as the sub-DAG obtained by cutting off all λ-terms in the scope of $\lambda_{\bar{x}}$. Assume that we have an assignment for arguments $a_0, \ldots, a_n$, and for all non-parameterized expressions in the scope of $\beta_p(\lambda_{\bar{x}})$. The partial β-reduction algorithm substitutes *param* expressions $x_0, \ldots, x_n$ with $a_0, \ldots, a_n$ and rebuilds $\lambda_{\bar{x}}$. Similar to full β-reduction, we perform a DFS post-order traversal of the DAG as follows.

1) Initially, we instantiate $\lambda_{\bar{x}}$ by assigning arguments $a_0, \ldots, a_n$ to the formal parameters $x_0, \ldots, x_n$.
2) While traversing down the DAG, we require special handling for the following expressions:
   - function applications $\lambda_{\bar{y}}(b_0, \ldots, b_m)$
     a) Visit arguments $b_0, \ldots, b_m$, obtain rebuilt arguments $b_0', \ldots, b_m'$.
     b) Do not assign rebuilt arguments $b_0', \ldots, b_m'$ to $\lambda_{\bar{y}}$ and stop down-traversal at $\lambda_{\bar{y}}$.
   - $ite(c, t_1, t_2)$
     Since we have an assignment for all non-parameterized expressions within the scope of $\beta_p(\lambda_{\bar{x}})$, we are able to evaluate condition c. Based on that we either select t_1 or t_2 to further traverse down (the other branch is omitted).
3) During up-traversal of the DAG we rebuild all visited expressions bottom-up and require special handling for the following expressions:
   - *param*: substitute *param* expression y_i with current instantiation b_i'
   - *if-then-else*: substitute expression $ite(c, t_1, t_2)$ with t_1 if $c = \top$, and t_2 otherwise

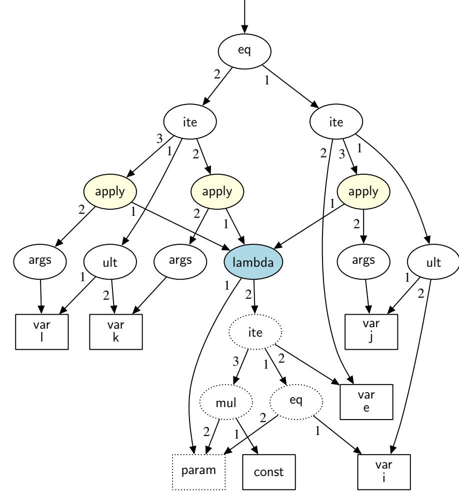


Fig. 3: Partial β-reduction of formula ψ_2.

For partial β-reduction, we have to modify the first of the two optimizations introduced for full β-reduction.

- *Skip expressions that do not need rebuilding*
  Given an expression e in the scope of partial β-reduction $\beta_p(\lambda_{\bar{x}})$. If e is not parameterized, in the context of partial β-reduction, e is not dependent on arguments passed to $\lambda_{\bar{x}}$ and may be skipped.

Example 3: Consider ψ_2 from Ex. 2. Applying partial β-reduction to ψ_2 yields the formula depicted in Fig. 3, where function application $f(i,j)$ has been reduced to $ite(j < i, e, g(j))$ and $f(k,l)$ to $ite(l < k, g(k), g(l))$.

V. Decision Procedure

The idea of *lemmas on demand* goes back to [7] and actually represents one extreme variant of the lazy SMT approach [16]. Around the same time, a related technique was developed in the context of bounded model checking [9], which lazily encodes all-different constraints over bit vectors (see also [2]). In constraint programming the related technique of lazy clause generation [15] is effective too.

In this section, we introduce lemmas on demand for non-recursive λ-terms based on the algorithm introduced in [3]. A top-level view of our lemmas on demand decision procedure for λ-terms (DP_λ) is illustrated in Fig. 4 and proceeds as follows. Given a formula ϕ, DP_λ uses a bit vector skeleton of the preprocessed formula π as formula abstraction $\alpha_\lambda(\pi)$. In each iteration, an underlying decision procedure DP_B determines the satisfiability of the formula abstraction refined by formula refinement ξ, i.e., in DP_B, we eagerly encode the refined formula abstraction Γ to SAT and determine

```
procedure DP_λ (φ)
    π ← preprocess(φ)
    ξ ← ⊤
    loop
        Γ ← α_λ(π) ∧ ξ
        (r,σ) ← DP_B(Γ)
        if r = unsatisfiable return unsatisfiable
        if consistent_λ(π,σ) return satisfiable
        ξ ← ξ ∧ α_λ(lemma_λ(π,σ))
```

Fig. 4: Lemmas on demand for λ-terms DP_λ.

its satisfiability by means of a SAT solver. As Γ is an over-approximation of ϕ, we immediately conclude with *unsatisfiable* if Γ is unsatisfiable. If Γ is satisfiable, we have to check if the current satisfying assignment σ (as provided by procedure DP_B) is consistent w.r.t. preprocessed formula π. If σ is consistent, i.e., if it can be extended to a valid satisfying assignment for the preprocessed formula π, we immediately conclude with *satisfiable*. Otherwise, assignment σ is spurious, $consistent_\lambda(\pi,\sigma)$ identifies a violation of the function congruence axiom EUF, and we generate a symbolic lemma $lemma_\lambda(\pi,\sigma)$ which is added to formula refinement ξ in its abstracted form $\alpha_\lambda(lemma_\lambda(\pi,\sigma))$.

Note that in ϕ, in contrast to the decision procedure introduced in [3], all array variables and array operations in the original input have been abstracted away and replaced by corresponding λ-terms and operations on λ-terms. Hence, various integral components of the original procedure (α_λ, $consistent_\lambda$, $lemma_\lambda$) have been adapted to handle λ-terms as follows.

VI. Formula Abstraction

In this section, we introduce a partial formula abstraction function α_λ as a generalization of the abstraction approach presented in [3]. Analogous to [3], we replace function applications by fresh bit vector variables and generate a bit vector skeleton as formula abstraction. Given π as the preprocessed input formula ϕ, our abstraction function α_λ traverses down the DAG structure starting from the roots, and generates an over-approximation of π as follows.

1) Each bit vector variable and symbolic constant is mapped to itself.
2) Each function application $\lambda_{\bar{x}}(a_0,\ldots,a_n)$ is mapped to a fresh bit vector variable.
3) Each bit vector term $t(y_0,\ldots,y_m)$ is mapped to $t(\alpha_\lambda(y_0),\ldots,\alpha_\lambda(y_m))$.

Note that by introducing fresh variables for function applications, we essentially cut off λ-terms and UF and therefore yield a pure bit vector skeleton, which is subsequently eagerly encoded to SAT.

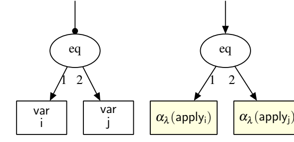


Fig. 5: Formula abstraction $\alpha_\lambda(\psi_1)$.

Example 4: Consider formula ψ_1 from Ex. 1, which has two roots. The abstraction function α_λ performs a consecutive down-traversal of the DAG from both roots. The resulting abstraction is a mapping of all bit vector terms encountered during traversal, according to the rules 1-3 above. For function applications (e.g. $\mathsf{apply_i}$) fresh bit vector variables (e.g. $\alpha_\lambda(\mathsf{apply_i})$) are introduced, where the remaining sub-DAGs are therefore cut off. The resulting abstraction $\alpha_\lambda(\psi_1)$ is given in Fig. 5.

VII. Consistency Checking

In this section, we introduce a consistency checking algorithm $consistent_\lambda$ as a generalization of the consistency checking approach presented in [3]. However, in contrast to [3], we do not propagate so-called access nodes but function applications and further check axiom EUF (while applying partial β-reduction to evaluate function applications under a current assignment) instead of checking array axioms A1 and A2. Given a satisfiable over-approximated and refined formula Γ, procedure $consistent_\lambda$ determines whether a current satisfying assignment σ (as provided by the underlying decision procedure DP_B) is spurious, or if it can be extended to a valid satisfying assignment for the preprocessed input formula π. Therefore, for each function application in π, we have to check both if the assignment of the corresponding abstraction variable is consistent with the value obtained by applying partial β-reduction, and if axiom EUF is violated. If $consistent_\lambda$ does not find any conflict, we immediately conclude that formula π is satisfiable. However, if current assignment σ is spurious w.r.t. preprocessed formula π, either axiom EUF is violated or partial β-reduction yields a conflicting value for some function application in π. In both cases, we generate a lemma as formula refinement. In the following we will equally use function symbols f, g, and h for UF symbols and λ-terms.

In order to check axiom EUF, for each λ-term and UF symbol we maintain a hash table ρ, which maps λ-terms and UF symbols to function applications. We check consistency w.r.t. π by applying the following rules.

I: For each $f(\bar{a})$, if $\bar{a}$ is not parameterized,
add $f(\bar{a})$ to $\rho(f)$

C: For any pair $s := g(\bar{a})$, $t := h(\bar{b}) \in \rho(f)$ check
$$\bigwedge_{i=0}^{n} \sigma(\alpha_\lambda(a_i)) = \sigma(\alpha_\lambda(b_i)) \rightarrow \sigma(\alpha_\lambda(s)) = \sigma(\alpha_\lambda(t))$$

B: For any $s := \lambda_{\bar{y}}(a_0, \dots, a_n) \in \rho(\lambda_{\bar{x}})$ with
$t := \lambda_{\bar{x}}[x_0 \backslash a_0, \dots, x_n \backslash a_n]_{\mathbf{p}}$,
check rule **P**, if **P** fails, check $eval(t) = \sigma(\alpha_\lambda(s))$

P: For any $s := \lambda_{\bar{y}}(a_0, \dots, a_n) \in \rho(\lambda_{\bar{x}})$ with
$t := g(b_0, \dots, b_m) = \lambda_{\bar{x}}[x_0 \backslash a_0, \dots, x_n \backslash a_n]_{\mathbf{p}}$,
if $n = m \wedge \bigwedge_{i=0}^{n} a_i = b_i$, propagate s to $\rho(g)$

Given a λ-term (UF symbol) f and a corresponding hash table $\rho(f)$. Rule **I**, the *initialization* rule, initializes $\rho(f)$ with all non-parameterized function applications on f. Rule **C** corresponds to the function congruence axiom and is applied whenever we add a function application $g(a_0, \dots, a_n)$ to $\rho(f)$. Rule **B** is a consistency check w.r.t. the current assignment σ, i.e., for every function application s in $\rho(f)$, we check if the assignment of $\sigma(\alpha_\lambda(s))$ corresponds to the assignment evaluated by the partially β-reduced term $\lambda_{\bar{x}}[x_0 \backslash a_0, \dots, x_n \backslash a_n]_{\mathbf{p}}$. Finally, rule **P** represents a crucial optimization of *consistent*$_\lambda$, as it avoids unnecessary conflicts while checking **B**. If **P** applies, both function applications s and t have the same arguments. As function application $s \in \rho(\lambda_{\bar{x}})$, rule **C** implies that $s = \lambda_{\bar{x}}(a_0, \dots, a_n)$. Therefore, function applications s and t must produce the same function value as $t := \lambda_{\bar{x}}[x_0 \backslash a_0, \dots, x_n \backslash a_n]_{\mathbf{p}} = \lambda_{\bar{y}}[x_0 \backslash a_0, \dots, x_n \backslash a_n]_{\mathbf{p}}$, i.e., function application t must be equal to the result of applying partial β-reduction to function application s. Assume we encode t and add it to the formula. If DP_B guesses an assignment s.t. $\sigma(\alpha_\lambda(t)) \neq \sigma(\alpha_\lambda(s))$ holds, we have a conflict and need to add a lemma. However, this conflict is unnecessary, as we know from the start that both function applications must map to the same function value in order to be consistent. We avoid this conflict by propagating s to $\rho(g)$.

Figure 6 illustrates our consistency checking algorithm *consistent*$_\lambda$, which takes the preprocessed input formula π and a current assignment σ as arguments, and proceeds as follows. First, we initialize stack S with all non-parameterized function applications in formula π (cf. `nonparam_apps`(π)) and order them top-down, according to their appearance in the DAG representation of π. The top-most function application then represents the top of stack S, which consists of tuples $(g, f(a_0, \dots, a_n))$, where f and g are initially equal and $f(a_0, \dots, a_n)$ denotes the function application propagated to function g. In the main consistency checking loop, we check rules **C** and **B** for each tuple as follows. First we check if $f(a_0, \dots, a_n)$ violates the function congruence axiom EUF w.r.t. function g and return $\bot$ if this is the case. Note that for checking rule **C**, we require an assignment for arguments $a_0, \dots, a_n$, hence we encode them on-the-fly. If rule **C** is not violated and function f is an uninterpreted function, we continue to check the next tuple on stack S. However, if f is a λ-term we still need to check rule **B**, i.e., we need to check if the assignment $\sigma(\alpha_\lambda(f(a_0, \dots, a_n)))$ is consistent with the value produced by $g[x_0 \backslash a_0, \dots, x_n \backslash a_n]_{\mathbf{p}}$. Therefore, we first encode all non-parameterized expressions in the scope of partial β-reduction $\beta_p(g)$ (cf. `encode(`g`)`) before applying partial β-reduction with arguments $a_0, \dots, a_n$, which yields term t. If term t has an assignment, we can immediately check if it differs from assignment $\sigma(\alpha_\lambda(f(a_0, \dots, a_n)))$ and return $\bot$ if this is the case. However, if term t does not have an assignment, which is the case when t has been instantiated from a parameterized expression, we have to compute the value for term t. Note that we could also encode term t to get an assignment $\sigma(t)$, but this might add a considerable amount of superfluous clauses to the SAT solver. Before computing a value for t we check if rule **P** applies and propagate $f(a_0, \dots, a_n)$ to h if applicable. Otherwise, we need to compute a value for t and check if t contains any function applications that were instantiated and not yet encoded (cf. `fresh_apps`(t)) and encode them if necessary. Finally, we compute

```
procedure consistent_λ(π, σ)
  S ← nonparam_apps(π)
  while S ≠ ∅
      (g, f(a_0,...,a_n)) ← pop(S)
      encode(f(a_0,...,a_n))
      /* check rule C */
      if not congruent(g, f(a_0,...,a_n))
          return ⊥
      add(f(a_0,...,a_n), ρ(g))
      if is_UF(g) continue
      encode(g)
      /* check rule B */
      t ← g[x_0\a_0,...,x_n\a_n]_p
      if assigned(t)
          if σ(t) ≠ σ(α_λ(f(a_0,...,a_n)))
              return ⊥
      elif t = h(a_0,...,a_n) /* check rule P */
          push(S, (h, f(a_0,...,a_n)))
          continue
      else
          apps ← fresh_apps(t)
          for a ∈ apps
              encode(a)
          if eval(t) ≠ σ(α_λ(f(a_0,...,a_n)))
              return ⊥
          for h(b_0,...,b_m) ∈ apps
              push(S, (h, h(b_0,...,b_m)))
  return ⊤
```

Fig. 6: Procedure *consistent*$_\lambda$ in pseudo-code.

the value for t (cf. eval(t)) and compare it to the assignment of $\alpha_\lambda(f(a_0,\dots,a_n))$. If the values differ, we found an inconsistency and return $\bot$. Otherwise, we continue consistency checking the newly encoded function applications *apps*. We conclude with $\top$, if all function applications have been checked successfully and no inconsistencies have been found.

