11th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science

Jan Bouda, Lukáš Holík, Jan Kofroň, Jan Strejček and Adam Rambousek

editors

MEMICS

2016

October 21—23

Masaryk University

www.memics.cz

MEMICS 2016

Jan Bouda, Lukáš Holík, Jan Kofroň, Jan Strejček,
and Adam Rambousek (Eds.)

MEMICS 2016

11th Doctoral Workshop on Mathematical
and Engineering Methods in Computer Science

Telč, Czech Republic, October 21-23, 2016

Masaryk University
Brno 2016

Editors

Jan Bouda
Faculty of Informatics
Masaryk University
Botanická 68a, Brno, Czech Republic

Lukáš Holík
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, Brno, Czech Republic

Jan Kofroň
Faculty of Mathematics and Physics
Charles University
Malostranské nám. 25, 118 00 Praha 1, Czech Republic

Jan Strejček
Faculty of Informatics
Masaryk University
Botanická 68a, Brno, Czech Republic

Adam Rambousek
Faculty of Informatics
Masaryk University
Botanická 68a, Brno, Czech Republic

Subject classification: Information technology

ISBN 978-80-210-8362-2

Preface

This volume contains the proceedings of the 11th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2016) held in Telč, Czech Republic, during October 21-23, 2016.

The aim of the MEMICS workshop series is to provide an opportunity for PhD students to present and discuss their work in an international environment. MEMICS focuses broadly at formal and mathematical methods in computer science and engineering and their applications.

In addition to regular papers, MEMICS workshops traditionally invite PhD students to submit a presentation of their recent research results that have already undergone a rigorous peer-review process and have been presented at a high-quality international conference or published in a recognized journal.

There were 28 submissions from PhD students. We received 14 regular papers; each one was thoroughly evaluated by at least three Program Committee members, who also provided extensive feedback to the authors. This year, we also organize a poster session presenting an ongoing work of students. In addition to regular papers, we received 5 presentation abstracts, and 9 poster abstracts. We accepted 9 regular papers, in addition to that the work described in other 2 regular papers will be presented, along with 5 presentations and 9 posters introducing already published and ongoing work, respectively. The presentation abstracts are also included in these proceedings.

The highlights of the MEMICS 2016 program includes three keynote lectures delivered by internationally recognized researchers from various areas of computer science. The speakers are:

- Armin Biere (Johannes Kepler University in Linz, Austria)
- Luca Bortolussi (University of Trieste, Italy)
- Roland Meyer (TU Kaiserslautern, Germany)

The MEMICS tradition of best paper awards continues also in the year 2016. The best contributed paper, presentation, and poster will be selected during the workshop, taking into account their scientific and technical contribution together with the quality of presentation.

The successful organization of MEMICS 2016 would not have been possible without generous help and support from the organizing institutions: Masaryk University, Brno University of Technology, and Charles University.

We thank the Program Committee members and the external reviewers for their careful and constructive work. We thank the Organizing Committee members who regularly help to create a unique and relaxed atmosphere that distinguishes MEMICS from other computer science meetings. We also gratefully acknowledge the support of the EasyChair system and the great cooperation with the EPTCS.

Last, but not least, we would like to thank to our sponsors:

– Lexical Computing
– Y Soft Corporation
– RedHat, Inc.

October 2016

Jan Kofroň
Jan Strejček
Lukáš Holík

Organisation

General Chair

Jan Bouda, Masaryk University, Czech Republic

Program Committee Chairs

Lukáš Holík, Brno University of Technology, Czech Republic
Jan Kofroň, Charles University, Czech Republic
Jan Strejček, Masaryk University, Czech Republic

Program Committee

Jiří Barnat, Masaryk University, Czech Republic
Ezio Bartocci, Vienna University of Technology, Austria
Armin Biere, Johannes Kepler University, Austria
Luca Bortolussi, University of Trieste, Italy
Jan Bouda, Masaryk University, Czech Republic
Martina Daňková, University of Ostrava, Czech Republic
Frédéric Dupuis, Masaryk University, Czech Republic
Piotr Gawron, Polish Academy of Sciences, Poland
Dana Hliněná, Brno University of Technology, Czech Republic
Antti Hyvärinen, University of Lugano, Switzerland
Jan Křetínský, Technical University of Munich, Germany
Martin Kalina, Slovak University of Technology in Bratislava, Slovakia
Stanislav Krajčí, Pavol Jozef Šafárik University in Košice, Slovakia
Erwin Laure, PDC KTH Stockholm, Sweden
Václav Matyáš, Masaryk University, Czech Republic
Luděk Matyska, Masaryk University, Czech Republic
Roland Meyer, University of Kaiserslautern, Germany
Miguel Navascués, Austrian Academy of Sciences, Austria
Marcin Pawlowski, University of Gdansk, Poland
Igor Peterlík, Inria Nancy – Grand Est, France
Philipp Rümmer, Uppsala university, Sweden
David Šafránek, Masaryk University, Czech Republic
Peter Vojtáš, Charles University, Czech Republic
Vít Vondrák, VŠB TUO, IT4Innovations, Czech Republic
Mário Ziman, Slovak Academy of Sciences, Slovakia

Steering Committee Chair

Tomáš Vojnar, Brno University of Technology, Czech Republic

Organizing Committee

Jan Sebastian Novotný, chair, Masaryk University, Czech Republic
Alena Janebová, Masaryk University, Czech Republic
Sylva Kočí, Masaryk University, Czech Republic

Additional Reviewers

Petr Hliněný
Daniel Kouřil
Jan Obdržálek

Invited Lectures—Abstracts

Bit-Blasting Considered Harmful

Armin Biere, Johannes Kepler University

Bit-precise reasoning is essential for many automated reasoning tasks and usually relies on SMT solvers for the theory of fixed-width bit-vectors. After applying simplification through rewriting these solvers usually rely on bit-blasting. The resulting propositional problem is then handed to SAT solvers. On the theoretical side we discuss the complexity of decision problems for bit-vectors. The main argument is that bit-blasting is actually exponential. We discuss subclasses where the exponential explosion is not immediate. On the practical side we show how to use local search for bit-vectors, which avoids bit-blasting. This approach is particularly effective on hard satisfiable bit-vector problems.

The machine learning way to formal verification

Luca Bortolussi, University of Trieste

Many current scientific and technological challenges are related to understanding, design and control of complex systems, from epidemic spreading to performance of computer systems, from biological networks to bike and car sharing. Our interest is to study their emergent properties, starting from a stochastic Markov model, with the machinery of formal verification. More precisely, the focus is on the formalisation of such properties in a suitable logical language and on the automatic verification of these properties in a given model (the so called model checking problem), which in a probabilistic setting takes the form of computing the probability with which such properties are satisfied. However, the models we can construct are always uncertain, in particular in the value of their parameters, due to lack of information and their phenomenological nature. To deal consistently with such an uncertainty, we have combined formal verification techniques with state-of-the-art Machine Learning statistical tools, based on Gaussian Processes.

We will show how to efficiently compute the satisfaction probability of a behavioural property when model parameters are unknown, but assumed to lie in a bounded interval, a method we called Smoothed Model Checking. Combining these ideas with Bayesian Optimisation, we can then efficiently solve system design problems, i.e. fixing controllable model parameters so that the system robustly exhibits a desired behaviour. This talk will be a gentle introduction to these ideas.

Summaries for Context-Free Games

Roland Meyer, TU Kaiserslautern

(based on joint work with Lukas Holik and Sebastian Muskalla) The motivation of our work is to generalize the language-theoretic approach from verification to synthesis. Central to language-theoretic verification are queries $L(G) \subseteq L(A)$, where G is a context-free grammar representing the flow of control in a recursive program and A is a finite automaton representing (iterative refinements of) the specification. When moving to synthesis, we replace the inclusion query by a strategy synthesis for an inclusion game. This means G comes with an ownership partitioning of the non-terminals. It induces a game arena defined by the sentential forms and the left-derivation relation. The winning condition is inclusion in the regular language given by A.

For the verification of recursive programs, the two major paradigms are summarization and saturation. Procedure summaries compute the effect of a procedure in the form of an input-output relation. Saturation techniques compute the pre*-image over the configurations of a pushdown system (including the stack). Both were extensively studied, optimized, and implemented. What speaks for summaries is that they seem to be used more often, as witnessed by the vast majority of verification tools participating in the software verification competition. The reason, besides simpler implementability, may be that the stack present in pre* increases the search space.

Saturation has been lifted to games and synthesis. We fill in the empty spot and propose the first summary-based solver and synthesis method for context-free inclusion games. Our algorithm is based on a novel representation of all plays starting in a non-terminal. The representation uses the domain of Boolean formulas over the transition monoid of the target automaton. The elements of the transition monoid are essentially procedure summaries. We show that our algorithm has optimal (doubly exponential) time complexity, that it is compatible with recent antichain optimizations, and that it admits a lazy evaluation strategy. Our preliminary experiments show encouraging results, indicating a speed up of three orders of magnitude over a competitor.

Table of Contents

II Presentation Abstracts

Part I

Regular Papers

Reducing Nondeterministic Tree Automata by Adding Transitions

Ricardo Manuel de Oliveira Almeida

University of Edinburgh, United Kingdom

Abstract. We introduce saturation of nondeterministic tree automata, a technique that adds new transitions to an automaton while preserving its language. We implemented our algorithm on `minotaut` - a module of the tree automata library `libvata` that reduces the size of automata by merging states and removing superfluous transitions - and we show how saturation can make subsequent merge and transition-removal operations more effective. Thus we obtain a Ptime algorithm that reduces the size of tree automata even more than before. Additionally, we explore how `minotaut` alone can play an important role when performing hard operations like complementation, allowing to both obtain smaller complement automata and lower computation times. We then show how saturation can extend this contribution even further. We tested our algorithms on a large collection of automata from applications of `libvata` in shape analysis, and on different classes of randomly generated automata.

1 Introduction

Tree automata are a generalization of word automata to non-linear words (i.e., trees) [10]. They have many applications in model checking [3,7], term rewriting [11] and related areas of formal software verification, e.g., shape analysis [13]. Several software packages for manipulating tree automata have been developed, e.g., Timbuk [4], Autowrite [11] and `libvata` [16] (on which other verification tools, like Forester [17], are based).

For nondeterministic automata, many questions about their languages are computationally hard. The language universality, equivalence and inclusion problems are `PSPACE`-complete for word automata and `EXPTIME`-complete for tree automata [10]. A common approach to solving many instances of the inclusion problem is via the computation of different notions of simulation preorders that at the same time under-approximate language inclusion and are computable in polynomial time [12,1]. These simulation preorders thus offer a trade-off between computability and expressiveness. Efficient reduction algorithms have been presented both for word automata [8] and for tree automata [2,6], where language inclusion is witnessed by the membership of a pair of states in a simulation preorder. In our paper, we focus on Heavy(x,y) [6], a polynomial-time algorithm for reducing tree automata, in the sense of obtaining a smaller automaton with the same language, though not necessarily with the absolute minimal number of states possible (in general, as with word automata, there is no unique non-deterministic automaton with the minimal possible number of states for a given language). Heavy(x,y) is based on an intricate combination of transition pruning and state quotienting techniques for tree automata, extending previous work on the words case [8]. Transition pruning is based on the notion that certain transitions may be removed from the automaton because 'better' ones remain. The notion of 'better' is given by comparing the states at the endpoints of the two transitions w.r.t. suitable simulation preorders. The Heavy(x,y) algorithm yields substantially smaller and sparser (i.e., using fewer transitions per state and per symbol) automata than all previously known reduction techniques, and it is still fast enough to handle large instances.

We start by optimizing the computation of simulation preorders in Heavy(x,y). This is done by identifying re-computations that can be skipped, which yields generally faster computation times. We then introduce the dual notion of transition pruning, in which transitions are added to the automaton if 'better' ones exist already. This technique is known as transition saturation and it was previously defined for word automata [9]. As in transition pruning, this technique compares the source states of the two transitions w.r.t. a simulation R_s on the states space, and the target states of the transitions w.r.t. a simulation R_t. If saturating an automaton with R_s and R_t preserves the language, we say that $S(R_s, R_t)$ is good for saturation. We provide a summary of all $S(R_s, R_t)$ we found to be or not to be good for saturation.

The motivation behind saturation is that it may allow for new merging of states and transition removal which were not possible by using Heavy alone. Thus saturating an automaton which has been reduced with Heavy(x,y) and then reducing it again might result in an even smaller automaton. We perform an experimental evaluation to measure how much smaller, on average, automata become by interleaving reduction methods with transition saturation. Our results indicate that generally one obtains automata with fewer states, but on some cases with more transitions, than the ones obtained by Heavy(x,y) alone.

In general, one wishes to reduce automata in order to make them more efficient to handle in subsequent computations. Thus, we present a second experimental evaluation showing that the complement automata are much smaller and faster to compute when the automata have previously been reduced with the techniques described above.

We implemented our algorithm as an extension of `minotaut` (source code available [5]), a module of the tree automata library `libvata` [16] where the Heavy algorithm is provided. The experiments described above were performed on a large collection of automata from applications of `libvata` in shape analysis, as well as on different classes of randomly generated tree automata.

2 Preliminaries

Trees and tree automata. A *ranked alphabet* Σ is a set of symbols together with a function $\# : \Sigma \to \mathbb{N}_0$. For $\sigma \in \Sigma$, $\#(\sigma)$ is called the *rank* of a. We define a *node* as a sequence in $\mathbb{N}^*$. For a node $v \in \mathbb{N}^*$, we define the i-th child of v to be the node vi, for some $i \in \mathbb{N}$.

Given a ranked alphabet Σ, a finite *tree* over Σ is defined as a partial mapping $t : \mathbb{N}^* \to \Sigma$ such that for all $v \in \mathbb{N}^*$ and $i \in \mathbb{N}$, if $vi \in dom(t)$ then **(1)** $v \in dom(t)$, and **(2)** $\#(t(v)) \geq i$. Note that the number of children of a node v may be smaller than $\#(t(v))$. In this case we say that the node is *open*. Nodes which have exactly $\#(t(v))$ children are called *closed*. Nodes which do not have any children are called *leaves*. A tree is closed if all its nodes are closed, otherwise it is open. By $\mathbb{C}(\Sigma)$ we denote the set of all closed trees over Σ and by $\mathbb{T}(\Sigma)$ the set of all trees over Σ.

A finite nondeterministic *top-down tree automaton* (TDTA) is a quadruple $A = (\Sigma, Q, \delta, I)$ where Q is a finite set of states, $I \subseteq Q$ is a set of initial states, Σ is a ranked alphabet, and $\delta \subseteq Q \times \Sigma \times Q^+$ is the transition relation. A TDTA has an unique final state, which we represent by ψ. The transition rules satisfy that if $\langle q, \sigma, \psi \rangle \in \delta$ then $\#(\sigma) = 0$, and if $\langle q, \sigma, q_1 \dots q_n \rangle \in \delta$ (with $n > 0$) then $\#(\sigma) = n$. Informally, a run of A reads an input tree top-down from the root, branching into sub-runs on subtrees as specified by the applied transition rules, and it accepts it if every branch ends in ψ. Formally, a run of A over a tree $t \in \mathbb{T}(\Sigma)$ (or a t-run in A) is a partial mapping $\pi : \mathbb{N}^* \to Q$ such that $v \in dom(\pi)$ iff either $v \in dom(t)$ or $v = v'i$ where $v' \in dom(t)$ and $i \leq \#(t(v'))$. Further, for every $v \in dom(t)$, there exists either **a)** a rule $\langle q, a, \psi \rangle$ such that $q = \pi(v)$ and $\sigma = t(v)$, or **b)** a rule $\langle q, \sigma, q_1 \dots q_n \rangle$ such that $q = \pi(v)$, $\sigma = t(v)$, and $q_i = \pi(vi)$ for each $i : 1 \leq i \leq \#(\sigma)$. A *leaf of a run* π on t is a node $v \in dom(\pi)$ such that $vi \in dom(\pi)$ for no $i \in \mathbb{N}$.

We write $t \overset{\pi}{\Longrightarrow} q$ to denote that π is a t-run of A such that $\pi(\epsilon) = q$. A run π is accepting if $t \overset{\pi}{\Longrightarrow} q \in I$. The *downward language of a state* q in A is defined by $D_A(q) = \{t \in \mathbb{C}(\Sigma) \mid t \overset{\pi}{\Longrightarrow} q, \text{for some run } \pi\}$, while the *language* of A is defined by $L(A) = \bigcup_{q \in I} D_A(q)$. We sometimes write simply A to refer to its language.

Downward and upward relations. The behaviour of states in TDTA can be compared by semantic preorders (and their induced equivalences), based on the upward- or downward behaviour of the automaton from these states.

Ordinary downward simulation on tree automata can be characterized by a game between two players, Spoiler and Duplicator. Given a pair of states (q, r), Spoiler wants to show that (q, r) is not contained in the simulation preorder relation, while Duplicator has the opposite goal. Starting in the initial configuration (q, r), Spoiler chooses a transition $q \xrightarrow{\sigma} \langle q_1 \dots q_n \rangle$, where $n = \#(\sigma)$, and Duplicator must imitate it *stepwise* by choosing a transition with the same symbol $r \xrightarrow{\sigma} \langle r_1 \dots r_n \rangle$. This yields n new configurations $(q_1, r_1), \dots, (q_n, r_n)$ from which the game continues independently. If a player ever cannot make a move then the other player wins. Duplicator wins every infinite game. Simulation holds iff Duplicator wins.

A tree branches as one goes downward, but 'joins in' side branches as one goes upward. Therefore a comparison of the upward behaviour of states depends also on the joining side branches as one goes upward in the tree. Thus upward simulation is only defined *relative* to a given other relation R that compares the downward behaviour of states 'joining in' from the sides [1]. One speaks, e.g., of upward simulation *of* R. Thus in the ordinary upward simulation game, starting in the initial configuration (q, r), Spoiler chooses a transition $q' \xrightarrow{\sigma} \langle q_1 \dots q_n \rangle$, where $q = q_i$ for some i and $n = \#(\sigma)$, and Duplicator must imitate it *stepwise* by choosing a transition with the same symbol $r' \xrightarrow{\sigma} \langle r_1 \dots r_n \rangle$, where $r = r_i$, and such that 1) $q_j R r_j$, for every $j \neq i$, and 2) $q \in I \implies r \in I$. The game continues from the configuration (q', r'), and Spoiler wins if Duplicator ever cannot respond to a move, otherwise Duplicator wins.

While in ordinary downward simulation (resp., upward simulation w.r.t. R) Duplicator only knows Spoiler's very next step, in downward k-lookahead simulation (resp., upward k-lookahead simulation w.r.t. R) Duplicator knows Spoiler's next k steps in advance (unless Spoiler's move ends in a deadlocked state - i.e., a state with no transitions). In the case where Duplicator knows *all* steps of Spoiler in the entire downward simulation game in advance (i.e., $k = \infty$), we talk of downward trace/language inclusion (resp., upward trace inclusion w.r.t. R). As the parameter k increases, the k-lookahead simulation relation becomes larger and thus approximates the respective trace inclusion relation better and better.

The downward/upward k-lookahead simulation preorder (denoted $\preceq^{k\text{-dw}}/\preceq^{k\text{-up}}(R)$, or just $\sqsubseteq^{\mathsf{dw}}/\sqsubseteq^{\mathsf{up}}(R)$ in the ordinary case) is the set of all pairs (p, q) for which Duplicator has a winning strategy in the respective game. For the downward/upward trace inclusion preorder we write $\subseteq^{\mathsf{dw}}/\subseteq^{\mathsf{up}}(R)$.

Downward/upward k-lookahead simulation is PTIME-computable for every fixed k and a good under-approximation of the respective trace inclusion (which is EXPTIME-complete in the downward case [10], and PSPACE-complete for $R = id$ in the upward case).

Transition pruning and state quotienting. Given a TDTA $A = (\Sigma, Q, \delta, I)$, certain transitions may be pruned without changing the language, because 'better' ones remain. Given a strict partial order $P \subseteq \delta \times \delta$ on the set of transitions, the pruned automaton is defined as $Prune(A, P) = (\Sigma, Q, \delta', I)$ where $\delta' = \{(p, \sigma, r) \in \delta \mid \nexists (p', \sigma, r') \in \delta.\, (p, \sigma, r)\, P\, (p', \sigma, r')\}$. I.e., if $t\, P\, t'$ then t may be pruned because t' is 'better' than t. $Prune(A, P)$ is unique and transitions are removed in parallel without re-computing P. Trivially, $L(Prune(A, P)) \subseteq L(A)$. If $L(Prune(A, P)) = L(A)$ also holds we say that P is *good for pruning* (GFP).