A. Lemma generation

Following [3], we introduce a lemma generation procedure $lemma_\lambda$, which generates a symbolic lemma whenever our consistency checker detects an inconsistency. Depending on whether rule **C** or **B** was violated, we generate a symbolic lemma as follows. Assume that rule **C** was violated by function applications $s := g(a_0,\dots,a_n)$, $t := h(b_0,\dots,b_n) \in \rho(f)$. We first collect all conditions that lead to the conflict as follows.

1) Find the shortest possible propagation path p^s (p^t) from function application s (t) to function f.
2) Collect all *ite* conditions $c^s_0,\dots,c^s_j$ ($c^t_0,\dots,c^t_l$) on path p^s (p^t) that were $\top$ under given assignment σ.
3) Collect all *ite* conditions $c^s_0,\dots,c^s_k$ ($c^t_0,\dots,c^t_m$) on path p^s (p^t) that were $\bot$ under given assignment σ.

We generate the following (in general symbolic) lemma:

$$\bigwedge_{i=0}^{j} c^s_i \wedge \bigwedge_{i=0}^{k} \neg c^s_i \wedge \bigwedge_{i=0}^{l} c^t_i \wedge \bigwedge_{i=0}^{m} \neg c^t_i \wedge \bigwedge_{i=0}^{n} a_i = b_i \rightarrow s = t$$

Assume that rule **B** was violated by a function application $s := \lambda_{\bar{y}}(a_0,\dots,a_n) \in \rho(\lambda_{\bar{x}})$. We obtained $t := \lambda_{\bar{x}}[x_0\backslash a_0,\dots,x_n\backslash a_n]_{\mathbf{p}}$ and collect all conditions that lead to the conflict as follows.

1) Collect *ite* conditions $c^s_0,\dots,c^s_j$ and $c^s_0,\dots,c^s_k$ for s as in steps 1-3 above.
2) Collect all *ite* conditions $c^t_0,\dots,c^t_l$ that evaluated to $\top$ under current assignment σ when partially β-reducing $\lambda_{\bar{x}}$ to obtain t.
3) Collect all *ite* conditions $c^t_0,\dots,c^t_m$ that evaluated to $\bot$ under current assignment σ when partially β-reducing $\lambda_{\bar{x}}$ to obtain t.

We generate the following (in general symbolic) lemma:

$$\bigwedge_{i=0}^{j} c^s_i \wedge \bigwedge_{i=0}^{k} \neg c^s_i \wedge \bigwedge_{i=0}^{l} c^t_i \wedge \bigwedge_{i=0}^{m} \neg c^t_i \rightarrow s = t$$

Example 5: Consider formula ψ_1 and its preprocessed formula abstraction $\alpha_\lambda(\psi_1)$ from Ex. 1. For the sake of better readability, we will use λ_x and λ_y to denote functions f and g, and further use a_i and a_j as a shorthand for $\alpha_\lambda(\mathsf{apply_i})$ and $\alpha_\lambda(\mathsf{apply_j})$. Assume we run DP_B on $\alpha_\lambda(\psi_1)$ and it returns a satisfying assignment σ such that $\sigma(i) \neq \sigma(j)$, $\sigma(a_i) = \sigma(a_j)$, $\sigma(i) < 0$ and $\sigma(a_i) \neq \sigma(-i)$. First, we check consistency for $\lambda_x(i)$ and check rule **C**, which is not violated as $\sigma(i) \neq \sigma(j)$, and continue with checking rule **B**. We apply partial β-reduction and obtain term $t := \lambda_x[x/i]_{\mathbf{p}} = \lambda_y(i)$ (since $\sigma(i) < 0$) for which rule **P** is applicable. We propagate $\lambda_x(i)$ to λ_y, check if $\lambda_x(i)$ is consistent w.r.t. λ_y, apply partial β-reduction, obtain $t := \lambda_y[y/i]_{\mathbf{p}} = -i$ and find an inconsistency according to rule **B**: $\sigma(a_i) \neq \sigma(-i)$ but we obtained $\sigma(a_i) = \sigma(-i)$. We generate lemma $i < 0 \rightarrow a_i = -i$. Assume that in the next iteration DB_P returns a new satisfying assignment σ such that $\sigma(i) \neq \sigma(j)$, $\sigma(a_i) = \sigma(a_j)$, $\sigma(i) < 0$, $\sigma(a_i) = \sigma(-i)$ and $\sigma(j) > \sigma(-i)$. We first check consistency for $\lambda_x(i)$, which is consistent due to the lemma we previously generated. Next, we check rule **C** for $\lambda_x(j)$, which is not violated since $\sigma(i) \neq \sigma(j)$, and continue with checking rule **B**. We apply partial β-reduction and obtain term $t := \lambda_x[x/j]_{\mathbf{p}} = j$ (since $\sigma(j) > \sigma(-i)$ and $\sigma(i) < 0$) and find an inconsistency as $\sigma(a_i) = \sigma(-i)$, $\sigma(a_i) = \sigma(a_j)$ and $\sigma(j) > \sigma(-i)$, but $\sigma(a_j) = \sigma(j)$. We then generate lemma $j > 0 \rightarrow a_j = j$.

VIII. Experiments

We applied our lemmas on demand approach for λ-terms on three different benchmark categories: (1) *crafted*, (2) *SMT'12*, and (3) *application*. For the *crafted* category, we generated benchmarks using SMT-LIB v2 macros, where the instances of the first benchmark set (*macro blow-up*) tend to blow up in formula size if SMT-LIB v2 macros are treated as C-style macros. The benchmark sets *fisher-yates SAT* and *fisher-yates UNSAT* encode an incorrect and correct but naive implementation of the Fisher-Yates shuffle algorithm [11], where the instances of the *fisher-yates SAT* also tend to blow up in the size of the formula if SMT-LIB v2 macros are treated as C-style macros. The *SMT'12* category consists of all non-extensional QF_AUFBV benchmarks used in the SMT competition 2012. For the *application* category, we considered the *instantiation* benchmarks[1] generated with LLBMC as presented in [10]. The authors also kindly provided the same benchmark family using λ-terms as arrays, which is denoted as *lambda*.

We performed all experiments on 2.83GHz Intel Core 2 Quad machines with 8GB of memory running Ubuntu 12.04.2 setting a memory limit of 7GB and a time limit for the *crafted* and the *SMT'12* benchmarks of 1200 seconds. For the *application* benchmarks, as in [10]

[1] http://llbmc.org/files/downloads/vstte-2013.tgz

	Solver	Solved	TO	MO	Time [10^3s]	Space [GB]
macro blow-up	Boolector	100	0	0	24.2	9.4
	Boolector$_{nop}$	100	0	0	18.2	8.4
	Boolector$_{\beta}$	28	49	23	91.5	160.0
	CVC4	21	0	79	95.7	551.6
	MathSAT	51	2	47	64.6	395.0
	SONOLAR	26	74	0	90.2	1.7
	Z3	21	0	79	95.0	552.2
fisher-yates SAT	Boolector	7	10	1	14.0	7.5
	Boolector$_{nop}$	4	13	1	17.3	7.0
	Boolector$_{\beta}$	6	1	11	15.0	76.4
	CVC4	5	1	12	15.7	83.6
	MathSAT	6	10	2	14.7	17.3
	SONOLAR	3	14	1	18.1	6.9
	Z3	6	12	0	14.8	0.2
fisher-yates UNSAT	Boolector	5	13	1	17.4	7.1
	Boolector$_{nop}$	4	14	1	18.2	6.9
	Boolector$_{\beta}$	9	0	10	12.1	72.0
	CVC4	3	4	12	19.2	82.1
	MathSAT	6	11	2	15.9	14.7
	SONOLAR	3	15	1	19.2	6.8
	Z3	10	9	0	11.2	2.2

TABLE I: Results *crafted* benchmark.

we used a time limit of 60 seconds. We evaluated four different versions of Boolector: (1) our lemmas on demand for λ-terms approach DP_{λ} (Boolector), (2) DP_{λ} without optimization rule **P** (Boolector$_{nop}$), (3) DP_{λ} with full β-reduction (Boolector$_{\beta}$), and (4) the version submitted to the SMT competition 2012 (Boolector$_{sc12}$). For comparison we used the following SMT solvers: CVC4 1.2, MathSAT 5.2.6, SONOLAR 2013-05-15, STP 1673 (svn revision), and Z3 4.3.1. Note that we limited the set of solvers to those which currently support SMT-LIB v2 macros and the theory of fixed-size bit vectors. As a consequence, we did not compare our approach to UCLID (no bit vector support) and Yices, which both have native λ-term support, but lack support for the SMT-LIB v2 standard.

As indicated in Tables I, II and III, we measured the number of solved instances (Solved), timeouts (TO), memory outs (MO), total CPU time (Time), and total memory consumption (Space) required by each solver for solving an instance. If a solver ran into a timeout, 1200 seconds (60 seconds for category *application*) were added to the total time as a penalty. In case of a memory out, 1200 seconds (60 seconds for *application*) and 7GB were added to the total CPU time and total memory consumption, respectively.

Table I summarizes the results of the *crafted* benchmark category. On the *macro blow-up* benchmarks, Boolector and Boolector$_{nop}$ benefit from lazy λ-term handling and thus, outperform all those solvers which try to eagerly eliminate SMT-LIB v2 macros with a very high memory consumption as a result. The only solver not having memory problems on this benchmark set is SONOLAR. However, it is not clear how SONOLAR handles SMT-LIB v2 macros. Surprisingly, on these benchmarks Boolector$_{nop}$ performs better than Boolector with optimization rule **P**, which needs further investigation. On the *fisher-yates SAT* benchmarks Boolector not only solves the most instances, but requires 107 seconds for the first 6 instances, for which Boolector$_{\beta}$, MathSAT and Z3 need more than 300 seconds each. Boolector$_{nop}$ does not perform as well as Boolector due to the fact that on these benchmarks optimization rule **P** is heavily applied. In fact, on these benchmarks, rule **P** applies to approx. 90% of all propagated function applications on average. On the *fisher-yates UNSAT* benchmarks Z3 and Boolector$_{\beta}$ solve the most instances, whereas Boolector and Boolector$_{nop}$ do not perform so well. This is mostly due to the fact that these benchmarks can be simplified significantly when macros are eagerly eliminated, whereas partial β-reduction does not yield as much simplifications. We measured overhead of β-reduction in Boolector on these benchmarks and it turned out that for the *macro blow-up* and *fisher-yates UNSAT* instances the overhead is negligible (max. 3% of total run time), whereas for the *fisher-yates SAT* instances β-reduction requires over 50% of total run time.

	Solver	Solved	TO	MO	Time [10^3s]	Space [GB]
SMT'12	Boolector	139	10	0	19.9	14.8
	Boolector$_{nop}$	134	15	0	26.3	14.5
	Boolector$_{\beta}$	137	11	1	21.5	22.7
	Boolector$_{sc12}$	140	9	0	15.9	10.3

TABLE II: Results *SMT'12* benchmark.

Table II summarizes the results of running all four Boolector versions on the *SMT'12* benchmark set. We compared our three approaches Boolector, Boolector$_{nop}$, and Boolector$_{\beta}$ to Boolector$_{sc12}$, which won the QF_AUFBV track in the SMT competition 2012. In comparison to Boolector$_{\beta}$, Boolector solves 5 unique instances, whereas Boolector$_{\beta}$ solves 3 unique instances. In comparison to Boolector$_{sc12}$, both solvers combined solve 2 unique instances. Overall, on the *SMT'12* benchmarks Boolector$_{sc12}$ still outperforms the other approaches. However, our results still look promising since none of the approaches Boolector, Boolector$_{nop}$ and Boolector$_{\beta}$ are heavily optimized yet. On these benchmarks, the overhead of β-reduction in Boolector is around 7% of the total run time.

Finally, Table III summarizes the results of the *application* category. We used the benchmarks obtained from the instantiation-based reduction approach presented in [10] (*instantiation* benchmarks) and compared our

	Solver	Solved	TO	MO	Time [s]	Space [MB]
instantiation	Boolector	37	8	0	576	235
	Boolector_{nop}	35	10	0	673	196
	Boolector_{β}	44	1	0	138	961
	Boolector_{sc12}	39	6	0	535	308
	STP	44	1	0	141	3814
lambda	Boolector	37	8	0	594	236
	Boolector_{nop}	35	10	0	709	166
	Boolector_{β}	45	0	0	52	676
	Boolector_{sc12}	-	-	-	-	-
	STP	-	-	-	-	-

TABLE III: Results *application* benchmarks.

new approaches to STP, the same version of the solver that outperformed all other solvers on these benchmarks in the experimental evaluation of [10]. On the *instantiation* benchmarks Boolector_{β} and STP solve the same number of instances in roughly the same time. However, Boolector_{β} requires less memory for solving those instances. Boolector, Boolector_{nop} and Boolector_{sc12} did not perform so well on these benchmarks because in contrast to Boolector_{β} and STP, they do not eagerly eliminate read operations, which is beneficial on these benchmarks. The *lambda* benchmarks consist of the same problems as *instantiation*, using λ-terms for representing arrays. On these benchmarks, Boolector_{β} clearly outperforms Boolector and Boolector_{nop} and solves all 45 instances within a fraction of time. Boolector_{sc12} and STP do not support λ-terms as arrays and therefore were not able to participate on this benchmark set. By exploiting the native λ-term support for arrays in Boolector_{β}, in comparison to the *instantiation* benchmarks we achieve even better results. Note that on the *lambda* (*instantiation*) benchmarks, the overhead in Boolector_{β} for applying full β-reduction was around 15% (less than 2%) of the total run time.