We obtain GFP relations by comparing the endpoints of transitions over the same symbol $\sigma \in \Sigma$. Given two binary relations R_{u} and R_{d} on Q, we define $P(R_{\mathsf{u}}, R_{\mathsf{d}}) = \{(\langle p, \sigma, r_1 \cdots r_n \rangle, \langle p', \sigma, r'_1 \cdots r'_n \rangle) \mid p\ R_{\mathsf{u}}\ p' \text{ and } (r_1 \cdots r_n)\ \hat{R}_{\mathsf{d}}\ (r'_1 \cdots r'_n)\}$, where $\hat{R}_{\mathsf{d}}$ is a suitable lifting of $R_{\mathsf{d}} \subseteq Q \times Q$ to $\hat{R}_{\mathsf{d}} \subseteq Q^n \times Q^n$: if R_{d} is some strict partial order $<_{\mathsf{d}}$, then $\hat{R}_{\mathsf{d}}$ is a binary relation $\hat{<}_{\mathsf{d}}$ s.t. 1) $\forall_{1 \leq i \leq n}.\, r_i \leq_{\mathsf{d}} r'_i$, and 2) $\exists_{1 \leq i \leq n}.\, r_i <_{\mathsf{d}} r'_i$; if R_{d} is a non-strict partial order $\leq_{\mathsf{d}}$, then only condition 1) applies. The relations $R_{\mathsf{u}}, R_{\mathsf{d}}$ are chosen such that $P(R_{\mathsf{u}}, R_{\mathsf{d}}) \subseteq \delta \times \delta$ is a strict partial order (i.e., of the two relations R_{u} and R_{d}, one must be a strict partial order) that is GFP; see the algorithm Heavy below.

Another method for reducing the size of automata is state quotienting. Given a suitable equivalence on the set of states, each equivalence class is collapsed into just one state. A preorder $\sqsubseteq$ induces an equivalence relation $\equiv := \sqsubseteq \cap \sqsupseteq$. Given $q \in Q$, $[q]$ denotes its equivalence class w.r.t. $\equiv$. For $P \subseteq Q$, $[P]$ denotes the set of equivalence classes $[P] = \{[p] \mid p \in P\}$. The quotient automaton is defined as $A/\equiv := (\Sigma, [Q], \delta_{A/\equiv}, [I])$, where $\delta_{A/\equiv} = \{\langle [q], \sigma, [q_1] \dots [q_n] \rangle \mid \langle q, \sigma, q_1 \dots q_n \rangle \in \delta_A\}$. Trivially, $L(A) \subseteq L(A/\equiv)$. If $L(A) = L(A/\equiv)$ also holds, $\equiv$ is said to be *good for quotienting* (GFQ).

The Heavy algorithm. Here we describe Heavy(x,y) [6], a tree automata reduction algorithm based on transition pruning and state quotienting. The parameters $x, y \geq 1$ describe the lookahead for the used downward/upward lookahead simulations, respectively, where larger values yield better reduction but are harder to compute. The algorithm is polynomial for fixed x, y, and doubly exponential in x (due to the downward branching of the tree) and single exponential in y otherwise. Let $Op(x, y)$ be the following sequence of operations on tree automata, where RU stands for removing useless states (i.e., states that cannot be reached from any initial state or from which no tree can be accepted): RU, quotienting with $\preceq^{x\text{-dw}}$, pruning with $P(id, \prec^{x\text{-dw}})$, RU, quotienting with $\preceq^{y\text{-up}}(id)$, pruning with $P(\prec^{y\text{-up}}(id), id)$, pruning with $P(\sqsubset^{\mathsf{up}}(id), \preceq^{x\text{-dw}})$, RU, quotienting with $\preceq^{y\text{-up}}(id)$, pruning with $P(\preceq^{y\text{-up}}(\sqsubseteq^{\mathsf{dw}}), \sqsubset^{\mathsf{dw}})$, RU. These operations are language preserving, since the used relations are GFP/GFQ [6].

The algorithm Heavy(1,1) just iterates $Op(1, 1)$ until a fixpoint is reached. The general algorithm Heavy(x,y) does not iterate $Op(x, y)$, but uses a double loop: it iterates the sequence Heavy(1,1) $Op(x, y)$ until a fixpoint is reached.

The Heavy algorithm is provided in the `minotaut` library [5], making use of `libvata`'s efficient computation of ordinary simulation (for a description of `minotaut`'s implementation of simulation with larger lookaheads see Section 3). Heavy behaves well in practice, significantly reducing both automata of program verification provenience and randomly generated automata [6].

3 Efficient Computation of Lookahead Simulations

We performed some optimizations on the computation of the maximal downward lookahead simulation used in Heavy(x,y). In the following we describe the key aspects of the computation in terms of a game between Spoiler and Duplicator. (Upward simulation is similar but simpler, since the tree branches downward.)

Fixpoint iteration with incremental moves. We represent binary relations over Q as boolean matrices of dimension $|Q| \times |Q|$. Starting with a matrix W in which all entries are set to TRUE, the algorithm consists of a downward refinement loop of W that converges to the maximal downward k-lookahead simulation. In each iteration of the refinement loop, for each pair p, q where $W[p][q]$ is still TRUE:

– Spoiler tries an attack *atk* consisting of a possible move from p of some depth $d \leq k$. Each such attack is built incrementally, for $d = 1, 2, \dots, k$, in order to give Duplicator a chance to respond already to a prefix of *atk* of depth $< k$.

- Duplicator then attempts to defend against the given attack of depth d, by finding a matching move *def* from q by the same symbols s.t. every leaf-state in *def* is in relation W with the corresponding state in *atk*. (Duplicator's search is done in depth-first mode.) If successful, Duplicator declares victory against this particular (prefix of an) attack and Spoiler tries a new one, since extending the current one to a higher depth is pointless. If unsuccessful and $d < k$, Spoiler builds an attack of the next depth level $d+1$, by extending *atk* with one new transition from each of its leaf-states. The extra information might enable Duplicator to find a successful defence then.
- Duplicator fails if he could not defend against an attack *atk* of the maximal depth, either where *atk* has depth $d = k$ or $d < k$ but *atk* cannot be extended any more due to all its leaf-states having no outgoing transitions.
- If Duplicator could defend against every attack (or some prefix of it) by Spoiler then $W[p][q]$ stays true, for now.
- In the worst case, for each Spoiler's attack of depth d, Duplicator must search through all defences of depth up-to d, but often Duplicator wins sooner.
- Similarly, in the worst case, Spoiler needs to try all possible attacks of depth k, but often Duplicator already wins against prefixes of some depth $d < k$.

Since the outcome of a local game depends on the values of W, the refinement loop might converge only after several iterations. The reached fixpoint represents a relation that is generally not transitive (for $k > 1$), but its transitive closure is the required maximal downward k-lookahead simulation preorder $\preceq^{k\text{-dw}}$.

An Optimization Based on Pre-Refinement. Following an approach implemented in `Rabit` [15] for word automata, we under-approximate non-simulation as follows. If there exists a tree of bounded depth d that can be read from state p but not from state q, then the pair (p, q) cannot be in k-lookahead simulation for any k. The pre-refinement step iterates through all pairs (p, q) and sets $W[p][q]$ to `false` if such a tree is found witnessing non-simulation. Our experiments show that, for most automata samples, running a pre-refinement with some modest depth d suffices to speed up the k-lookahead downward simulation computation.

We now present an optimization that allows to compute lookahead simulation faster. The idea is that attacks which are *good* (i.e., successful) or *bad* (i.e., unsuccessful) may be remembered to skip unnecessary re-computations.

Semi-global caching of Spoiler's attacks. An attack is seen as *good* or *bad* within the scope of the *whole game*. Consider the game configuration (p_1, q_1) in Figure 1. Although q_1 can read all trees of depth 3 that p_1 can read, there are *good* attacks from p_2 both against q_2 and against q_3. Duplicator will find and store these if, when defending against the attack $ac(e, e)$, he first tries the transition to q_2 (which can only read d), or when defending against $ad(e, e)$ he first tries the transition to q_3 (which can only read c). After trying possibly all attempts, Duplicator is able to defend against the attack and Spoiler now tries the b-transition from p_1 to p_2. However, all possible sub-attacks are now the same, which makes Duplicator announce defeat on them immediately without any exploration.

In Appendix B two different ways of performing this caching of Spoiler's attacks can be found. The three versions present a trade-off between expressiveness and space required to encode attacks. Our tests indicate that the semi-global version indeed speeds up the computation on automata with high transition overlaps (i.e., where many states are shared by different transitions).

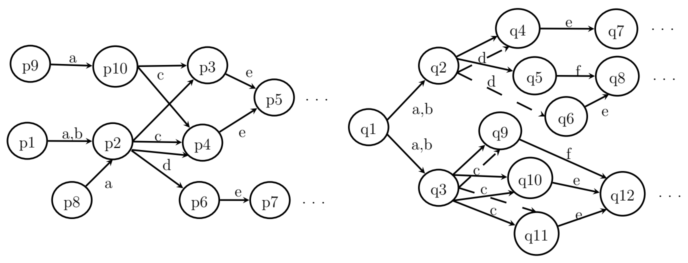


Fig. 1: For $W = \{(p_5, q_7), (p_5, q_8), (p_7, q_8), (p_7, q_{12})\}$, all versions of the optimization allow some attacks to be skipped when computing the 3-lookahead downward simulation.

4 Saturation of Tree Automata

In Section 2, we described the transition pruning technique, which removes a transition if a 'better' one remains. In this section, we introduce its dual notion, saturation, which adds a transition if a 'better' one exists already. The motivation behind saturation is to pave the way for further reductions when the Heavy algorithm has reached a fixpoint on the automaton (see Section 5). Saturation has been defined for the words case before [9], here we apply it to tree automata.

Definition 1. *Let $A = (\Sigma, Q, \delta, I)$ be a TDTA, $\Delta = Q \times \Sigma \times Q^+$ and $S \subseteq \Delta \times \Delta$ a reflexive binary relation on Δ. The* S-saturated automaton *is defined as $Sat(A, S) := (\Sigma, Q, \delta_S, I)$, where*

$$\delta_S = \{\langle p', a, q'_1 \dots q'_{\#(a)}\rangle \in \Delta \mid \exists \langle p, a, q_1 \dots q_{\#(a)}\rangle \in \delta \cdot \langle p', a, q'_1 \dots q'_{\#(a)}\rangle \; S \; \langle p, a, q_1 \dots q_{\#(a)}\rangle\}.$$

Since S is reflexive, any transition in the initial automaton is preserved and so $A \subseteq Sat(A, S)$. When the converse inclusion also holds, we say that S is *good for saturation* (GFS). Note that the GFS property is downward closed in the space of reflexive relations, i.e., if R is GFS and $id \subseteq R' \subseteq R$, then R' too is GFS. (or if R' is not GFS, then R too is not GFS).

Given two binary relations R_s and R_t on Q, we define
$S(R_s, R_t) = \{(\langle p, \sigma, r_1 \cdots r_n\rangle, \langle p', \sigma, r'_1 \cdots r'_n\rangle) \mid pR_s p' \text{ and } (r_1 \cdots r_n)\hat{R}_t(r'_1 \cdots r'_n)\}$,

where $\hat{R}_t$ is the standard lifting of $R_t \subseteq Q \times Q$ to $\hat{R}_t \subseteq Q^n \times Q^n$. Informally, a transition t' is added to the automaton if there exists already a transition t s.t. its source state is R_s-larger than the source state of t', and its target states are $\hat{R}_t$-larger than the target states of t'. Theorem 1 below proves that $S(\supseteq^{\mathsf{dw}}, \subseteq^{\mathsf{dw}})$ is GFS. Since the GFS property is downward closed, it follows that $S(\supseteq^{\mathsf{dw}}, \sqsubseteq^{\mathsf{dw}})$, $S(\supseteq^{\mathsf{dw}}, id)$, $S(\sqsupseteq^{\mathsf{dw}}, \subseteq^{\mathsf{dw}})$, $S(\sqsupseteq^{\mathsf{dw}}, \sqsupseteq^{\mathsf{dw}})$, $S(\sqsupseteq^{\mathsf{dw}}, id)$, $S(id, \subseteq^{\mathsf{dw}})$ and $S(id, \sqsubseteq^{\mathsf{dw}})$ too are GFS. In Theorem 2 (see Appendix A for a proof), we prove that $S(\subseteq^{\mathsf{up}}(id), \supseteq^{\mathsf{up}}(id))$ is GFS. Thus it follows that $S(\subseteq^{\mathsf{up}}(id), \sqsupseteq^{\mathsf{up}}(id))$, $S(\subseteq^{\mathsf{up}}(id), id)$, $S(\sqsubseteq^{\mathsf{up}}(id), \supseteq^{\mathsf{up}}(id))$, $S(\sqsubseteq^{\mathsf{up}}(id), \sqsupseteq^{\mathsf{up}}(id))$, $S(\sqsubseteq^{\mathsf{up}}(id), id)$, $S(id, \supseteq^{\mathsf{up}}(id))$ and $S(id, \sqsupseteq^{\mathsf{up}}(id))$ too are GFS.

Theorem 1. *$S(\supseteq^{\mathsf{dw}}, \subseteq^{\mathsf{dw}})$ is GFS.*

Proof. Let A be a TDTA and $A_S = Sat(A, S(\supseteq^{\mathsf{dw}}, \subseteq^{\mathsf{dw}}))$. We will use induction on $n \geq 1$ to show that for every tree t of height n and every run π_S of A_S s.t. $t \xRightarrow{\pi_S} p$, for some state p, there exists a run π of A s.t. $t \xRightarrow{\pi} p$. This shows, in particular, that $A_S \subseteq A$.

In the base case $n = 1$, t is a leaf-node σ, for some $\sigma \in \Sigma$. Thus for every run π_S of A_S such that $t \xRightarrow{\pi_S} p$, for some state p, there exists $\langle p, \sigma, \psi \rangle \in \delta_S$. By the definition of δ_S, there exists $\langle q, \sigma, \psi \rangle \in \delta$ s.t. $q \subseteq^{\mathsf{dw}} p$. Consequently, there exists a run π in A s.t. $t \xRightarrow{\pi} q$. By $q \subseteq^{\mathsf{dw}} p$, there also exists a run π' of A s.t. $t \xRightarrow{\pi'} p$.

For the induction step, let t be a tree of height $n > 1$ and a its root symbol. Thus for every run π_S of A_S s.t. $t \xRightarrow{\pi_S} p$, for some state p, there exist $\langle p, a, q_1 \ldots q_{\#(a)} \rangle \in \delta_S$ and, for each $i : (1 \leq i \leq \#(a))$, a run π_{S_i} of A_S s.t. $t_i \xRightarrow{\pi_{S_i}} q_i$. By the definition of δ_S, there exists $\langle p', a, q'_1 \ldots q'_{\#(a)} \rangle \in \delta$ s.t. $p' \subseteq^{\mathsf{dw}} p$ and, for every $i : (1 \leq i \leq \#(a))$, $q'_i \supseteq^{\mathsf{dw}} q_i$. Applying the induction hypothesis to each of the subtrees t_i, we know that for every t_i-run π_{S_i} of A_S ending in q_i there is also a t_i-run π_i of A ending in q_i. And since $q'_i \supseteq^{\mathsf{dw}} q_i$ for every $i : (1 \leq i \leq \#(a))$, for each t_i there exists a run π'_i of A s.t. $t_i \xRightarrow{\pi'_i} q'_i$. Since there exists $\langle p', a, q'_1 \ldots q'_{\#(a)} \rangle \in \delta$, we obtain that there is a run π'' of A s.t. $t \xRightarrow{\pi''} p'$. From $p' \subseteq^{\mathsf{dw}} p$, it follows that there is also a run π''' of A s.t. $t \xRightarrow{\pi'''} p$.

Theorem 2. *$S(\subseteq^{\mathsf{up}}(id), \supseteq^{\mathsf{up}}(id))$ is GFS.*

The counterexample in Fig. 2 shows that $S(\equiv^{\mathsf{dw}}, \equiv^{\mathsf{up}}(R))$ is not GFS for any relation $R \subseteq Q \times Q$. The remaining counterexamples can be found in Appendix A:

- Figure 8 shows that $S(id, \equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}}))$ is not GFS.
- Figure 9 shows that $S(\equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}}), id)$ is not GFS.
- Figure 10 is inspired by an example for a similar result for linear trees (i.e., words) [9]. It shows that $S(\equiv^{\mathsf{up}}(R), \equiv^{\mathsf{dw}})$ is not GFS for any relation $R \subseteq Q \times Q$.

In Figure 3 we present a table that summarizes these results. The negative results follow from the counterexamples given and the fact that the GFS property is downward closed.

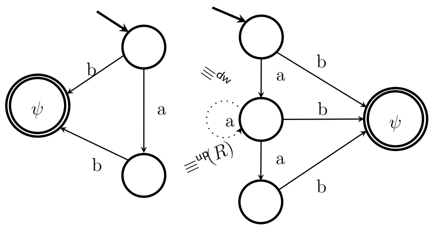


Fig. 2: $S(\equiv^{\mathsf{dw}}, \equiv^{\mathsf{up}}(R))$ is not GFS for any relation $R \subseteq Q \times Q$: if we add the dotted transition, the linear tree *aaab* is now accepted. The symbol b has rank 0 and a rank 1.

S	id	$\sqsubseteq^{\mathsf{dw}}$	$\subseteq^{\mathsf{dw}}$	$\sqsupseteq^{\mathsf{up}}(id)$	$\supseteq^{\mathsf{up}}(id)$	$\sqsupseteq^{\mathsf{up}}(\sqsupseteq^{\mathsf{dw}})$	$\supseteq^{\mathsf{up}}(\supseteq^{\mathsf{dw}})$
id	✓	✓	✓	✓	✓	×	×
$\sqsupseteq^{\mathsf{dw}}$	✓	✓	✓	×	×	×	×
$\supseteq^{\mathsf{dw}}$	✓	✓	✓	×	×	×	×
$\sqsubseteq^{\mathsf{up}}(id)$	✓	×	×	✓	✓	×	×
$\subseteq^{\mathsf{up}}(id)$	✓	×	×	✓	✓	×	×
$\sqsubseteq^{\mathsf{up}}(\sqsubseteq^{\mathsf{dw}})$	×	×	×	×	×	×	×
$\subseteq^{\mathsf{up}}(\subseteq^{\mathsf{dw}})$	×	×	×	×	×	×	×

Fig. 3: GFS relations for tree automata. Relations which are GFS are marked with ✓ and those which are not are marked with ×.

5 Experimental Results

As we saw in Section 2, the automaton computed by Heavy corresponds to the local minimum of the sequence of reduction techniques used, i.e., no smaller automaton can be reached by applying that same sequence of steps again. The motivation behind saturation is to change this scenario, since modifying an automaton while preserving its language may leave it in a state where a different local minimum is reachable by applying Heavy again. Since saturation adds transitions, in the end an automaton will either have 1) the same number of states and the same or larger number of transitions, 2) the same number of states but fewer transitions, or 3) fewer states. We say that scenarios 2) and 3) correspond to an automaton 'better' than the initial one, and scenario 1) to a 'worse' one.