Benchmarks, binaries of Boolector and all log files of our experiments can be found at: http://fmv.jku.at/difts-rev-13/lloddifts13.tar.gz.

IX. Conclusion

In this paper, we introduced a new decision procedure for handling non-recursive and non-extensional λ-terms as a generalization of the array decision procedure presented in [3]. We showed how arrays, array operations and SMT-LIB v2 macros are represented in Boolector and evaluated our new approach with 3 different benchmark categories: *crafted*, *SMT'12* and *application*. The *crafted* category showed the benefit of lazily handling SMT-LIB v2 macros where eager macro elimination tends to blow-up the formula in size. We further compared our new implementation to the version of Boolector that won the QF_AUFBV track in the SMT competition 2012. With the *application* benchmarks, we demonstrated the potential of native λ-term support within an SMT solver. Our experiments look promising even though we employ a rather naive implementation of β-reduction in Boolector and also do not incorporate any λ-term specific rewriting rules except full β-reduction.

In future work we will address the performance bottleneck of the β-reduction implementation and will further add λ-term specific rewriting rules. We will analyze the impact of various β-reduction strategies on our lemmas on demand procedure and will further add support for extensionality over λ-terms. Finally, with the recent and ongoing discussion within the SMT-LIB community to add support for recursive functions, we consider extending our approach to recursive λ-terms.

X. Acknowledgements

We would like to thank Stephan Falke, Florian Merz and Carsten Sinz for sharing benchmarks and Bruno Duterte for explaining the implementation and limits of lambdas in SMT solvers, and more specifically in Yices.

References

[1] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
[2] A. Biere and R. Brummayer. Consistency Checking of All Different Constraints over Bit-Vectors within a SAT Solver. In *FMCAD*, pages 1–4. IEEE, 2008.
[3] R. Brummayer and A. Biere. Lemmas on Demand for the Extensional Theory of Arrays. *JSAT*, 6(1-3):165–201, 2009.
[4] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *CAV*, volume 2404 of *LNCS*, pages 78–92. Springer, 2002.
[5] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.
[6] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *Proc. ETAPS'08*, pages 337–340, 2008.
[7] L. M. de Moura, H. Rueß, and M. Sorea. Lazy Theorem Proving for Bounded Model Checking over Infinite Domains. In *CADE*, volume 2392 of *LNCS*. Springer, 2002.
[8] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, Aug. 2006.
[9] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *ENTCS*, 89(4):543–560, 2003.
[10] S. Falke, F. Merz, and C. Sinz. Extending the Theory of Arrays: memset, memcpy, and Beyond. In *Proc. VSTTE'13*.
[11] R. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research*. Oliver and Boyd, 1953.
[12] V. Ganesh and D. L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *Proc. CAV'07*. Springer-Verlag, 2007.
[13] F. Lapschies, J. Peleska, E. Gorbachuk, and T. Mangels. SONOLAR SMT-Solver. System Desc. SMT-COMP'12. http://smtcomp.sourceforge.net/2012/reports/sonolar.pdf, 2012.
[14] J. McCarthy. Towards a Mathematical Science of Computation. In *IFIP Congress*, pages 21–28, 1962.
[15] O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
[16] R. Sebastiani. Lazy Satisability Modulo Theories. *JSAT*, 3(3-4):141–224, 2007.
[17] S. A. Seshia. *Adaptive Eager Boolean Encoding for Arithmetic Reasoning in Verification*. PhD thesis, CMU, 2005.

CHIMP: a Tool for Assertion-Based Dynamic Verification of SystemC Models

Sonali Dutta
Rice University
Email:Sonali.Dutta@rice.edu

Deian Tabakov
Schlumberger
Email:deian@dtabakov.com

Moshe Y. Vardi
Rice University
Email:vardi@cs.rice.edu

***Abstract*—CHIMP is a tool for assertion-based dynamic verification of SystemC models. The various features of CHIMP include automatic generation of monitors from temporal assertions, automatic instrumentation of the model-under-verification (MUV), and three-way communication among the MUV, the generated monitors, and the SystemC simulation kernel during the monitored execution of the instrumented MUV. Empirical results show that CHIMP puts minimal runtime overhead on the monitored execution of the MUV.**

A newly added path in CHIMP results in a significant (over 75%) reduction of average monitor generation and compilation time. The average size of the monitors is reduced by over 60%, without increasing runtime overhead.

I. Introduction

SystemC (IEEE Standard 1666-2005) has emerged as a de facto standard for modeling of hardware/software systems [4], supporting different levels of abstraction, iterative model refinement, and execution of the model during each design stage. SystemC is implemented as a C++ library with macros and base classes for modeling processes, modules, channels, signals, ports, and the like, while an event-driven simulation kernel allows efficient simulation of concurrent models. Thus, on one hand, SystemC leverages the natural object-oriented encapsulation, data hiding, and well-defined inheritance mechanism of C++, and, on the other hand, it allows modeling and efficient simulation of hardware/software designs by its simulation kernel and predefined hardware-specific macros and classes. The SystemC code is available as open source, including a single-core reference simulation kernel, referred to as the *OSCI kernel*; see http://www.accellera.org/downloads/standards/systemc.

Work supported in part by NSF grants CNS 1049862 and CCF-1139011, by NSF Expeditions in Computing project ”ExCAPE: Expeditions in Computer Augmented Program Engineering”, by BSF grant 9800096, and by gift from Intel.

The growing popularity of SystemC has motivated research aimed at assertion-based dynamic verification (ABDV) of SystemC models [9]. ABDV involves two steps: generating run-time monitors from input assertions [8], and executing the model-under-verification (MUV) while running the monitors along with the model. The monitors observe the execution of the MUV and report if the observed behavior is consistent with the specified behavior. For discussion of related work in ABDV of SystemC see [7].

CHIMP, available as an open-source tool[1], implements the monitoring framework for temporal SystemC properties described in [9]–see discussion below. CHIMP consists of (1) *off-the-self components*, (2) *modified components*, and (3) an *original component*. CHIMP has two *off-the-self components*: (1) `spot-1.1.1` [3], a C++ library used for LTL-to-Büchi conversion , and (2) `AspectC++-1.1` [6], a C++ aspect compiler that is used for instrumentation of the MUV. CHIMP has two *modified components*: (1) a patched version of the `OSCI kernel-2.2` to facilitate communication between the kernel and the monitors [9], and (2) an extension of `Automaton-1.11` [5], a Java tool used for determinization and minimization of finite automata, with the ability to read automata descriptions from file. Finally, the *original component* is `MONASGEN`, a C++ tool for automatic generation of monitors from assertions [8] and for automatic generation of an aspect advice file for instrumentation [11]. Fig. 1 shows the five components of CHIMP as described above. The component Automaton-1.11 is dotted because a newly added improved path in CHIMP has been able to remove the dependency of CHIMP on Automaton-1.11 (explained in Section V).

CHIMP takes the MUV and a set of temporal assertions about the behavior of that MUV as inputs,

[1] www.sourceforge.net/projects/chimp-rice

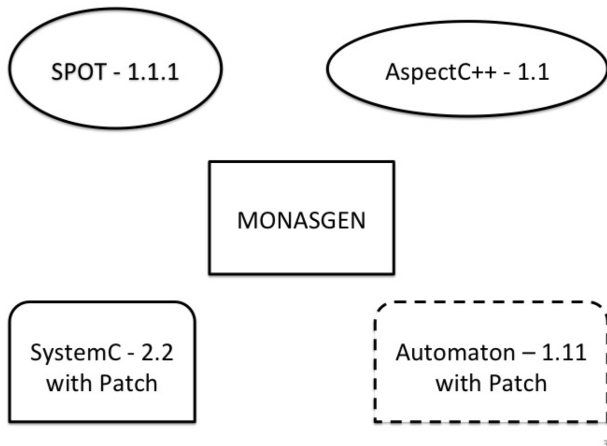


Fig. 1. CHIMP components

and outputs "FAILED" or "NOT FAILED", for each assertion. CHIMP performs white-box validation of user code and black-box validation of library code. (If a user wishes to do white-box validation of library code, it can be accomplished by treating the library code as part of the user code.) The two main features of CHIMP are: (1) CHIMP generates C++ monitors, one for each assertion to be verified, and (2) CHIMP automatically instruments the user model [11] to expose the user model's states and syntax to the monitors. CHIMP puts nominal overhead on the runtime of the MUV, supports a rich set of assertions, and can handle a wide array of abstractions, from statement level to system level.

A recently added path in CHIMP from LTL to monitor can bypass the old performance bottleneck, Automaton-1.1, and improves the monitor generation and compilation time by a significant amount. It also reduces the size of the generated monitors notably. This entirely removes the dependency of CHIMP on Automaton-1.1.

The theoretical and algorithmic foundations of CHIMP were described in [8]–[11]. In this paper we describe the architecture, usage, and recent evolution of CHIMP. The rest of the paper is organized as follows. Section II describes the syntax and semantics of assertions. Section III presents an overall picture about the usage, implementation and performance of CHIMP. Section IV describes the C++ monitor generated by CHIMP. Section V describes the new improved path in CHIMP and its improved performance. Finally, Section VI presents a summary and talks about future work.

II. ASSERTIONS: SYNTAX AND SEMANTICS

CHIMP accepts input assertions, defined using the temporal specification framework for SystemC described in [10], where a temporal assertion consists of a linear temporal formula accompanied by a Boolean expression serving as a clock. This supports assertions written at different levels of abstraction with different temporal resolutions. The framework of [10] proposes a set of SystemC-specific Boolean variables that refer to SystemC's software features and its simulation semantics; see examples below. Input assertions in CHIMP are of the form "$\langle LTL\ formula \rangle @ \langle clock\ expression \rangle$", where the $LTL\ formula$ expresses a temporal property and the $clock\ expression$ denotes when CHIMP should sample the execution trace of the MUV.

Several of the additional Boolean variables proposed in [10] refer to the simulation phases. According to SystemC's formal semantics, there are 18 predefined kernel phases. CHIMP has Boolean variables that enable referring to these phases. For example MON_DELTA_CYCLE_END denotes the end of each delta cycle and MON_THREAD_SUSPEND denotes the suspension moment of each thread. By using these variables as clock expressions, we can sample the execution trace at different temporal resolutions. (By default, CHIMP samples at each kernel phase.) Other Boolean variables refer to SystemC events, which are key elements of SystemC's event-driven simulation semantics. CHIMP is also able to sample the execution at various key locations, e.g., function calls and function returns. This gives the user the flexibility to write assertions at different levels of abstraction, from the level of individual C++ statements to the level of SystemC kernel phases.

The Boolean primitives supported by CHIMP are summarized below; see, [10] and [11] for further details:

Function primitives: Let `f()` be a C++ function in the user or library code. The Boolean primitives **f:call** and **f:return** refer to locations in the source code that contain the function call, and to locations immediately after the function call, respectively. The primitives **f:entry** and **f:exit** refer to the locations immediately before the first executable statement and immediately after the last executable statement in `f()`, respectively. If `f()` is a library function then the entry and exit primitives are not supported (black-box verification model for libraries).

Value primitives: If a function `f()` has k arguments, CHIMP defines variables $f:1,\ldots,f:k$, where the value and type of $f:i$ are equal to the value and type

of the i_{th} parameter of function `f()` before executing the first statement in the definition of `f()`. CHIMP also defines the variable f : 0, whose value and type are equal to the value and type of the object that `f()` returns. For example, if the function `int divide(int dividend, int divisor)` is defined in the MUV, then the formula *G (division:entry -> "division:2 != 0")* asserts that the divisor is nonzero whenever division function starts execution.

Phase primitives: The user can refer to the 18 predefined kernel states [10] in the assertions to specify when the state of the MUV should be sampled. For example, the assertion *G ("p == 0") @ MON_DELTA_CYCLE_END* requires the value of variable p to be zero at the end of every delta cycle.

Event primitives: For each SystemC event E, CHIMP provides a Boolean primitive `E.notified` that is true only when the OSCI kernel actually notifies E. For example, the assertion *G (s.value_changed_event().notified) @MON_UPDATE_PHASE_END* says that the signal `s` changes value at the end of every update phase.

III. Usage, Implementation and Performance

Running CHIMP consists of three steps: (1) In the first step, the user writes a configuration file containing all assertions to be verified, as well as other necessary information. The user can also provide inputs through command-line switches. For each LTL assertion in the configuration file, MONASGEN first generates a non-deterministic Büchi automaton on words (NBW) using SPOT, then converts that NBW to a minimal deterministic finite automaton on words (DFW), using the Automaton-1.11 tool for determinization and minimization. Then, MONASGEN generates the C++ monitors from the DFW, one monitor for each assertion (Fig. 2). (2) MONASGEN produces an aspect-advice file that is then used by AspectC++ to generate the instrumented MUV (Fig. 3). (3) Finally, the monitors and the instrumented MUV are compiled together and linked to the patched OSCI kernel, and the code is then run to execute the simulation with inputs provided by user. The inputs can be generated using any standard stimuli-generation technique. For every assertion, CHIMP produces output indicating if the assertion held or failed for that particular input (Fig. 4).

For experimental evaluation we used a SystemC model with about 3,000 LOC, implementing a system for reserving and purchasing airplane tickets. The users

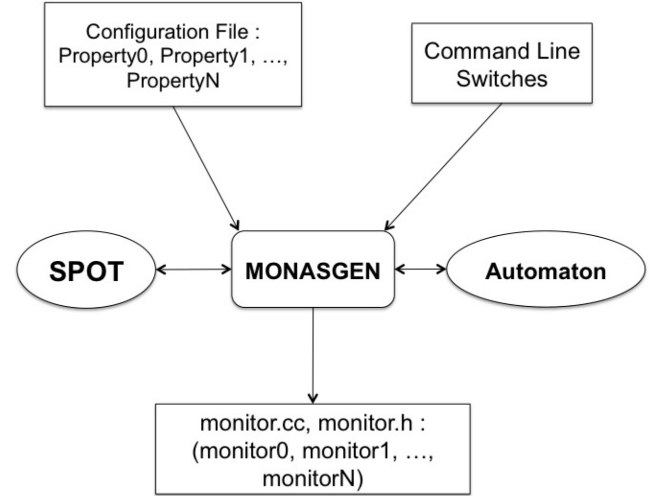


Fig. 2. Monitor generation flow

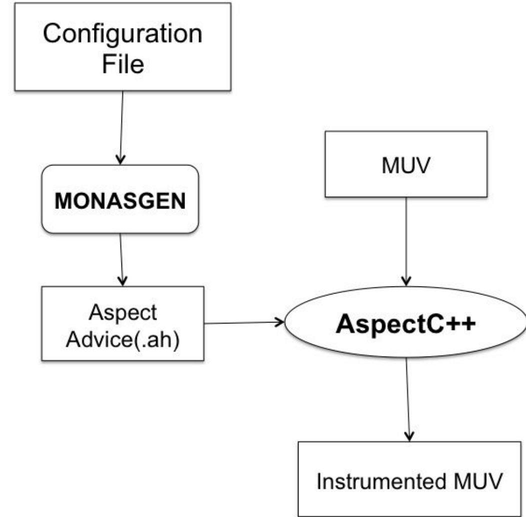


Fig. 3. MUV instrumentation flow

of the system submit requests and the system uses a randomly generated flight database to find a direct flight or a sequence of up to three connecting flights. Those are returned to the user for approval, payment and booking. This model is intended to run forever. It is inspired by actual request/grant subsystems currently used in hardware design.

We used a patched version of the 2.2.0 OSCI kernel and compiled it using the default settings in the Makefile. The empirical results below were measured on a Pentium 4, 3.20GHz CPU, 1 GB RAM machine running GNU

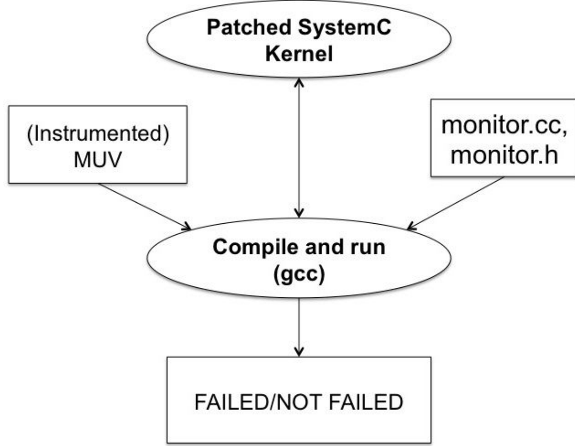


Fig. 4. Running instrumented MUV with monitors using patched OSCI kernel

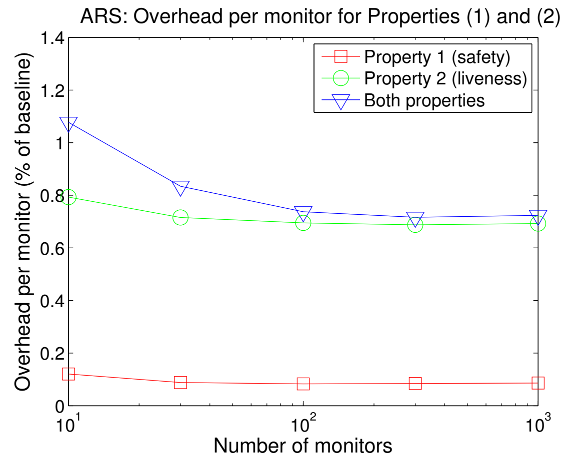


Fig. 5. Run time overhead for monitoring Properties (1) and (2)

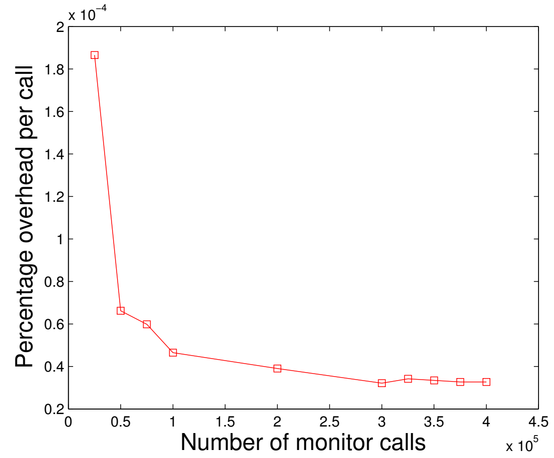


Fig. 6. Instrumentation overhead per monitor call as a percentage of the baseline run time. Y axis is ((instrumentation overhead − baseline)/(baseline × number of monitors)) × 100%

Linux. To assess runtime overhead imposed by the monitors, we measured the effect of running with different assertions and also increasing number of assertions [9]. In each case we first ran the model without monitors and user-code instrumentation to establish the baseline, and then ran several simulations with instrumentation and an increasing number of copies of the same monitor. The results report runtime overhead per monitor as a percentage of the baseline. (For these experiments monitors were constructed manually.)

The first property we checked is a safety property asserting that whenever the event *new_requests_nonfull* is notified, the corresponding queue *new_planning_requests* must have space for at least one request.

```
 G "new_planning_requests.size() < capacity"
@ new_requests_nonfull.notified
```
(1)

The second property says that the system must propagate each request through each channel (or through each module) within 5 cycles of the slow clock. This property is a conjunction of 16 bounded liveness assertions similar to the one shown here.

```
...
// Propagate through module within 5 clock ticks
ALWAYS (io_module_receive_transaction($1) ->
 ( within [5 slow_clock.pos()]
   io_module_send_to_master($2) & ($1 == $2)
 ) AND ...
```
(2)

Fig. 5 presents the runtime overhead of monitoring the above two properties as a percentage of the baseline. Checking Property 2 is relatively expensive because it requires a lot of communication from the model to the monitor. It is shown in [1] that the finite state monitors have less runtime overhead than transaction-based monitors generated by tools like Synopsys VCS.

We also evaluated the cost of instrumentation separately by simulating one million SystemC clock cycles with focus on the overhead of instrumentation [11]. The average wall-clock execution time of the system over 10 runs without instrumentation was 33 seconds. We call this "baseline execution". Fig. 6 shows the cost of the instrumentation per monitor call, as a percentage of the

baseline execution. The data suggest that there is a fixed cost of the instrumentation, which, when amortized over more and more calls, leads to lower average cost. The average cost per call stabilizes after 300,000 calls, and is less than $0.5 \times 10^{-4}\%$.

Another testbench we used is an Adder model that implements the squaring function by repeated increment by 1. It uses all three kinds of event notifications. It is scalable as it spawns a SystemC thread for each addition in the squaring function. We monitored several properties of the Adder model using CHIMP.

To study the effect of monitor encoding on runtime overhead, Tabakov and Vardi [8] describes 33 different monitor encodings, and shows that front_det_switch encoding with state minimization and no alphabet minimization is the best in terms of runtime overhead. The downside is that this flow suffers from a rather slow monitor generation and compilation time. This led us to develop a new flow for the tool, as described below.

IV. CHIMP MONITORS

In CHIMP, the $LTL\ \ formula$ in every assertion $\langle LTL\ \ formula \rangle$ @ $\langle clock\ \ expression \rangle$ is converted into a C++ monitor class. Each C++ monitor has a **step()** function that implements the transition function of the DFW generated from the $LTL\ \ formula$ (as described in Fig. 9). This step() function is called at every sampling point defined by $clock\ \ expression$. The monitor.h and monitor.cc files, generated by MONASGEN, contains one monitor class for every assertion and a class called `local_observer` that is responsible for invoking the callback function, which invokes *step()* function of the appropriate monitor class at the right sampling point during the monitored simulation. Different encodings have been explored for writing the monitor's *step()* function. For the new flow of CHIMP (described below), Fig. 13 shows that front_det_ifelse encoding is the best among all of them in terms of runtime overhead.

In front_det_ifelse encoding, the C++ monitor produced is deterministic in terms of transitions. This means from one state, either one or no transition is possible. If no transition is possible from a state, the monitor rejects and outputs "FAILED". Else, the monitor keeps executing until the end of simulation and outputs "NOT FAILED". In front_det_ifelse encoding, each state of the monitor is encoded as an integer, from 0 upto the total number of states. The *step()* function of the monitor uses an outer if-elseif statement block to determine the next state. The possible transitions from each state are encoded using an inner if-elseif block, the condition statement being the guard on a transition.

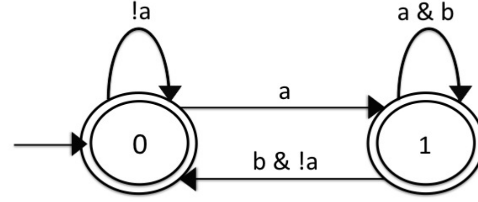


Fig. 7. The DFW generated by MONASGEN from the LTL formula $G(a \rightarrow Xb)$. All the states are accepting. The DFW rejects bad prefixes by no available transition. State 0 is the initial state.

As a running example, we show how the monitor *step()* function is encoded for an assertion $G(a \rightarrow Xb)$ @ clk. clk here is some boolean expression specifying when the *step()* function needs to be called during the simulation. This assertion asserts that, always, if a is true, then in the next clock cycle b has to be true. The DFW generated by CHIMP for this assertion is shown in Fig. 7.

Listing 1 shows the *step()* function of the C++ monitor generated by CHIMP for the assertion $G(a \rightarrow Xb)$ @ clk. The variables `current_state` and `next_state` are member variables of the monitor class and store the current automaton state and next automaton state respectively. If next state becomes -1 after the execution of the *step()* function, it means that no transition can be made from the current state. In this case, the monitor outputs "FAILED". The initial state 0 of the monitor is assigned to the variable `current_state` inside the constructor of the monitor class as shown in Listing 2.

Listing 1. The *step()* function of the monitor for $G(a \rightarrow Xb)$

```
/**
* Simulate a step of the monitor.
*/
void
monitor0::step() {
  //If the property has not failed yet
  if (status == NOT_FAILED) {
    //number of steps executed so far
    num_steps++;
    //assign next state to current state
    current_state = next_state;
    //make next state invalid
    next_state = -1;
```

```
    if (current_state == 0) {
      if(!(a)  )
        { next_state = 0; }
      else if((a)  )
        { next_state = 1; }
    } // if (current_state == 0)

    else if (current_state == 1) {
      if((b) && !(a)  )
        { next_state = 0; }
      else if((a) && (b)  )
        { next_state = 1; }
    } // if (current_state == 1)

  //FAILED if no transition possible
   bool not_stuck = (next_state != -1);
   if (! not_stuck) {
     property_failed();
   }
  } // if (status == NOT_FAILED)
} // step()
```

Listing 2. The constructor of the monitor for $G(a \rightarrow Xb)$

```
/**
 * The constructor
 */
monitor0::monitor0(...):
sc_core::mon_prototype() {
  next_state = 0; // initial state id
  current_state = -1;
  status = NOT_FAILED;
  num_steps = 0;
  . . .
} // Constructor
```

The sampling points can be kernel phases, e.g., MON_DELTA _CYCLE_END, or event notification, e.g., E.notified (E is an event). In such cases, the OSCI kernel needs to communicate with the `local_observer` at the right time (when a delta cycle ends or when event E is notified) to call the *step()* function of the monitor. This communication is done by the patch, put on OSCI kernel. This patch contains a class called `mon_observer`, which communicates with the `local_observer` class on behalf of the OSCI kernel.

V. Evolution of CHIMP

Fig. 8 shows the old path of generating C++ monitor from an LTL property in CHIMP. First, MONASGEN

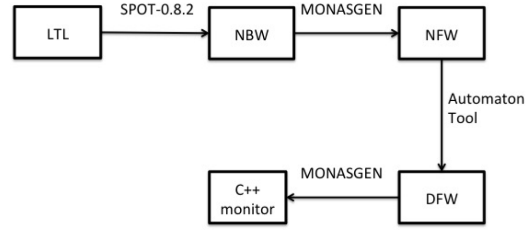


Fig. 8. Old LTL to monitor path in CHIMP

Fig. 9. New LTL to monitor path in CHIMP

uses SPOT to convert the LTL formula to a nondeterministic Büchi Automaton (NBW). Then MONASGEN prunes the NBW to remove all states that do not lead to any accepting state, and generate the corresponding nondeterministic finite automaton (NFW), see [2]. MONASGEN then uses the Automaton Tool to determinize and minimize the NFW to generate minimal deterministic finite automaton (DFW). Finally MONASGEN converts the DFW to C++ monitor.

The main bottleneck in this flow was minimization and determination of NFW using the Automaton tool as it consumes 90% of total monitor generation and compilation time. Also, the Automaton tool may generate multiple edges between two states, resulting in quite large monitors. The newest version of CHIMP introduces a new path as shown in Fig. 9, which bypasses Automaton tool completely and uses only SPOT. So the component Automaton-1.11 in Fig. 1 is not needed by CHIMP anymore. This new path leverages the new functionality of SPOT-1.1.1 to convert an LTL formula to a minimal DFW that explicitly rejects all bad prefixes.

After replacing the Automaton Tool by SPOT to convert the NBW to minimal DFW, the improvement in compilation time and monitor size is evident. This new flow results in 75.93% improvement in monitor generation and compilation time. To evaluate the performance of the revised CHIMP, we use the same set of 162 pattern formulas and 1200 random formulas as mentioned in [8]. The scatter plot on Fig. 10 shows the comparison of monitor generation and compilation time of the new flow vs the old flow. Most of the points are above the line with slope 1, which indicates that for most of the monitors, the

generation and compilation time by old CHIMP is more than by new CHIMP. Fig. 10 - Fig. 14 are all scatter plots[2] and interpreted in the same way.