Our experiments on test automata consisted of first reducing them with Heavy and then alternating between saturation and reduction successively until either a fixpoint is reached or the automata becomes 'worse'. Just like in the case of Heavy, there is no ideal order to apply the saturation/reduction techniques, so we tested multiple possibilities, from which we highlight two versions, Sat1(x,y) and Sat2(x,y), where $x, y \geq 1$ are the lookaheads used for computing k-downward and k-upward simulations, respectively (see Figure 4). In both Sat1 and Sat2, we chose an order for the operations that ensures that the effect of the saturations is not necessarily cancelled by the reductions immediately after. Intuitively, Sat1 starts by applying both saturations together, in an attempt to obtain a highly dense automaton where more states may be quotiented. Sat2, on the other hand, prevents the automaton from becoming too dense, by interleaving each downward saturation with the upward reductions it may allow. Moreover, each upward reduction not only may allow for new downward saturations to be performed, but it may also have its effect cancelled if the upward saturation is performed immediately after. Thus, in Sat2 downward saturation and upward reductions are iterated in an inner loop before performing any upward saturation. Both versions return the 'best' automaton ever encountered.

```
Sat1(x,y)
Loop:
  Sat. w/ $S(\succeq^{x\text{-dw}}, \preceq^{x\text{-dw}})$
  Sat. w/ $S(\preceq^{y\text{-up}}(id), \succeq^{y\text{-up}}(id))$
  Quot. w/ $\preceq^{y\text{-up}}(id)$
  Prune w/ $P(\sqsubset^{\text{up}}(id), \preceq^{x\text{-dw}})$
  Quot. w/ $\preceq^{y\text{-up}}(id)$
  Prune w/ $P(\preceq^{y\text{-up}}(\sqsubseteq^{\text{dw}}), \sqsubset^{\text{dw}})$
  Run Heavy(x,y)
```

```
Sat2(x,y)
Loop:
  Loop:
    Sat. w/ $S(\succeq^{x\text{-dw}}, \preceq^{x\text{-dw}})$
    Quot. w/ $\preceq^{y\text{-up}}(id)$
    Prune w/ $P(\prec^{y\text{-up}}(id), id)$
    Prune w/ $P(\sqsubset^{\text{up}}(id), \preceq^{x\text{-dw}})$
    Quot. w/ $\preceq^{y\text{-up}}(id)$
    Prune w/ $P(\preceq^{y\text{-up}}(\sqsubseteq^{\text{dw}}), \sqsubset^{\text{dw}})$
  Sat. w/ $S(\preceq^{y\text{-up}}(id), \succeq^{y\text{-up}}(id))$
  Run Heavy(x,y)
```

Fig. 4: Two saturation-based reduction methods. Both versions return the 'best' automaton ever encountered.

We tested the different saturation-based reduction methods on a set of 14,498 automata (57 states and 266 transitions on avg.) from the shape analysis tool Forester [17]. We can see (Figure 5) that, on average, the two versions produced automata containing *both* fewer states and, especially, fewer transitions than Heavy alone. However, this came at the expense of longer running times.

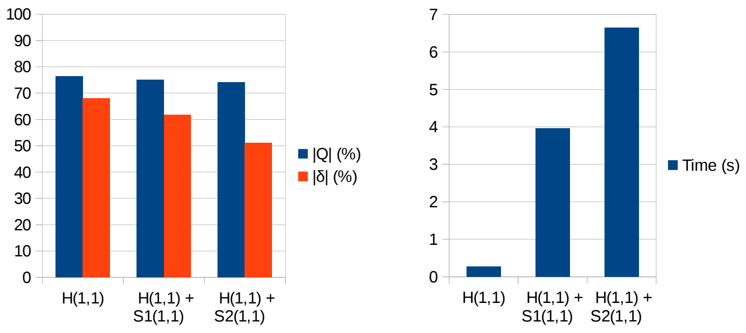
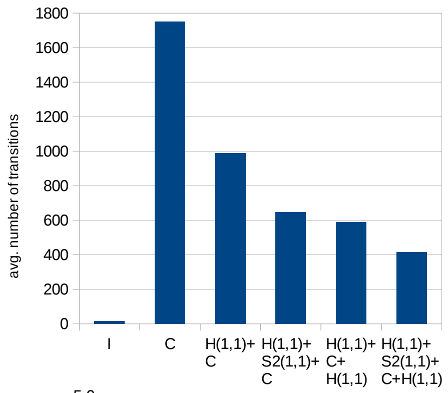


Fig. 5: Reduction of Forester automata using saturation methods. The left chart gives the avg. number of states and transitions that remained (in percentage) after application of each method; the right chart compares their running times. Heavy(1,1) followed by Sat2(1,1) reduced the automata the most, but it was also the slowest method.

The results that follow focus on the advantage of reducing automata when computing their complement (for which we use `libvata`'s implementation of the difference algorithm [14]). We started by testing on a subset of the Forester sample (Fig. 6 and Fig. 11 in App. C), and we compared direct complementation with reducing automata (with Heavy(1,1) optionally followed by Sat2(1,1)) prior to the complementation and with a final reduction using Heavy(1,1). Due to memory reasons, direct complementation was not feasible for large automata. Thus the sample used is the subset of Forester containing all automata with at most 14 states, in a total of 760 automata. As we can see, all reduction methods yielded significantly smaller complement automata than direct complementation, on average, while running either with similar times or substantially faster. This difference was particularly notorious when the automata were first reduced with both Heavy(1,1) and Sat2(1,1), which, compared to direct complementation, resulted in automata with fewer states (18 vs 27, see Figure 11 in App. C) and fewer transitions (649 vs 1750) and at much lower times (0.02s vs 4.86s). Applying Heavy(1,1) in the end reduced the automata even more, with a very low time cost.

The next experiments were performed on sets of randomly generated tree automata, according to a generalization of the Tabakov-Vardi model of random



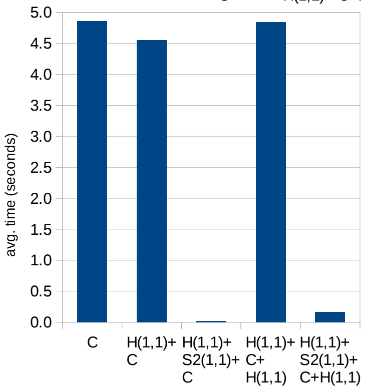


Fig. 6: Reducing and complementing Forester automata with at most 14 states. The complement automata have fewer transitions and are faster to compute if the complementation is preceded by applying Heavy(1,1) and Sat2(1,1) - H(1,1)+S2(1,1)+C - or just Heavy(1,1) - H(1,1)+C. Applying Heavy(1,1) in the end reduces even more. We include the initial number of transitions (I) for comparison purposes.

word automata [18]. Given parameters n, s, td (transition density) and ad (acceptance density), it generates tree automata with n states, s symbols (each of rank 2), $n * td$ randomly assigned transitions for each symbol, and $n * ad$ randomly assigned leaf rules. Figure 7 shows the results of complementing automata with $n = 4$ and varying td. While the automata tested are very small, for some values of td their complements are quite complex (more than 400 transitions on average). As we can see, applying Heavy not only before but also after the complementation on average yielded significantly smaller automata, especially in terms of transitions, while running with similar times to direct complementation (all average times were below 0.1s). Moreover, the saturation method achieved reductions in the states space which were not possible with Heavy alone. This came at the cost of higher running times and also of returning automata with more transitions - but with still far less transitions then those obtained with direct complementation. Note that for very dense automata ($td \geq 4.0$), the average size of the complement became particularly small. This is because more than half of the automata generated with such td were universal, and thus their complements were empty.

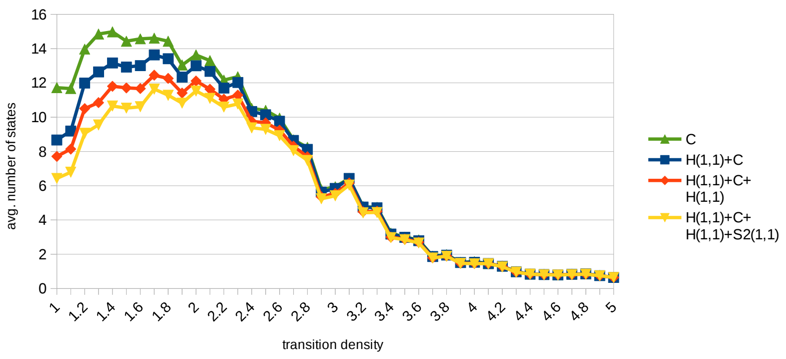


Fig. 7: Reducing and complementing Tabakov-Vardi random tree automata with 4 states. Each data point is the average of 300 automata. In general, applying Heavy(1,1) before the complementation (H(1,1)+C) yielded automata with fewer states, on avg., than direct complementation (C). When Heavy(1,1) is also used after the complementation, the difference is even more significant - H(1,1)+C+H(1,1) - and even more when Sat2(1,1) is used - H(1,1)+C+H(1,1)+S2(1,1).

We also tested our algorithms on random automata with 7 states (Figure 13 in App. C), whose complement automata can have, on avg., up to 100 states and more than 30,000 transitions. As above, reducing automata with Heavy both before and after the complementation returned automata with significantly fewer transitions than direct complementation (3,000 vs 35,000 in some cases), but the former was clearly slower (avg. times up to 90s) than the latter (avg. times up to 2.5s) on the automata region where the difference between the two methods was most drastic. Still, for highly dense automata ($td \geq 4$), direct complementation was responsible for the highest times recorded (avg. times between 135s and 2170s). Due to the size of the complement automata, the saturation methods revealed to be too slow to be viable in this case.

All experiments were run on an Intel Core i5 @ 3.20GHz x 4 machine with 8GB of RAM using a 64-bit version of Ubuntu 16.04.

References

1. Parosh Aziz Abdulla, Ahmed Bouajjani, Lukás Holík, Lisa Kaati & Tomás Vojnar (2008): *Computing Simulations over Tree Automata.* In: *TACAS*, *LNCS* 4963, pp. 93–108. Available at `http://dx.doi.org/10.1007/978-3-540-78800-3_8`.
2. Parosh Aziz Abdulla, Lukás Holík, Lisa Kaati & Tomás Vojnar (2009): *A Uniform (Bi-)Simulation-Based Framework for Reducing Tree Automata. Electr. Notes Theor. Comput. Sci.* 251, pp. 27–48. Available at `http://dx.doi.org/10.1016/j.entcs.2009.08.026`.
3. Parosh Aziz Abdulla, Axel Legay, Julien d'Orso & Ahmed Rezine (2006): *Tree Regular Model Checking: A Simulation-Based Approach. J. Log. Algebr. Program.* 69(1-2), pp. 93–121. Available at `http://dx.doi.org/10.1016/j.jlap.2006.02.001`.
4. T. Genet et al. (2015): *Timbuk.* `http://www.irisa.fr/celtique/genet/timbuk/`.
5. R. Almeida (2016): *minotaut.* `https://github.com/ric-almeida/heavy-minotaut`.
6. Ricardo Almeida, Lukáš Holík & Richard Mayr (2016): *Reduction of Nondeterministic Tree Automata*, pp. 717–735. Springer Berlin Heidelberg, Berlin, Heidelberg. Available at `http://dx.doi.org/10.1007/978-3-662-49674-9_46`.
7. Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz & Tomás Vojnar (2006): *Abstract Regular Tree Model Checking of Complex Dynamic Data Structures.* In: *SAS*, *LNCS* 4134, pp. 52–70. Available at `http://dx.doi.org/10.1007/11823230_5`.
8. Lorenzo Clemente & Richard Mayr (2013): *Advanced automata minimization.* In Roberto Giacobazzi & Radhia Cousot, editors: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, ACM, pp. 63–74. Available at `http://doi.acm.org/10.1145/2429069.2429079`.
9. Lorenzo Clemente & Richard Mayr (2016): *Efficient Reduction of Nondeterministic Automata with Application to Language Inclusion Testing. Submitted to LMCS.*
10. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison & M. Tommasi (2008): *Tree Automata Techniques and Applications.* Available on: `http://www.grappa.univ-lille3.fr/tata`. Release November, 18th 2008.
11. I. Durand (2015): *Autowrite.* `http://dept-info.labri.fr/~idurand/autowrite`.
12. Kousha Etessami (2002): *A Hierarchy of Polynomial-Time Computable Simulations for Automata.* In: *CONCUR*, *LNCS* 2421, pp. 131–144. Available at `http://dx.doi.org/10.1007/3-540-45694-5_10`.

13. Lukás Holík, Ondrej Lengál, Adam Rogalewicz, Jirí Simácek & Tomás Vojnar (2013): *Fully Automated Shape Analysis Based on Forest Automata*. In: *CAV*, *LNCS* 8044, pp. 740–755. Available at `http://dx.doi.org/10.1007/978-3-642-39799-8_52`.
14. Haruo Hosoya (2010): *Foundations of XML Processing: The Tree-Automata Approach*, 1st edition. Cambridge University Press, New York, NY, USA. Available at `http://dx.doi.org/10.1017/CBO9780511762093`.
15. LanguageInclusion.org (Access date:17.12.2015): *RABIT: Ramsey-based Buchi automata inclusion testing*. `http://languageinclusion.org/doku.php?id=tools`.
16. Ondrej Lengál, Jirí Simácek & Tomás Vojnar (2015): *Libvata: highly optimised nondeterministic finite tree automata library*. `http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/`.
17. Ondrej Lengál, Jirí Simácek, Tomás Vojnar, Peter Habermehl, Lukás Holík & Adam Rogalewicz (2015): *Forester: tool for verification of programs with pointers*. `http://www.fit.vutbr.cz/research/groups/verifit/tools/forester/`.
18. Deian Tabakov & Moshe Y. Vardi (2007): *Model Checking Buechi Specifications*. In Remco Loos, Szilárd Zsolt Fazekas & Carlos Martín-Vide, editors: *LATA 2007. Proceedings of the 1st International Conference on Language and Automata Theory and Applications.*, Report 35/07, Research Group on Mathematical Linguistics, Universitat Rovira i Virgili, Tarragona, pp. 565–576.

A Proofs and Counterexamples

For Lemma 1 and Theorem 2 below we make use of the following auxiliary definitions. For every tree $t \in \mathbb{T}(\Sigma)$ and every t-run π, let $level_i(\pi)$ be the tuple of states that π visits at depth i in the tree, read from left to right. Formally, let $(v_1, \ldots, v_n)$, with each $v_j \in \mathbb{N}^i$, be the set of all tree positions of depth i s.t. each $v_j \in dom(\pi)$, in lexicographically increasing order. Then $level_i(\pi) = (\pi(v_1), \ldots, \pi(v_n)) \in Q^n$. We say that $st \in Q^*$ is a subtuple of $level_i(\pi)$, and write $st \leq level_i(\pi)$, if all states in st also appear in $level_i(\pi)$ and in the same order. By lifting preorders on Q to preorders on Q^n, we can compare tuples of states w.r.t. $\subseteq^{\mathsf{up}}(id)$.

Lemma 1. *Let A be a TDTA and $(p_1, \ldots, p_n)$ and $(q_1, \ldots, q_n)$ two tuples of states of A such that $(p_1, \ldots, p_n) \subseteq^{\mathsf{up}}(id)(q_1, \ldots, q_n)$. Then, for every $t \in \mathbb{T}(\Sigma)$, every accepting t-run π and every tuple $(v_1, \ldots, v_n)$ of some leaves of π of the same depth i (i.e., $(v_1, \ldots, v_n) \leq level_i(\pi)$) s.t. $(\pi(v_1), \ldots, \pi(v_n)) = (p_1, \ldots, p_n)$, there exists an accepting t-run π' of A such that $(\pi'(v_1), \ldots, \pi'(v_n)) = (q_1, \ldots, q_n)$ and $\pi'(v) = \pi(v)$ for every leaf v of π' other than $v_1, \ldots, v_n$.*

Proof. Let π be an accepting t-run of A s.t. $(\pi(v_1), \ldots, \pi(v_n)) = (p_1, \ldots, p_n)$. We say that an accepting t-run π'' is i-good iff i) for every node v_j of π'', with $j \leq i$, $\pi''(v_j) = q_j$, and ii) for every v_j, with $i < j \leq n$, $\pi''(v_j) = p_j$. We will show, by induction on i, that for every i there exists an accepting t-run π''' which is i-good and s.t. $\pi'''(v) = \pi(v)$ for every leaf v of π''' other than $v_1, \ldots, v_n$. For the particular case of $i = n$ this proves the lemma.

The base case $i = 0$ is trivial, since the accepting t-run π is 0-good itself.

For the induction step, let π_1 be an accepting $(i-1)$-good t-run of A. If $i > n$, the lemma holds trivially. Otherwise, we have $\pi_1(v_i) = p_i \subseteq^{\mathsf{up}}(id)\ q_i$ and thus there exists an accepting t-run π_2 of A s.t. $\pi_2(v_i) = q_i$. And since the upward trace inclusion is parameterized by id, it follows, in particular, that for every leaf v other than v_i, $\pi_2(v) = \pi_1(v)$. Thus, π_2 is an accepting i-good t-run of A. Moreover, we have that, on leaves other than $v_1, \ldots, v_n$, the run π_2 coincides with π_1 and consequently, by the induction hypothesis, with π.

Theorem 2 $S(\subseteq^{\mathsf{up}}(id), \supseteq^{\mathsf{up}}(id))$ is GFS.

Proof. Let A be a TDTA and $A_S = Sat(A, S(\subseteq^{\mathsf{up}}(id), \supseteq^{\mathsf{up}}(id)))$. If $\hat{t} \in A_S$, then there exists an accepting $\hat{t}$-run $\hat{\pi}$ of A_S. We will show that there exists an accepting $\hat{t}$-run of A, which proves $A_S \subseteq A$.

Let us first define an auxiliary notion. For every $t \in \mathbb{T}(\Sigma)$ and every t-run π, we say that π is i-good iff it does not contain any transition of $\delta_S - \delta$ from any position $v \in \mathbb{N}^*$ s.t. $|v| < i$, i.e., all transitions used in the first i levels of the tree are of A.

Next, we will show, by induction on i, that for every i there exists an accepting i-good $\hat{t}$-run $\hat{\pi}'$ of A_S s.t. $level_i(\hat{\pi}') = level_i(\hat{\pi})$. For i equal to the height of $\hat{t}$, this implies that there exists an accepting $\hat{t}$-run of A.

The base case $i = 0$ is trivial, since $\hat{\pi}$ is 0-good itself.

For the induction step, let us first define some auxiliary notions. For every $t \in \mathbb{T}(\Sigma)$ and every t-run π, we say that $level_{i'}(\pi)$ is j-good iff π does not contain a transition of $\delta_S - \delta$ from a state $\pi(v_k)$, s.t. $k \leq j$ and $\pi(v_k)$ is the k-th state of $level_{i'}(\pi)$. We now say that an accepting $\hat{t}$-run $\hat{\pi}''$ of A_S is $(i-1, j)$-good iff i) it is $(i-1)$-good, ii) $level_{i-1}(\hat{\pi}'')$ is j-good, and iii) $level_i(\hat{\pi}'') = level_i(\hat{\pi})$.

We will now show, by induction on j, that for every j there exists an accepting $(i-1, j)$-good $\hat{t}$-run of A_S. Since trees are finitely-branching, we have that for a sufficiently large j there is an accepting $\hat{t}$-run $\hat{\pi}'''$ of A_S which is i-good. And since, in particular, $level_i(\hat{\pi}''') = level_i(\hat{\pi})$, this will conclude the outer induction.

For the base case $(i-1, 0)$, we know by the hypothesis of the outer induction that there exists an accepting $(i-1)$-good $\hat{t}$-run π_1 s.t. $level_{i-1}(\pi_1) = level_{i-1}(\hat{\pi})$. Then the $\hat{t}$-run π_2 which, on the levels below i, coincides with π_1 and, on the levels from i up, coincides with $\hat{\pi}$ too is accepting and $(i-1)$-good. Thus π_2 is $(i-1, 0)$-good.

For the induction step, let π_1 be an accepting $(i-1, j-1)$-good $\hat{t}$-run of A_S, and let π_1' be the prefix of π_1 which only uses transitions of A. π_1' is thus an accepting run of A over some prefix tree $\hat{t}'$ of $\hat{t}$. Let v_j be the node of $\hat{t}$ s.t. $\pi_1'(v_j)$ is the j-th state of $level_{i-1}(\pi_1')$ and $\sigma = \hat{t}(v_j)$ a symbol of rank r.

If $r = 0$, then v_j is a leaf of $\hat{t}$ and so there exists a transition $\langle \pi_1'(v_j), \sigma, \psi \rangle$ in A_S. By the definition of δ_S, there exists a transition $\langle p, \sigma, \psi \rangle$ in A s.t. $\pi_1'(v_j) \subseteq^{\mathsf{up}} (id)\ p$. Thus there exists an accepting $\hat{t}'$-run π_2 of A s.t. $\pi_2(v_j) = p$ and for any leaf v of π_2 other than v_j, $\pi_2(v) = \pi_1'(v)$. We now obtain a run over $\hat{t}$ again by extending π_2 downwards according to π_1, i.e., $\pi_2(vv') := \pi_1(vv')$, for every leaf v of π_2 other than v_j and for every $v' \in \mathbb{N}^*$. It follows that $level_i(\pi_2) = level_i(\pi_1) = level_i(\hat{\pi})$. π_2 is clearly a $(i-1)$-good $\hat{t}$-run of A_S and $level_{i-1}(\pi_2)$ is j-good. Thus π_2 is an accepting $(i-1, j)$-good $\hat{t}$-run of A_S.