The new flow also merges multiple edges between two states into one edge guarded by the disjunction of the guards on all edges between the states. In this way the average reduction in monitor size in bytes is 61.27%. Fig. 11 shows the comparison of the size in bytes of the monitors generated by the new flow vs the old flow.

Since the main focus of CHIMP has always been minimizing runtime overhead, we need to ensure that this new flow does not incur more runtime overhead compared to the old flow as a cost of reduced monitor generation and compilation time. So we ran the same set of 162 pattern formulas and 1200 random formulas as mentioned above, to compare the runtime overhead incurred by the new flow vs the old flow. Fig. 12 shows that the runtime overhead of CHIMP using the new flow has been reduced compared to the old flow. The reduction is 7.97% on average.

As CHIMP has evolved to follow a different and more efficient path and the monitors have been reduced in size, it was not clear a priori which monitor encoding is the best. We conducted the same experiment as in [8] with the same set of formulas, 162 pattern formulas and 1200 random formulas, to identify the best encodings in terms of both runtime overhead and monitor generation and compilation time. We identified two new best encodings. Fig. 13 shows that the new best encoding in terms of runtime overhead is now front_det_ifelse. Fig. 14 shows that the new best encoding in terms of monitor generation and compilation time is back_ass_alpha. CHIMP now provides two configurations for monitor generation, best_runtime, which has minimum runtime overhead and best_compiletime, which has minimum monitor generation and compilation time. Since the bigger concern is usually runtime overhead, best_runtime is the default configuration, given that its monitor generation and compilation time is very close to that of best_compiletime.

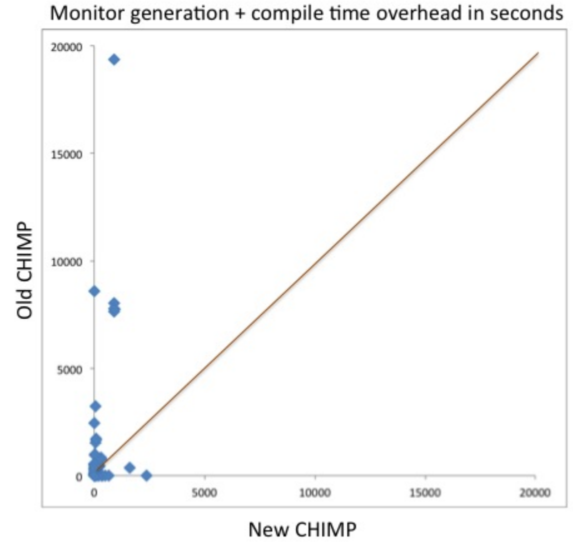


Fig. 10. Comparison of monitor generation and compilation time of the old CHIMP vs new CHIMP

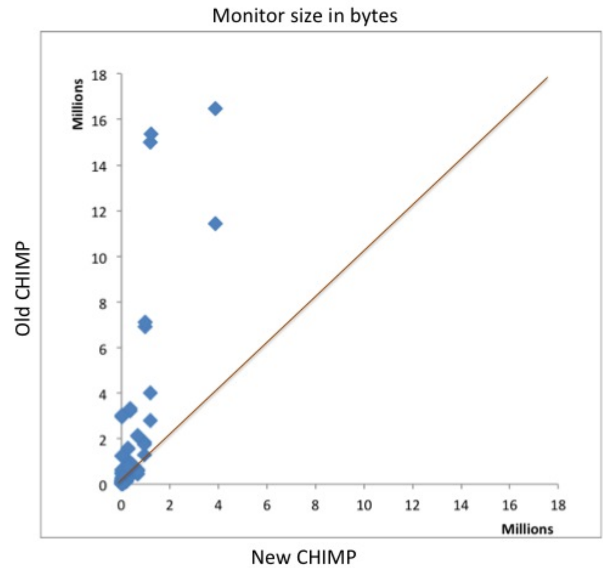


Fig. 11. Comparison of monitor size (bytes) generated by the old CHIMP vs new CHIMP

VI. Conclusion

We present CHIMP, an Assertion-Based Dynamic Verification tool for SystemC models, and show that it puts minimal overhead on the runtime of the MUV. We show how the new path in CHIMP results in significant reduction of monitor generation and compilation time and monitor size, as well as runtime overhead. In the future we plan to look at the possibility of verifying parametric properties, for example $G(send(id) \rightarrow F(receive(id)))$ where one can pass a parameter (here id), to the variables in the LTL formula of the assertion. The above property means that always if the message with ID id is sent, it should be received eventually. Also at present the user needs to know the system architecture to declare an assertion. We would like to make CHIMP work with assertions that are declared in the elaboration phase.

[2] http://en.wikipedia.org/wiki/Scatter_plot

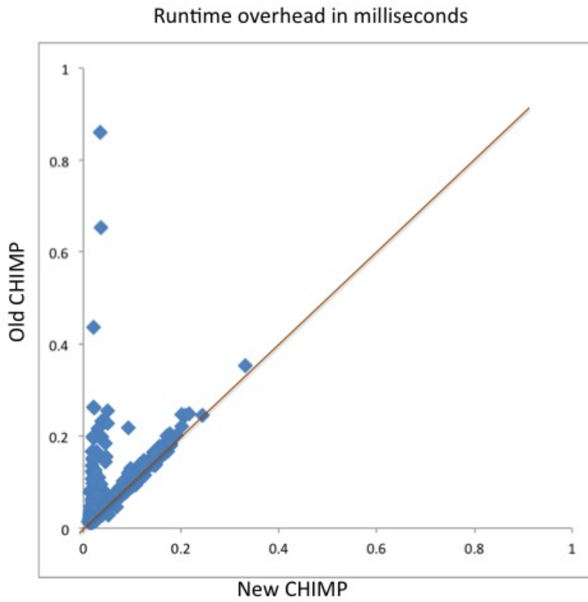


Fig. 12. Comparison of runtime overhead incurred by old CHIMP vs new CHIMP

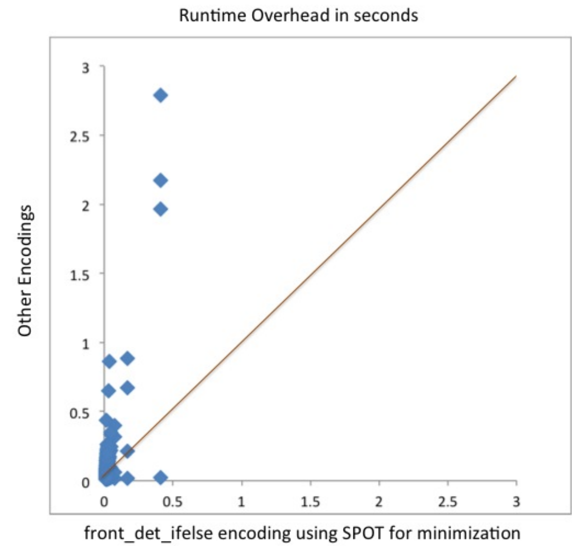


Fig. 13. Comparison of runtime overhead of front_det_ifelse encoding vs all other encodings

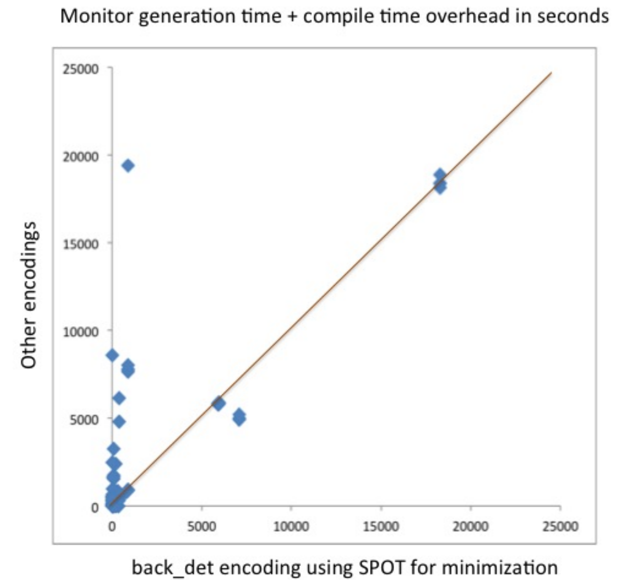


Fig. 14. Comparison of monitor generation time and compile time overhead of back_ass_alpha encoding vs all other encodings

References

[1] R. Armoni, D. Korchemny, A. Tiemeyer, M. Vardi, and Y. Zbar. Deterministic dynamic monitors for linear-time assertions. In *Proc. Workshop on Formal Approaches to Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*. Springer, 2006.

[2] M. d'Amorim and G. Roşu. Efficient monitoring of ω-languages. In *Proc. 17th International Conference on Computer Aided Verification*, pages 364–378, 2005.

[3] A. Duret-Lutz and D. Poitrenaud. SPOT: An extensible model checking library using transition-based generalized Büchi automata. *Modeling, Analysis, and Simulation of Computer Systems*, 0:76–83, 2004.

[4] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy. Automatic generation of schedulings for improving the test coverage of Systems-on-a-Chip. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 171–178, Washington, DC, USA, 2006. IEEE Computer Society.

[5] A. Møller. dk.brics.automaton. http://www.brics.dk/automaton/, 2004.

[6] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[7] D. Tabakov. *Dynamic Assertion-Based Verification for SystemC*. PhD thesis, Rice University, Houston, 2010.

[8] D. Tabakov, K. Rozier, and M. Y. Vardi. Optimized temporal monitors for SystemC. *Formal Methods in System Design*, 41(3):236–268, 2012.

[9] D. Tabakov and M. Vardi. Monitoring temporal SystemC properties. In *Proc. 8th Int'l Conf. on Formal Methods and Models for Codesign*, pages 123–132. IEEE, July 2010.

[10] D. Tabakov, M. Vardi, G. Kamhi, and E. Singerman. A temporal language for SystemC. In *FMCAD '08: Proc. Int. Conf. on Formal Methods in Computer-Aided Design*, pages 1–9. IEEE Press, 2008.

[11] D. Tabakov and M. Y. Vardi. Automatic aspectization of SystemC. In *Proceedings of the 2012 workshop on Modularity in Systems Software*, MISS '12, pages 9–14, New York, NY, USA, 2012. ACM.

Abstraction-Based Livelock/Deadlock Checking for Hardware Verification

In-Ho Moon and Kevin Harer
Synopsys Inc.
{mooni, kevinh}@synopsys.com

ABSTRACT

Livelock/deadlock is a well known and important problem in both hardware and software systems. In hardware verification, a livelock is a situation where the state of a design changes within only a smaller subset of the states reachable from the initial states of the design. Deadlock is a special case in which there is only one state in a livelock. However, livelock/deadlock checking has never been actively used in hardware verification in practice, mainly due to the complexity of the computation which involves finding strongly connected components.

This paper presents a practical abstraction-based livelock/deadlock checking algorithm for hardware verification. The proposed livelock/deadlock checking works on FSMs rather than the whole design. For each FSM, we make an abstract machine of manageable size from the cone of influence of the FSM. Once a livelock is found on an abstract machine, the livelock is justified on the concrete machine with trace concretization. Experimental results shows that the proposed abstraction-based livelock checking finds real livelock errors in industrial designs.

1. INTRODUCTION

Livelock/deadlock is a well known and important problem in both hardware and software systems. In hardware verification, a livelock is a situation where the state of a design changes within only a subset of the states reachable from the initial states of the design. In a state transition graph, a livelock is a set of states from which there is no path going to any other states that are reachable from the initial state. Since deadlock is a special case in which there is only one state in a livelock, deadlock checking can be done by livelock checking. Thus, livelock implies both livelock and deadlock in this paper. However, livelock checking[1] has never been actively used in hardware verification in practice, mainly due to the complexity of the computation which involves finding SCCs (Strongly Connected Components). Thus, livelock checking has been on hardware designer's wish list to verify their designs.

There have been many approaches on finding SCCs [13, 27, 28, 3, 20, 12]. Among these work, Xie and Beeral proposed a symbolic method finding *terminal SCCs* (in short, TSCCs) using BDDs (Binary Decision Diagrams [4]) in [27]. In a state transition graph, a TSCC is an SCC that does not have any outgoing edges to any state outside the SCC. Thus, a TSCC becomes a livelock group when the TSCC has any incoming edges to the SCC in the state transition graph representing a hardware design. However, even though the method in [27] is an improved method from its previous work [13] in symbolic approaches, it is still infeasible to apply the method to the industrial designs, simply due to the capacity problem of symbolic methods. In general, any BDD-based method can handle only up to several hundred latches without any abstraction or approximation techniques, whereas there can be millions of latches in industrial designs.

In this paper, we first present an improved BDD-based algorithm finding TSCCs from[27] in the following aspects. First, initial state is taken into account in finding TSCCs. Especially, the improved algorithm handles multiple initial states efficiently. Secondly, unreachable TSCCs are distinguished from reachable TSCCs which are more interesting to designers. Thirdly, we provide more intuitive state classification as main, transient, and livelock groups as opposed to transient and recurrence classes in [27]. In our classification, a set of transient states is further classified into main and transient groups. Recurrence class in [27] is mapped into livelock group in our classification. Main group is an SCC that contains the initial state. Transient group is a set of states that belong to neither main nor livelock group. There is one or zero main group in a design per one initial state.

This paper also presents a practical approach for checking livelock using abstraction techniques. The proposed livelock checking works on FSMs(Finite State Machines)[2] rather than the whole design. For each FSM, we make an abstract machine (by localization reduction [15]) of manageable size by the improved BDD method from the COI(Cone of Influence) of the FSM. Once a livelock is found on an abstract machine, the livelock is justified on the concrete machine with trace concretization using SAT (Satisfiability[9]) and simulation. When there is no livelock on the abstract machine, there is no guarantee for no livelock on the concrete machine. However, the bigger the abstract size is, the more confidence we have that no livelock exists on the concrete machine. The key benefit of this abstraction-based livelock checking is that it enables finding real livelock groups that cannot be found by tackling whole design directly.