If $r > 0$, then v_j is not a leaf and so there exists a transition $\langle \pi_1'(v_j), \sigma, \pi_1(v_j 1) \ldots \pi_1(v_j r) \rangle$ in A_S. By the definition of δ_S, there exists a transition *trans*: $\langle p, \sigma, q_1 \ldots q_r \rangle$ in A s.t. $\pi_1'(v_j) \subseteq^{\mathsf{up}}(id) p$ and
1) $(q_1 \ldots q_r) \subseteq^{\mathsf{up}} (id)(\pi_1(v_j 1) \ldots \pi_1(v_j r))$. From $\pi_1'(v_j) \subseteq^{\mathsf{up}} (id)\ p$ we have that there exists an accepting $\hat{t}'$-run π_2 of A s.t. $\pi_2(v_j) = p$ and $\pi_2(v) = \pi_1'(v)$, for every leaf v of π_2 other than v_j. Extending π_2 with *trans* we obtain an accepting run of A s.t. $\pi_2(v_j k) := q_k$ for each child $v_j k$ of v_j. Applying Lemma 1 to 1), we obtain that there exists an accepting run π_3 of A over the same prefix tree of $\hat{t}$ as π_2 s.t. 2) $\pi_3(v_j k) = \pi_1(v_j k)$ for each child $v_j k$ of v_j, and $\pi_3(v) = \pi_2(v) = \pi_1(v)$ for every leaf v of π_3 other than $v_j 1, \ldots, v_j r$. We now obtain a run over $\hat{t}$ again by extending π_3 downwards according to π_1, i.e., 3) $\pi_3(vv') := \pi_1(vv')$, for every leaf v of π_3 other than $v_j 1, \cdots, v_j r$ and for every $v' \in \mathbb{N}^*$. π_3 is clearly a $(i-1)$-good $\hat{t}$-run of A_S and $level_{i-1}(\pi_3)$ is j-good. From 2) and 3), we obtain that $level_i(\pi_3) = level_i(\pi_1) = level_i(\hat{\pi})$. Thus π_3 is an accepting $(i-1, j)$-good $\hat{t}$-run of A_S.

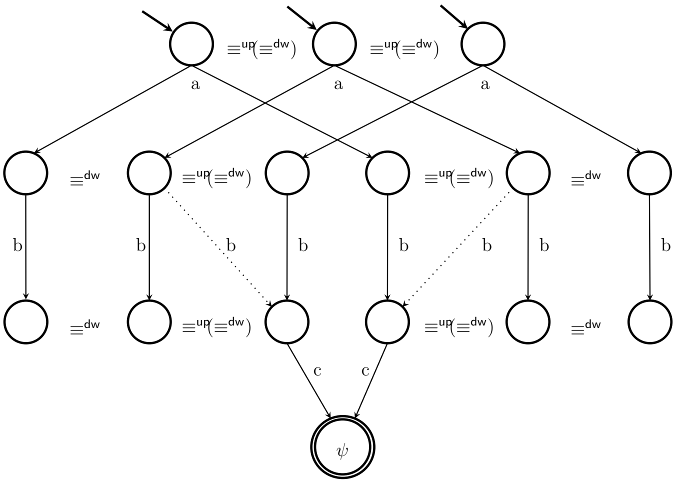

a
a
a
$\equiv^{up}(\equiv^{dw})$
$\equiv^{up}(\equiv^{dw})$
$\equiv^{dw}$
$\equiv^{up}(\equiv^{dw})$
$\equiv^{up}(\equiv^{dw})$
$\equiv^{dw}$
b
b
b
b
b
b
b
b
$\equiv^{dw}$
$\equiv^{up}(\equiv^{dw})$
$\equiv^{up}(\equiv^{dw})$
$\equiv^{dw}$
c
c
ψ

Fig. 8: $S(id, \equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}}))$ is not GFS: if we add the dotted transitions, the tree $a(b(c), b(c))$ is now accepted. The symbols c, b and a have ranks 0, 1 and 2, resp.

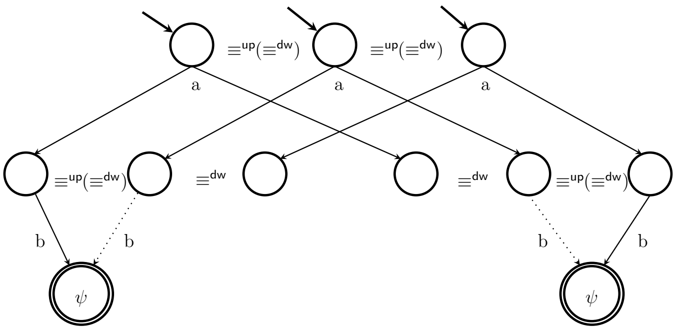

a
a
a
$\equiv^{up}(\equiv^{dw})$
$\equiv^{up}(\equiv^{dw})$
$\equiv^{up}(\equiv^{dw})$
$\equiv^{dw}$
$\equiv^{dw}$
$\equiv^{up}(\equiv^{dw})$
b
b
b
b
ψ
ψ

Fig. 9: $S(\equiv^{\mathsf{up}}(\equiv^{\mathsf{dw}}), id)$ is not GFS: if we add the dotted transitions, the tree $a(b, b)$ is now accepted. The symbols b and a have ranks 0 and 1, respectively.

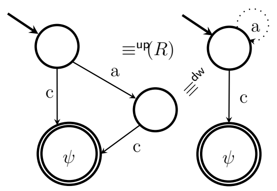


Fig. 10: $S(\equiv^{\mathsf{up}}(R), \equiv^{\mathsf{dw}})$ is not GFS for any relation $R \subseteq Q \times Q$ (example adapted from [9]): if we add the dotted transition, the linear tree aac is now accepted. The symbol c has rank 0 and a has rank 1.

B Variants of the Optimization to the Lookahead Simulation Game

In Section 3 we presented an optimization to the computation of the k-lookahead downward simulation based on the caching of attacks in the simulation game between Spoiler and Duplicator. In this appendix we present two alternative versions to this optimization, where we change the scope of the attacks cached.

Local caching of Spoiler's attacks. Whenever Spoiler uses a transition t in an attack, Duplicator can memorize which states in the automaton are able to defend against the target states of t. In Fig. 1 from Section 3, in a round of the simulation game from (p_2, q_2), Spoiler is attempting the attack $d(e, e)$ leading to p_5, p_7. Duplicator tries responding with a d-transition to (q_4, q_5), and since there is a e-transition from q_4 to q_7 and $p_5 \, W \, q_7$, Duplicator caches the information that, against q_4, the first sub-attack is a *bad* one. However, q_5 can only read f and so Duplicator will have to try a different defence. Duplicator now tries the d-transition leading to q_4 and q_6 instead. Thanks to the information recorded, Duplicator now only needs to find a defence from q_6 against p_6, which exists since q_6 goes to q_8 by e and $p_5 \, W \, q_8$, and so Duplicator declares victory against this particular attack.

Conversely, if the game configuration was (p_2, q_3), after trying to defend against the attack $c(e, e)$ using the c-transition to q_9 and q_{10}, Duplicator could reuse the information that the sub-attack e is *good* against q_9 when trying the c-transition to q_9 and q_{11}.

Global caching of Spoiler's attacks. Here we expand the scope to the entire W-refinement. E.g., the *good* attacks from p_2 against q_2 or against q_3 can be recalled even when a game from a different configuration, say, (p_8, q_1) is played. However, the information about the *bad* attack from, say, p_3 against q_4 cannot be used outside of the local game in which it was saved, since Duplicator could only defend against it based on the state of W at the time. Note the asymmetry between *good* and *bad* attacks: *good* attacks remain *good* for the rest of the entire computation, but *bad* attacks may become *good* after W changes.

C More Charts from the Experimental Results

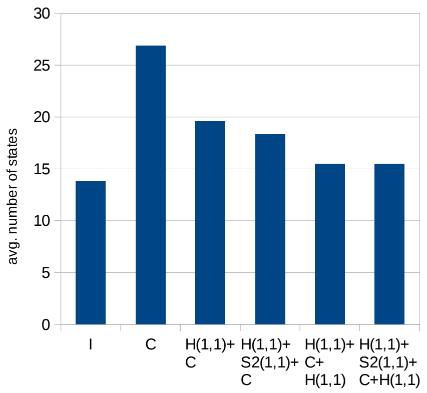
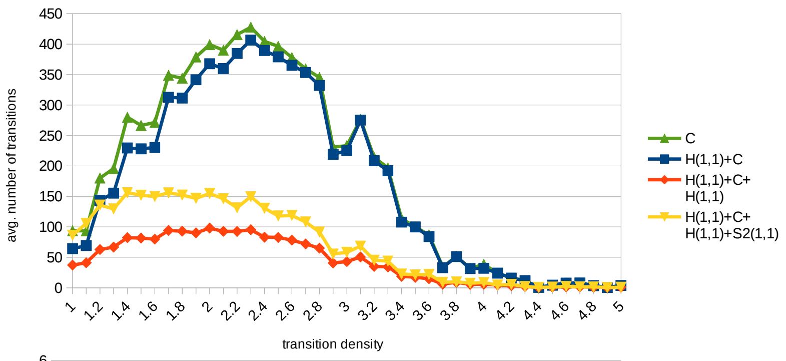


Fig. 11: Reducing and complementing Forester automata with at most 14 states. The complement automata have fewer states if the complementation is preceded by applying Heavy(1,1) and Sat2(1,1) - H(1,1)+S2(1,1)+C - or just Heavy(1,1) - H(1,1)+C. Applying Heavy(1,1) in the end reduces even more. We include the initial number of states (I) for comparison purposes.



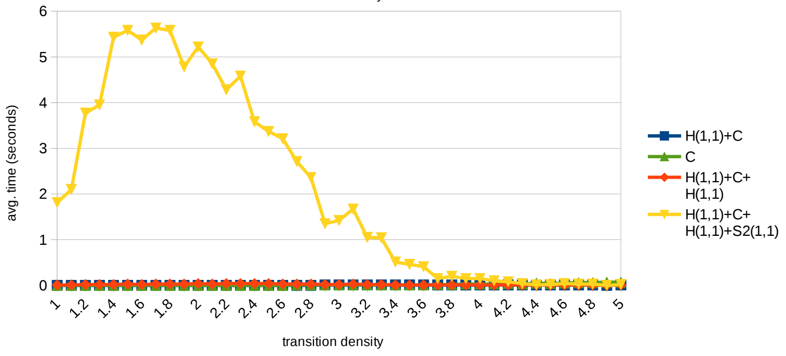


Fig. 12: Reducing and complementing Tabakov-Vardi random tree automata with 4 states. Each data point is the average of 300 automata. In general, applying Heavy(1,1) before the complementation (H(1,1)+C) yielded automata with fewer states and transitions, on average, than direct complementation (C). When Heavy is used both before and after the complementation, the difference is even more significant: H(1,1)+C+H(1,1) produced automata with less than 1/3 of the transitions of C for nearly all values of *td*. Running Heavy followed by Sat2 after the complementation (H(1,1)+C+H(1,1)+S2(1,1)) offered a trade-off between reduction in the states space and in the number of transitions (as well as in time).