Once an FSM is given, its COI is first computed. Then, an abstract machine is computed by finding N *influential latches* from the COI. Influential latches are the latches that are likely related with the FSM. N is either pre-defined or a user-defined number of latches in the abstract machine, or

[1] Livelock checking is different from liveness checking and the difference will be explained in Section 2.3.

[2] FSMs are either automatically extracted [26] or any sets of sequential elements that are user-specified.

gradually increased. In general, N is up to a few hundred latches. Influential latches are computed mainly by approximate state decomposition [6]. However, in many cases, the size of COIs is too big for even approximate state decomposition. Thus, a structural abstraction is applied by using connectivity and sequential depth before approximate state decomposition. This structural abstraction reduces the COI to a manageable size by approximate state decomposition.

There is another important hardware property called *toggle deadlock*. A state variable has a toggle deadlock if the state variable initially toggles and the state variable becomes a constant after a certain number of transitions. However, notice that this is not a constant variable since it initially toggles. Toggle deadlock may or may not happen depending on input stimuli in simulation. Therefore, toggle deadlock is also an important property to check with formal approaches.

Experimental results shows that the proposed abstraction-based approach finds real livelock and toggle deadlock errors from industrial designs.

The contributions of this paper are in the three aspects.

- Improved algorithm for livelock checking
  The proposed algorithm improved the existing algorithm [27] in many aspects, such as providing new state classification with initial state, handling multiple initial states, refining the search space efficiently with care states, early termination, and trimming out transient states.
- Abstraction-based livelock checking
  This paper presents theories and an implementation on abstraction-based livelock checking to handle large designs in practice.
- Toggle deadlock checking
  To the best of our knowledge, this paper presents the first method to solve toggle deadlock problem.

The remainder of this paper is organized as follows. Section 2 briefly recapitulates finding SCCs and TSCCs, and describes related work. Section 3 describes our improved algorithm for finding TSCCs. Section 4 explains how livelock is checked on FSM using abstraction. Section 5 describes how to check toggle deadlocks. Experimental results are presented and discussed in Section 6. We conclude with Section 7.

2. PRELIMINARIES

2.1 Finding SCCs

Given a graph, $G = (V, E)$ where G is an infinite transition system of the Kripke structure [8], V is a finite set of states and $E \subseteq V \times V$ is the set of edges, a strongly connected component (SCC) is a maximal set of state $U \subseteq V$ such that for every pair $(u, v) \in U$, u and v are reachable from each other, that is, u is reachable from v and v is reachable from u [27, 28].

Finding SCCs has a variety of applications in formal verification such as Buchi emptiness [11], LTL model checking [25], CTL model checking with fairness constraints [10], Liveness checking [16, 1], and so on.

The traditional approach to find SCCs is to use Tarjan's method [23]. Since this method manipulates the states of the graph explicitly, even though it runs in linear time in the size of the graph, the size of the graph grows exponentially as the number of state variables grows.

To overcome this state explosion problem in explicit algorithms, there have been many publications on symbolic algorithms. Ravi *et al.* [20] provided a taxonomy of those symbolic algorithms. One is SCC-hull algorithms (without enumerating SCCs) [11, 14, 24], and the other is SCC enumeration algorithms [28, 3, 12, 20]. The details are in [20].

2.2 Finding TSCCs

Even though TSCCs are a subset of SCCs in the states of a design, the algorithms for finding TSCCs can be significantly optimized since not all SCCs are of interest.

This section recapitulates the work on finding TSCCs by Xie and Beeral [27]. This algorithm classifies all states into either recurrence or transient class. Recurrence class is a set of TSCCs and the rest belongs to transient class. Let S be the set of states. With $i, j \in S$, $i \to j$ denotes that there is at least one path from i to j. Definition 1 defines *forward set* and *backward set* of a state.

DEFINITION 1. *The forward set of state $i \in S$, denoted by $F(i)$, is the set of states that have a path from i. That is, $F(i) = \{j \in S \mid i \to j\}$. Similarly, the backward set of state i, denoted by $B(i)$, is the set of states that have a path to i. That is, $B(i) = \{j \in S \mid j \to i\}$.*

LEMMA 1. *Let $i, j \in S$. If $j \in F(i)$, then $F(j) \subseteq F(i)$. Similarly, if $j \in B(i)$, then $B(j) \subseteq B(i)$.*

THEOREM 1. *A state $i \in S$ is recurrent if and only if $F(i) \subseteq B(i)$. In other words, i is transient if and only if $F(i) \nsubseteq B(i)$.*

THEOREM 2. *If state $i \in S$ is transient, then states in $B(i)$ are all transient. If state i is recurrent, on the other hand, states in $F(i)$ are all recurrent. In the latter case, set $F(i)$ is a recurrence class, and set $B(i) \backslash F(i)$ (if not empty) contains only transient states.*

Lemma 1, Theorem 1 and 2 are from [27]. Lemma 1 shows a subset relation between two forward sets as well as two backward sets when j is in either $F(i)$ or $B(i)$. Theorem 1 and 2 show how a state is determined whether the state belongs to either recurrence or transient class. Based on Theorem 1 and 2, all TSCCs can be found by performing forward and backward reachability iteratively. The detailed algorithm can be found in [27] and our improved algorithm is described in Section 3.2 with the comparisons to the original algorithm.

2.3 Related work

There are two types of properties in model checking; safety and liveness properties [16]. A safety property represents 'something bad never happens', whereas a liveness property represents 'something good eventually happens'. Liveness checking with a liveness property can be performed by finding SCCs [20]. Liveness checking can also be performed by safety checking with proper transformations [1].

Livelock checking is different from liveness checking in the sense that liveness checking requires a liveness property to work on a design, whereas livelock checking does not require any property and works on a design directly.

There have been many publications on finding all SCCs [24, 28, 3, 12]. Even though all TSCCs can be found by any of these approaches on finding all SCCs, it is not necessary to find all SCCs for finding all TSCCs since we are interested in finding only all TSCCs for livelock checking.

Hachtel *et al.* [13] proposed a symbolic approach to find all recurrence classes concurrently identifying all TSCCs by computing *transitive closure* [17] on the transition graph

with the Markov chain. Due to the complexity of transitive closure, this approach takes significantly more time and memory than a reachability-based approach does.

Qadeer *et al.* [19] proposed an algorithm to find single TSCC in the context of *safe replacement* in sequential equivalence checking [21, 22]. In this approach, multiple TSCCs are not considered.

Xie and Beeral proposed a reachability-based algorithm to find all TSCCs iteratively [27]. This is also a symbolic approach that outperforms the method in [13]. However, this approach does not consider initial states.

None of the above previous work on finding TSCCs was used in real designs in practice, due to the design sizes. Our abstraction-based approach is the first in publication to handle large designs in practice.

Case *et al.* [5] proposed a method finding *transient signals* using ternary simulation. A transient signal is a toggle deadlock on over-approximate reachable states. The toggle deadlock checking in this paper finds transients signals in exact reachable states.

3. IMPROVED LIVELOCK CHECKING

3.1 State Classification

The state classification in [27] consists of one transient class and one or more recurrence classes. However, in hardware verification, initial states are given to verify the hardware behavior only in reachable state space. One problem of the state classification in [27] is that there is no distinction between reachable TSCCs and unreachable TSCCs from the initial states. Also, the reachable TSCCs may vary depending on initial states.

We propose a new state classification that is shown in Figure 1, assuming that there is one single initial state. Handling multiple initial states is explained in Section 3.3.

DEFINITION 2. *STSCC is a sink TSCC that has incoming edges from any states outside the TSCC.*

We first define *sink TSCC* (in short, STSCC) in Definition 2. The new state classification consists of main group, transient group, livelock groups (reachable STSCCs) and unreachable TSCCs for a given initial state. The transient class in [27] is further classified into main group or transient group. Main group is an SCC containing the initial state and there exists either one or no main group. The recurrence classes in [27] are further classified into livelock groups (reachable STSCCs) and unreachable TSCCs. When there is no livelock, there exists only one SCC which is the main group.

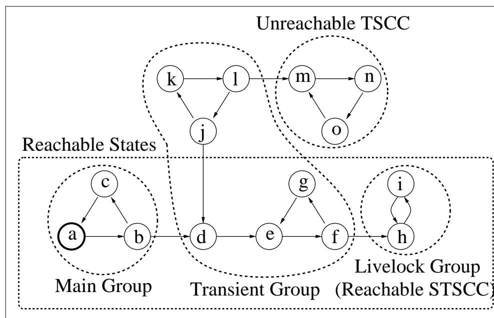


Figure 1: State classification.

In Figure 1, there are states a through o and a is the initial state that is marked with thick circle. Among all states, the reachable states are a through i inside the dotted rectangle. The unreachable states are j through o outside the dotted rectangle. There are five SCCs that are $\{a, b, c\}, \{e, f, g\}, \{h, i\}, \{j, k, l\}, and\{m, n, o\}$. Since a is the initial state, $\{a, b, c\}$ becomes the main group. $\{h, i\}$ and $\{m, n, o\}$ are TSCCs and only $\{h, i\}$ is a livelock group (reachable STSCC) since it is reachable from a. $\{m, n, o\}$ is called an unreachable TSCC. The rest states, $\{d, e, f, g, j, k, l\}$, belong to the transient group in which the states are contained in neither the main group nor the TSCCs.

3.2 Finding Livelock

We first define transition relation in Definition 3 to explain our algorithms to check livelock.

DEFINITION 3. *Let $x = \{x_1, \ldots, x_n\}$, $y = \{y_1, \ldots, y_n\}$, and $w = \{w_1, \ldots, w_p\}$ be sets of variables ranging over $B = \{0, 1\}$. A (finite state) machine is a pair of boolean functions $\langle Q(x, w, y), I(x)\rangle$, where $Q : B^{2n+p} \rightarrow B$ is 1 if and only if there is a transition from the state encoded by x to the state encoded by y under the input encoded by w. $I : B^n \rightarrow B$ is 1 if the state encoded by x is an initial state. $Q(x, w, y)$ is called transition relation. The sets x, y, and w are called the present state, next state, and input variables, respectively.*

The procedure $ComputeForwardSet$ in Figure 2 is a modified version of the procedure $forward_set$ in [27] in order to compute forward set of a given state s only within the given care states *careSet* in the procedure and to perform *early termination* when *stop* is not ZERO (empty BDD). $\Downarrow$ represents a *restrict operator* [7] that is used to minimize the transition relation with respect to *careSet* in Line 2. The minimized transition relation is denoted by $\tilde{Q}$. In Line 7, $y \leftarrow x$ represents that y variables are replaced by x variables by BDD substitution. Early termination is another big difference from $forward_set$ in [27] and is used in Figure 3. This is to bail out computing forward set as soon as any newly reached state intersects with the states in *stop* as in Line 11. *BddIteConstant* is a BDD ITE(if-then-else) operation without creating a new BDD node. O is an array of states to store newly reached states at each iteration and O is called *onion rings*. These onion rings are used later in Section 3.3. $ComputeForwardSet$ returns the forward set $F(s)$ and the onion rings O.

```
   ComputeForwardSet(Q, careSet, s, stop) {
1      F(s) = ZERO;
2      Q̃(x, w, y) = Q(x, w, y) ⇓ careSet;
3      frontier(x) = s;
4      Put s in O;
5      while (frontier(x) ≠ ZERO) {
6          image(y) = ∃x, w. Q̃(x, w, y) ∧ frontier(x);
7          image(x) = image(y)|y←x;
8          F(s) = F(s) ∨ image(x);
9          frontier = image(x) ∧ ¬F(s);
10         Put frontier in O;
11         if (BddIteConstant(frontier, stop, ZERO) != ZERO)
12             break;
13     }
14     return (F(s), O);
   }
```

Figure 2: Computing forward set.

$ComputeBackwardSet$ is a dual procedure to $Compute ForwardSet$, except not using *stop* and not computing the onion rings O.

Figure 3 is a procedure for finding TSCCs from the given set of states S. The procedure $FindTSCCs$ is a modified version of the procedure $State_classification$ in [27]. The modified procedure utilizes care states *careSet*, assuming S

is not necessarily all state space. T is a set of transient states in S, and R is an array of TSCCs in S. *PickOneState* in Line 5 picks a random state from *careSet* as a seed state to find a TSCC. In Line 7, early termination is used in computing the forward set $F(s)$, by setting *stop* in Figure 2 as the negation of $B(s)$. This is because while we compute $F(s)$ within $B(s)$ for the state s, once any state outside $B(s)$ is reachable from s, all states in $B(s)$ are transient. Another big difference is trimming transient states in Line 12 and 16. $TrimTransient(Q, careSet, T, dir)$ trims out the transient states from the current care states by the given direction (dir) that is either PREFIX, SUFFIX, or BOTH. PREFIX(SUFFIX) means to trim out the lasso prefix(suffix) states. This is the same technique used in finding SCCs [20]. Finally, *FindTSCCs* returns R (a set of TSCCs) and T (a set of transient states).

```
   FindTSCCs(Q, S) {
1      R = { };
2      T = ZERO;
3      careSet = S;
4      while (careSet ≠ ZERO) {
5          s = PickOneState(careSet);
6          B(s) = ComputeBackwardSet(Q, careSet, s);
7          F(s) = ComputeForwardSet(Q, careSet, s, ¬B(s));
8          if (F(s) ⊆ B(s)) {
9              R = R ∪ F(s);
10             T = T ∨ (B(s) ∧ ¬F(s));
11             careSet = careSet ∧ ¬B(s);
12             TrimTransient(Q, careSet, T, PREFIX);
13         } else {
14             T = T ∨ (s ∨ B(s));
15             careSet = careSet ∧ ¬(s ∨ B(s));
16             TrimTransient(Q, careSet, T, BOTH);
17         }
18     }
19     return (R, T);
   }
```

Figure 3: Finding TSCCs.

```
   FindLivelock(Q, S, s) {
1      (F(s), O) = ComputeForwardSet(Q, S, s, ZERO);
2      B(s) = ComputeBackwardSet(Q, S, s);
3      reached = F(s) ∨ s;
4      if (F(s) ⊆ B(s)) {
5          M = F(s);
6          R = { };
7          T = ZERO;
8      } else {
9          M = F(s) ∧ B(s);
10         careSet = F(s) ∧ ¬(M ∨ s);
11         TrimTransient(Q, careSet, T, PREFIX);
12         (R, T_R) = FindTSCCs(Q, careSet);
13         if (s ∉ M)
14             T_R = T_R ∨ s;
15         T_U = B(s) ∧ ¬M;
16         T = T_R ∨ T_U;
17     }
18     return (M, R, T, reached, O);
   }
```

Figure 4: Finding livelock.

Figure 4 shows the procedure to perform our new state classification. As explained in Section 3.1, we find main group (M), transient group (T), and livelock groups (R) from the given initial state (s) within the given care states (S). *FindLivelock* starts computing forward set $F(s)$ and backward set $B(s)$ in Line 1 and 2. In Line 3, *reached* is the reached states from s in S. If $F(s) \subseteq B(s)$ in Line 4, there is no livelock in S. In this case, $F(s)$ becomes the main group and both R and T are set to empty in Line 5-7. If $F(s) \not\subseteq B(s)$ in Line 8, there must exist at least one livelock group. In this case, M is computed by intersecting $F(s)$ and $B(s)$ in Line 9. *careSet* is set to a subset of $F(s)$ in Line 10. The lasso prefix states in *careSet* are trimmed out in Line 11. T_R represents the set of transient states that are reachable from s. In Line 12, R and T_R are computed by calling *FindTSCCs* with *careSet*. If $s \notin M$ (means that the main group is empty), s is added to T_R in Line 13-14. T_U represents the set of transient states that are unreachable from s and T_U is computed in Line 15. T is computed by union of T_R and T_U in Line 16.

3.3 Multiple Initial States

It is possible for a design to have multiple initial states when some of the state variables do not have concrete initial values. In the presence of multiple initial states, finding livelock groups has to be devised correctly to avoid false positives and redundant computations.

Figure 5 shows an example with multiple initial states. In this example, there are six states, $S = \{a, b, c, d, e, f\}$. There are two SCCs, $\{a, b, c\}$ and $\{d, e, f\}$. We can see that $\{d, e, f\}$ is a TSCC. a and d are initial states, $I = \{a, d\}$, as shown with thick circles. Suppose that we compute livelock by calling $FindLivelock(Q, S, I)$. Then, we get $F(I) = B(I) = M = \{a, b, c, d, e, f\}$ and $R = \{\}$ which is not correct since there is a reachable TSCC. Now, let us try to call *FindLivelock* for each single initial state. First for the initial state a, we get $F(a) = \{a, b, c, d, e, f\}$ and $B(a) = \{a, b, c\}$. This gives us $M_a = \{a, b, c\}$ and $R_a = \{d, e, f\}$. There is a livelock group R_a for the initial state a. Now, for the initial state d, $F(d) = \{d, e, f\}$ and $B(d) = \{a, b, c, d, e, f\}$. This gives us $M_d = \{d, e, f\}$ and $R_d = \{\}$ and $T_U = \{a, b, c\}$. There is no livelock group for the initial state d. Therefore, we can see that livelock checking has to be applied for each single initial state separately in the presence of multiple initial states.

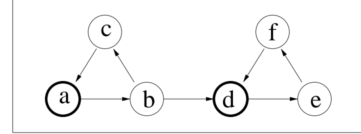


Figure 5: Multiple initial states.

THEOREM 3. *When there are two initial states (i_0 and i_1), if i_1 is included in the reached states from i_0, the livelock groups from i_1 are a subset of the livelock groups from i_0.*

PROOF. Since i_1 is included in the reached states from i_0, i_1 is in either main, transient, or livelock groups from i_0. When i_1 is in the main group, the same livelock groups from i_1 are obtained. When i_1 is in the transient group, all or a subset of the livelock groups i_1 is obtained. When i_1 is in one of the livelock groups, the livelock group including i_1 becomes the main group from i_1, and no livelock group exists from i_1 since the other livelock groups from i_0 become unreachable TSCCs from i_1. From the above three cases, no new livelock group is obtained from i_1 compared to the ones from i_0. Therefore, the livelock groups from i_1 are a subset of the livelock groups from i_0. □

Theorem 3 says that when there is large number of initial states, we can skip livelock checking for any initial states that are already in the forward sets of other initial states. In Figure 5, livelock checking for the initial state d can be skipped because of $d \in F(a)$, assuming that a is used first. However, there is an order dependency on which initial state is used first. If d is used first, we still need to run livelock checking with a. In practice, the number of calls to

FindLivelock is greatly reduced because of Theorem 3 in the presence of multiple initial states.

Figure 6 is the top-level procedure that checks livelock with multiple initial states. *CheckLivelock* takes transition relation(Q), a set of states(S), a set of initial states(I), and a concrete machine(C) as procedure inputs. The use of C is explained in Section 4. *CheckLivelock* first finds livelock groups in the reachable states in Line 1-17 and then it finds TSCCs in the unreachable states in Line 18-23. The *while* loop (Line 6-17) performs livelock checking for a current initial state s until all initial states are covered with iteration index k. For this, *remaining* is initially set to I in Line 3 and updated by eliminating the newly reached states $reached_k$ from *remaining* in Line 13. *reached* is the reached states from all initial states. *reached* is initially set to ZERO in Line 1 and updated by adding $reached_k$ that is the reached states from s in Line 12. Then, the next initial state is chosen from *remaining* in Line 15. T_U is the union of unreachable transient states from each initial state. T_U is initially set to ZERO in Line 2 and updated by adding the unreachable states of T_k in Line 14. For the current initial state s, *FindLivelock* is called in Line 7. $|R_k|$ represents the number of livelock groups in R_k in Line 8. For each R_k^j, a trace $trace_k^j$ is generated in Line 9 and the livelock is reported with the trace in Line 10. Generating trace is explained in Section 4.2 and reporting livelock is explained in Section 4.3.

```
   CheckLivelock(Q, S, I, C) {
1      reached = ZERO;
2      T_U = ZERO;
3      remaining = I;
4      k = 0;
5      s = PickOneState(I);
6      while (s ≠ ZERO) {
7          (M_k, R_k, T_k, reached_k, O_k) = FindLivelock(Q, S, s);
8          for (j = 0; j < |R_k|; j++) {
9              trace_k^j = GenerateTrace(C, R_k^j, s, O_k);
10             ReportLivelock(s, M_k, R_k^j, T_k, trace_k^j);
11         }
12         reached = reached ∨ reached_k;
13         remaining = remaining ∧ ¬reached_k;
14         T_U = T_U ∨ (T_k ∧ ¬reached_k);
15         s = PickOneState(remaining);
16         k++;
17     }
18     careSet = ¬(reached ∨ T_U);
19     if (careSet ≠ ZERO) {
20         R_k = FindTSCCs(Q, careSet);
21         for (j = 0; j < |R_k|; j++)
22             ReportUnreachLivelock(R_k^j);
23     }
   }
```

Figure 6: Checking livelock.

Once all reachable livelock groups are found, we next find unreachable TSCCs. We set the care states *careSet* to the negation of all visited states so far in Line 18, then call *FindTSCCs* with *careSet* in Line 20. If there is any unreachable TSCC, the TSCC is reported in Line 22.

4. LIVELOCK CHECKING ON FSM

To check whether a livelock exists in a design or not, the checking should be done on the whole design. However, this is infeasible due to the size of the design in practice. Thus, we propose a practical method for checking livelock on FSMs on the design.

Even when we check livelock on an FSM, the entire COI logic of the FSM must be considered in order to get an exact result on livelock. However, this is still computationally very expensive or not feasible, in most real designs. Thus, we propose a framework for abstraction-based livelock checking on an abstracted COI of the FSM. Once we find a livelock on the abstract machine, we justify whether the livelock exists on the concrete machine. Notice that a livelock on the abstract machine can be mapped into more than one livelock on the concrete machine.

Figure 7 shows how an abstract machine is obtained from the COI of an FSM. Suppose an FSM that has two state variables f and g. Then, we compute the COI of the FSM. Suppose that there are state variables $\{a, b, c, d, e\}$ in the COI of the FSM. The size of the abstract machine is predefined and let us suppose that the size is N. Then, a set of influential latches from the COI is computed from the FSM variables. The minimum abstract machine is the FSM itself and the maximum abstract machine is the concrete machine. In this example, N=4 and we get the abstract machine $\{f, g, d, e\}$.

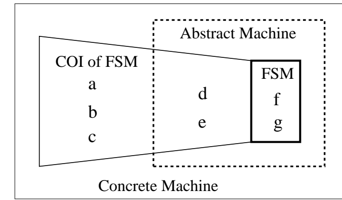


Figure 7: Abstract machine.

THEOREM 4. *If any state in a livelock group on an abstract machine is reachable from the initial state on the concrete machine, the livelock exists on the concrete machine.*

PROOF. Since the abstraction is an over-approximation, the set of all transitions on the abstract machine is a superset of the set of all transitions on the concrete machine. Since there is no path from any state in the livelock group to any state in the main group on the abstract machine, there is still no path from any projected states of the livelock group on the concrete machine to any projected states of the main group on the concrete machine. Now, suppose that the livelock does not exist on the concrete machine. In order for the livelock not to exist on the concrete machine, the only condition is that there is no path from the projected main group to the projected livelock group on the concrete machine. In other words, the projected livelock group has to be unreachable from the initial state. However, this contradicts the assumption that any state of the livelock group is reachable from the initial state on the concrete machine. Therefore, the livelock group still exists on the concrete machine. □

Thanks to Theorem 4, this abstraction-based livelock finds a livelock on small abstract machine using BDD-based symbolic method, then justifies the existence of the livelock on the concrete machine by trace concretization in Section 4.2, by using SAT techniques that can handle large designs. The abstraction-based livelock checking is an incomplete method in the sense that it does not provide the proof of no livelock unless the checking is performed on a concrete machine. No livelock on an abstract machine does not guarantee no livelock on the concrete machine. However, the abstraction-based livelock checking enables finding real livelock errors on industrial large designs.

4.1 Causality Checking

Let V be the set of state variables in an abstract machine for livelock checking. Suppose that $R(V)$ is the reached states in the abstract machine and $L(V)$ is a livelock group

containing a TSCC. Also, suppose that v is a state variable in V. We are interested in whether v *contributes* to the livelock as in Definition 4. This is called *variable causality.*

DEFINITION 4. *When a livelock exists in the abstract machine, a variable v in V* contributes *to the livelock if the livelock disappears by eliminating v from the abstract machine. In other words, there is no livelock in another abstract machine that is composed of the variables, $V\backslash v$.*

Equation 1 shows a condition for existence of livelock.

$$L(V) \subset R(V) \quad (1)$$

Now, let $\tilde{R}$ be the quantified reached states and $\tilde{L}$ be the quantified livelock states with respect to a state variable v, as shown in Equation 2 and 3.

$$\tilde{R}(V\backslash v) = \exists_v . R(V) \quad (2)$$

$$\tilde{L}(V\backslash v) = \exists_v . L(V) \quad (3)$$

Then, it is determined by Equation 4 to check whether the variable v contributes to the livelock. Theorem 5 says that if Equation 4 holds, v contributes to the livelock.

$$\tilde{L}(V\backslash v) \subset \tilde{R}(V\backslash v) \quad (4)$$

THEOREM 5. *When a livelock group is found on an abstract machine ($L(V) \subset R(V)$), if $\tilde{L}(V\backslash v) \subset \tilde{R}(V\backslash v)$ holds for a variable v, the variable v contributes to the livelock.*

PROOF. Let M_1 be the machine consisting of V and suppose that a livelock group exists in M_1. Let M_2 be the machine consisting of $(V\backslash v)$ by eliminating v from M_1. Also, let T_1 (T_2) be the set of transitions in M_1 (M_2), respectively. Since M_2 is an over-approximated machine from M_1, M_2 has more transitions than M_1 ($T_1 \subset T_2$). Let T_d be the difference between T_1 and T_2. If there is any transition (in T_d) that makes a path from any state in the livelock to any state in the main group in M_2, the livelock group merges into the main group and both groups become a single SCC, yielding $\tilde{L}(V\backslash v) = \tilde{R}(V\backslash v)$. Thus, M_2 becomes a machine without the livelock. This means that v is a necessary variable to have the livelock in M_1. Therefore, if $\tilde{L}(V\backslash v) = \tilde{R}(V\backslash v)$, v contributes to the livelock. □

This causality checking can also be applied to a set of variables, especially with FSM variables, in order to report whether the livelocks are related with the FSM. Let F be the set of variables in an FSM and C be the set of variables in the COI of the FSM. Suppose that $R(F, C)$ is the reached states in the abstract machine and $L(F, C)$ is a livelock group containing a TSCC. The quantified reached states and the quantified livelock states are computed in Equation 5 and 6 with respect to the FSM variables, respectively.

$$\tilde{R}(C) = \exists_F . R(F, C) \quad (5)$$

$$\tilde{L}(C) = \exists_F . L(F, C) \quad (6)$$

Then, Equation 7 shows the causality checking with the FSM variables to check whether the FSM variables contribute to the livelock.

$$\tilde{L}(C) = \tilde{R}(C) \quad (7)$$

4.2 Trace concretization

Once a livelock group is found on an abstract machine, we need to justify whether the livelock group is reachable on the concrete machine. This can be done by the following three steps. The first step is to pick a target state in the livelock group. The target state is chosen randomly from the livelock group, but is one of the closest states to the initial states by using the onion rings O_k in Figure 6. The second step is to generate an abstract trace. Starting from the target state, an abstract trace can be computed by applying BDD-based pre-image computation iteratively until the initial state is reached. The third step is to generate a concrete trace by making a BMC (Bounded Model Checking [2]) problem from the abstract trace, in order to see whether the livelock group is reachable on the concrete machine. An efficient approach for concretization was proposed in [18]

4.3 Reporting Livelock

Once a concrete trace is generated for a livelock group, the livelock is real on the concrete machine. We report the livelock group with the state classification mentioned in Section 3.1. A livelock group is reported with its initial state, the main group, transient group, and the unreachable states in terms of the number of states and the percentage in each group on the abstract machine.

By looking at the transient and livelock groups, we can see what fraction of the state space is in problematic zone. A good design is expected to have only one main group per one initial state without any transient and livelock groups, unless the design has an intended reset sequence to a normal mode.

5. TOGGLE DEADLOCK CHECKING

There is another important design property, called *toggle deadlock* that is related to livelock. A livelock may occur for multiple state variables of a design, whereas a toggle deadlock may occur on a single state variable. A state variable has a toggle deadlock if the variable initially toggles, but the variable gets stuck at a constant value after a certain number of cycles.

Figure 8 shows an example of toggle deadlock. There are two state variables $\{a, b\}$ and four states $\{s_0, s_1, s_2, s_3\}$ as in the example. Provided that s_0 is the initial state, the main group is $\{s_0, s_1\}$ and the livelock group is $\{s_2, s_3\}$. Once the state transition reaches to s_2 that is a state in the livelock group, the value of b gets stuck at 1, whereas a still toggles. Thus, we say that b has a toggle deadlock.

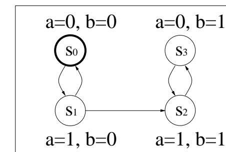


Figure 8: Toggle deadlock.

THEOREM 6. *If there is no STSCC in a design, there is no toggle deadlock on any variable.*

PROOF. To be a toggle deadlock, a variable is supposed to toggle at a cycle and to hold the value forever from the cycle. No STSCC implies that there is only main group in the design. If a variable appears as constant in the main group, the variable is a constant. However, the main group does not have any prefix behavior. This means it is not possible for the variable to get toggled before the main group. Therefore, no STSCC implies no toggle deadlock. □

Theorem 6 shows that toggle deadlock occurs in the presence of a livelock. It is also possible that there is no toggle deadlock on a design that has a livelock. Thus, toggle deadlock on a state variable can be computed by two steps. First,

Design	Statistics					Results										
	L	I	F	T	COI	N	Llk	Dlk	New1			New2			TraceGen	
									Time	Mem	Ops	Time	Mem	Ops	Time	Len
D1	1163	1330	7	-	632	30	0	-	16:20	107.1	44	15:31	107.9	41	-	-
						60	0	-	41:11	136.7	66	41:31	137.7	66	-	-
						90	0	-	2:30:33	224.1	81	2:24:50	224.3	81	-	-
						120	-	-	time-out (> 24h)			time-out (> 24h)			-	-
D2	385	352	25	-	68	30	0	-	0:01	21.0	12	0:01	21.1	12	-	-
						60	0	-	0:40	38.7	28	0:40	38.7	28	-	-
						68	12032	-	4:47:52	95.7	386439	0:53:58	96.5	70329	-	-
D3-F1	32541	912	2	-	28941	30	1	-	0:49	140.4	54	0:49	140.5	55	6:37	66
D3-F2	32541	912	4	-	28930	4	1	-	0:09	129.1	9	0:09	129.1	12	1:54	14
			-	4	28930	30	-	1	1:31	132.8	168	1:31	132.8	172	2:11	14

Table 1: Experimental results.

we find STSCCs on an abstract machine from the state variable. The abstract machine is made in the same way as in livelock checking on FSM. Secondly, we evaluate the value of the state variable in the livelock if the livelock exists.

Figure 9 shows the procedure that checks toggle deadlock on a given state variable t. *CheckToggleDeadlock* takes transition relation (Q), a set of states (S), a set of initial states (I), a concrete machine (C), and the state variable (t) as procedure inputs. *CheckToggleDeadlock* is similar to *CheckLivelock* in Figure 6. For each reachable livelock group R_k^j in Line 6, *TestToggleDeadlock* checks whether the value of t toggles or not in the livelock group and returns *dlk* and c in Line 7. *dlk* represents whether the state variable is in toggle deadlock or not, and c is the constant value (0 or 1) in the case of toggle deadlock.

```
   CheckToggleDeadlock(Q, S, I, C, t) {
1      remaining = I;
2      k = 0;
3      s = PickOneState(I);
4      while (s ≠ ZERO) {
5          (R_k, reached_k, O_k) = FindLivelock(Q, S, s);
6          for (j = 0; j < |R_k|; j++) {
7              (dlk, c) = TestToggleDeadlock(R_k^j, t);
8              if (dlk) {
9                  trace_k^j = GenerateTrace(C, R_k^j, s, O_k);
10                 ReportToggleDeadlock(s, R_k^j, trace_k^j, c);
11             }
12         }
13         remaining = remaining ∧ ¬reached_k;
14         s = PickOneState(remaining);
15         k++;
16     }
   }
```

Figure 9: Checking toggle deadlock.

6. EXPERIMENTAL RESULTS

We have implemented the proposed livelock checking and toggle deadlock checking algorithms. Table 1 shows our experimental results on livelock and toggle deadlock checking, generated on a 1.4 GHz Intel processor machine with 4 GB memory running Red Hat Linux.

The first column lists the design names. The next five columns present the statistics on the designs, in terms of the number of latches (L), the number of inputs (I), the number of latches in FSM (F), the number of toggle signals to check (T), and the number of latches in the COI of either FSM and a toggle signal (COI). The next three columns show the results on livelock and toggle deadlock checking. The column with N shows how many latches were in the abstract machine. The column with *Llk* shows how many livelock groups are found and the column with *Dlk* shows how many toggle deadlock are found. The next six columns compare the performance between two methods ($New1$ and $New2$), in terms of time($Time$), memory(Mem), and the number of image/pre-image computations(Ops). $New1$ is the proposed method without the trimming technique, whereas $New2$ is the proposed method with the trimming technique. The times are in the form of *hh:mm:ss* and the memory consumptions are in M-byte. The final two columns($TraceGen$) show the results on trace generation on concrete machine for the livelock or toggle deadlock found by $New2$, and $Time$ shows the time spent for trace generation and Len shows the trace length.

We have chosen 3 industrial designs ($D1$, $D2$, and $D3$). For each design, we have run livelock or toggle deadlock checking on several sizes of abstract machines with the multiples of 30 latches. We have set the maximum run time to 24 CPU hours.

In $D1$, there is one FSM automatically extracted. The FSM consists of 7 latches and contains 632 latches in its COI. We can see that the run time is exponentially increased, depending on the size of the abstract machine. On this design, the livelock checking became infeasible when N=120.

In $D2$, there is also one FSM that was user-specified. The FSM consists of 25 latches and contains only 68 latches in its COI. This design has a livelock group. However, the livelock was not detected when N=30 and N=60. The livelock was detected only when all the latches in the COI were included in the abstract machine. In other words, the abstract machine is the concrete machine at the FSM point of view. Since the livelock was found on the concrete machine, trace concretization is not required since the abstract trace in Section 4.2 is already a concrete trace.

$D2$ is the only design showing a significant performance difference between $New1$ and $New2$ in the table. This is because this design has many transient states as well as many livelock groups. In this case, the trimming technique significantly reduced the number of image/pre-image operations from 386K to 70K (5.5X reduction) that gave big speed-up from 5 hours to 1 hour (5X speed-up). This shows that the trimming technique helps the performance when there are many transient states. When there is no transient states, the trimming technique becomes a pure overhead as shown in $D3$. However, the overhead is almost negligible from the experiment.

In $D3$, there are two FSMs ($F1$ and $F2$). $F1$ is composed of 2 latches and a livelock was found with N=30 within 49 seconds. The livelock was justified by trace concretization that took 397 seconds, and the trace length was 66. $F2$ is composed of 4 latches and a livelock was found with N=4 (the FSM itself) in 9 seconds. The livelock was also justified by trace concretization that took 114 seconds, and the trace length was 14. We have also tried the toggle dead-

lock checking on $F2$ separately from the livelock checking. A toggle deadlock was found in 90 seconds and the concrete trace was generated in 131 seconds. $D3$ shows the value of abstraction-based livelock and toggle deadlock checking.

Table 2 shows a comparison on finding all SCCs with four algorithms (XB [28], Lockstep [3], Skeleton [12], IXB [20]) on the design $D2$ from Table 1. In this design, the number of recurrent states is 2.07e8 and the number of transient states is 1.2e6 that is only 0.6% of all states. However, it turned out that how to handle these transient states efficiently is the key factor in the performance. One main difference between XB and IXB is that IXB trims out those transient states as much as possible. This trimming technique makes the IXB method outperform on this design: faster in time (more than 15X) and fewer number of image operations (more than 10X) than the other methods. This explains why $New2$ outperformed on $D2$ in Table 1. Table 2 also shows why livelock checking is done by finding TSCCs instead of SCCs. Finding all livelock groups took 54 minutes, whereas finding all SCCs took 100 minutes (2X) even with IXB.

Method	Time	Memory	Ops	SCCs	States
XB	84:07:56	98.2	1013333	19458	2.08e8
Lockstep	45:55:53	237.3	2590724		
Skeleton	26:01:54	266.5	2609008		
IXB	1:39:47	92.5	102990		

Table 2: Finding all SCCs in D2.

7. CONCLUSIONS

We have presented a framework for abstraction-based livelock and toggle deadlock checking, in order to handle large designs in practice. Since exact livelock and toggle deadlock checking is infeasible on real designs directly, our approach is to check livelock and toggle deadlock on abstract machine of either an FSM or a toggle signal. Once we find a livelock or toggle deadlock, we justify the livelock or toggle deadlock on the concrete machine by concretizing the abstract trace on the concrete machine.

Even though the proposed approach does not prove the non-existence of livelock or toggle deadlock on a design unless the design is small enough to handle, this approach finds livelocks or toggle deadlocks on the design if there exists.

To the best of our knowledge, it is the first approach to use the abstraction-based livelock checking and also the first approach for checking toggle deadlock. The experimental results showed that the abstraction-based approach finds livelock errors on the real designs.

As future work, we are interested in improving the concretization, finding more accurate influential latches, and optimizing the computations with multiple FSMs or toggle signals by considering the overlaps in their COIs.

8. REFERENCES

[1] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *International Workshop in Formal Methods for Industrial Critical Systems*, pages 160–177, 2002.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 193–207, Amsterdam, The Netherlands, Mar. 1999. LNCS 1579.

[3] R. Bloem, H. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in n log n symbolic steps. In *Formal Methods in Computer Aided Design*, pages 37–54, 2000.

[4] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, Aug. 1986.

[5] M. Case, H. Mony, J. Baumgartner, and R. Kanzelman. Enhanced verification by temporal decomposition. In *Formal Methods in Computer Aided Design*, pages 37–54, 2009.

[6] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi. Automatic state space decomposition for approximate fsm traversal based on circuit analysis. *IEEE Transactions on Computer-Aided Design*, 15(12):1451–1464, Dec. 1996.

[7] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 126–129, Nov. 1990.

[8] O. G. E. M. Clarke and D. Peled. *Model Checking*. The MIT Press, 1999.

[9] N. Een and N. Sorensson. *MiniSat*. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat.

[10] E. A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8:275–306, 1987.

[11] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium of Logic in Computer Science*, pages 267–278, June 1986.

[12] R. Gentilini, C. Piazza, and A. Policriti. Computing strongly connected components in a linear number of symbolic steps. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 573–582, 2003.

[13] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian analysis of large finite state machines. *IEEE Transactions on Computer-Aided Design*, 15(12):1479–1493, Dec. 1996.

[14] R. Hojati, H. Touati, R. P. Kurshan, and R. K. Brayton. Efficient ω-regular language containment. In *Computer Aided Verification*, pages 371–382, Montréal, Canada, June 1992.

[15] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.

[16] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, Mar. 1977.

[17] Y. Matsunaga, P. C. McGeer, and R. K. Brayton. On computing the transitive closure of a state transition relation. In *Proceedings of the Design Automation Conference*, pages 260–265, June 1993.

[18] K. Nanshi and F. Somenzi. Constraints in one-to-many concretization for abstraction refinement. In *Proceedings of the Design Automation Conference*, pages 569–574, 2009.

[19] S. Qadeer, R. K. Brayton, V. Singhal, and C. Pixley. Latch redundancy removal without global reset. In *Proceedings of the International Conference on Computer Design*, pages 432–439, 1996.

[20] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 143–160. Springer-Verlag, Nov. 2000. LNCS 1954.

[21] V. Singhal. *Design replacements for sequential circuits*. Ph.D. dissertation, University of California at Berkeley, 1996.

[22] V. Singhal, C. Pixley, A. Aziz, and R. K. Brayton. Theory of safe replacements for sequential circuits. *IEEE Transactions on Computer-Aided Design*, 20(2):249–265, Feb. 2001.

[23] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.

[24] H. J. Touati, R. K. Brayton, and R. P. Kurshan. Testing language containment for ω-automata using BDD's. *Information and Computation*, 118(1):101–109, Apr. 1995.

[25] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, UK, June 1986.

[26] T.-H. Wang and T. Edsall. Practical FSM analysis for verilog. In *IVC-VIUF '98: Proceedings of the International Verilog HDL Conference and VHDL International Users Forum*, pages 52–58, 1998.

[27] A. Xie and P. A. Beeral. Efficient state classification of finite-state markov chains. *IEEE Transactions on Computer-Aided Design*, 17(12):1334–1339, Dec. 1998.

[28] A. Xie and P. A. Beeral. Implicit enumeration of strongly connected components and an application to formal verification. *IEEE Transactions on Computer-Aided Design*, 19(10):1225–1230, Oct. 2000.

Keyword Index

abstraction 4
abstraction-based 46
assertion 38

Boolector 28

Data-path equivalence checking 9
deadlock checking 46
dynamic verification 38

IC3 19

Lambda 28
Lemmas on Demand 28
livelock checking 46

Model Checking 19
modelchecking 4

Polynomial equivalence checking 9

QF_BV SMT solving 9

reparameterization 4

SAT solver 19
SMT 28
synthesis 4
systemC 38

verification 4

Author Index

Biere, Armin 28
Brayton, Robert 9

Cabodi, Gianpiero 19
Case, Michael 9
Cleaveland, Rance 1

Dutta, Sonali 38

Een, Niklas 4

Fujita, Masahiro 2

Goswami, Dhiraj 3

Harer, Kevin 46

Long, Jiang 9

Mishchenko, Alan 4, 19
Moon, In-Ho 46

Niemetz, Aina 28

Palena, Marco 19
Preiner, Mathias 28

Tabakov, Deian 38

Vardi, Moshe Y. 38