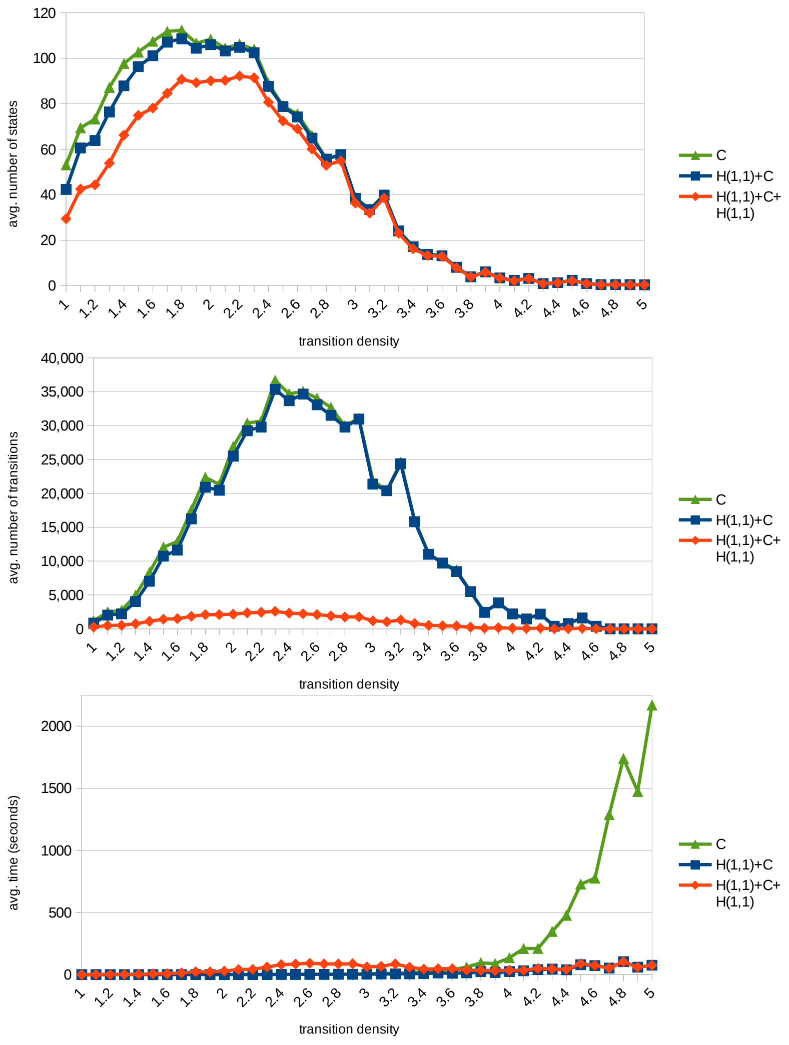


Fig. 13: Reducing and complementing Tabakov-Vardi random tree automata with 7 states. Each data point is the average of 300 automata. In general, applying Heavy(1,1) before the complementation (H(1,1)+C) yields smaller automata than direct complementation (C), on average. When Heavy is used both before and after the complementation (H(1,1)+C+H(1,1)), the difference is even more significant: the automata produced by H(1,1)+C+H(1,1) had between 4 and 24 times less transitions than those yielded by C, but the greater reductions took longer to compute. C still took the longest times recorded, for highly dense automata.

Characterizing DAG-depth of directed graphs

Matúš Bezek[⋆]

Faculty of Informatics,
Masaryk University,
Brno, Czech Republic,
xbezek@fi.muni.cz

Abstract. We study DAG-depth, a structural depth measure of directed graphs, which naturally extends the tree-depth of ordinary graphs. We define a DAG-depth decomposition as a strategy for the cop player in the lift-free version of the cops-and-robber game on directed graphs and prove its correctness. The DAG-depth decomposition is related to DAG-depth in a similar way as an elimination tree is related to the tree-depth. We study the size aspect of DAG-depth decomposition and provide a definition of mergeable and optimally mergeable vertices in order to make the decomposition smaller and acceptable for the cop player as a strategy. We also provide a way to find the closure of a DAG-depth decomposition, which is the largest digraph for which the given decomposition represents a winning strategy for the cop player.

1 Introduction

Structural width parameters are numeric parameters associated with graphs. They represent different properties of graphs. Examples of such parameters are path-width [11], tree-width [12] and clique-width [3]. The first two were defined by Robertson and Seymour in 1980s, clique-width was defined by Courcelle et al. in 1991. Informally, path-width measures how close a graph is to a path and the other two similarly relate to trees.

As a directed analog of tree-width, directed tree-width [7] was defined by Johnson et al. in 1998. This line of research continued in Obdržálek's definition of DAG-width [10] in 2006. Another digraph measure Kelly-width [6] was defined by Hunter and Kreutzer in 2007. In 2010 Ganian et al. analyzed [5] digraph width measures and reasons why the search for the "perfect" directed analog of tree-width has not been successful so far.

All these parameters are tightly correlated with different versions of a cops-and-robber game with an infinitely fast robber. The essence of this game is to catch the robber by placing the cops in the vertices and moving them.

Structural depth parameters are analogously correlated with the so-called lift-free version of the game, defined in Section 2.2. An example of such a parameter is tree-depth [9], defined by Nešetřil and Ossona de Mendez in 2005. In

[⋆] Research supported by the Center of Excellence – Institute for Theoretical Computer Science; Czech Science Foundation Project no. P202/12/G061.

2012 Adler et al. defined [1] a hypergraph analog of tree-depth. In that work Adler et al. also studied generalizations of the elimination tree for hypergraphs.

A directed analog of tree-depth was defined under the name DAG-depth [4] by Ganian et al. in 2009. Its definition, however, did not provide any structural insight into the parameter since there was no naturally associated decomposition with it.

We define a DAG-depth decomposition of a digraph and show that it can be used as a winning strategy for the cop player in the lift-free version of the cops-and-robber game in directed graph. The main issue is that an optimal decomposition usually has to contain multiple copies of original vertices.

2 Preliminaries

We deal with directed graphs.

An *outdegree* $d^+_D(v)$ of the vertex $v \in V(D)$ is the number of edges going from v. An *indegree* $d^-_D(v)$ of the vertex $v \in V(D)$ is the number of edges coming to v. An *out-neighborhood*, denoted by $N^+_D(v)$, is the set of vertices x such that the edge (v, x) exists in D.

An acyclic directed graph is shortly called a DAG. In DAG, vertex u is a parent of v if the digraph contains an edge (u, v). Vertex v is then a child of u. The vertex u is an ancestor of v if the digraph contains a path from u to v. If u is an ancestor of v, then v is a descendant of u.

One of the ways to extend connectivity to directed graphs is the concept of *reachable fragments*. Reachable fragments are maximal, by inclusion, sets of vertices such that every fragment R contains a vertex called the source, from which there is a path to every vertex of R.

2.1 Original cops-and-robber game

The *cops-and-robber* game was first introduced [8] in 1982 by LaPaugh. The variant we are interested in was introduced [13] in 1989 by Seymour and Thomas. The main difference between them is that in the version by Seymour and Thomas the robber is infinitely fast, while in LaPaugh's version he is not.

The game by Seymour and Thomas is played by one player on a finite undirected graph G. The player controls k cops. At any time each of them either stays on some vertex or is temporarily removed from the graph. The player moves the cops, he can remove them from the graph and place them back into any vertex he wants.

The robber always stands on some vertex of G. During the game, he can move at any time along the edges at infinite speed. He is not allowed to run through a cop but he can see when the cop is being placed on some vertex and he can run through that vertex before the cop lands.

The cop player wins when the cops catch a robber, i.e. when the robber is in some vertex v such that there is a cop placed in each vertex of $N^+(v)$ and also on v. The player loses if the robber is able to avoid getting caught.

The robber is always aware of cops' position and the player is aware of robber's position. The minimal number of cops needed to catch a robber on a graph is called the *cop number* of the graph.

2.2 Lift-free version of the game

In the *lift-free* version of the cops-and-robber game there is an additional rule; once the cop is placed to some vertex, he stays there until the end of the game. In each turn the cop player puts a cop onto some vertex of the graph. The game ends when the robber is caught or the cop player runs out of cops. If the robber is caught, the cop player wins, otherwise he loses.

2.3 Extension to directed graphs

The concept of the cops-and-robber game can be naturally extended to directed graphs. The robber can only move along the edges in the right direction.

The aforementioned DAG-depth [4], introduced by Ganian et al. in 2009, is given as follows.

Definition 1 (DAG-depth). *Let D be a digraph and $R_1, \dots, R_p$ the reachable fragments of D. The DAG-depth $ddp(D)$ is inductively defined:*

$$ddp(D) = \begin{cases} 1 & \text{if } |V(D)| = 1 \\ 1 + \min_{v \in V(D)}(ddp(D - v)) & \text{if } p = 1 \text{ and } |V(D)| > 1 \\ \max_{1 \le i \le p} ddp(R_i) & \text{otherwise} \end{cases}$$

DAG-depth is directly related to the lift-free game as follows (where a proof is quite obvious):

Theorem 1. *Let D be a digraph. There exists a lift-free winning strategy for k cops, if and only if DAG-depth of D is less or equal to k.*

3 DAG-depth decomposition

The aim of this section is to define a DAG-depth decomposition (Definition 2) from the recursive definition of DAG-depth (Definition 1) the same way as an elimination tree is obtained from the definition of tree-depth. The decomposition aims to represent a game plan for the cop player.

The main difference between the tree-depth and DAG-depth cases is that in undirected graphs two connected components cannot have any vertices in common, while distinct maximal reachable fragments in directed graphs can have some vertices in common. This naturally brings complications and ambiguity.

There could be two ways to resolve this. Either just ignore it and let the decomposition have more copies of one vertex. But that would mean the decomposition could grow exponentially large (see Section 4, and exponentially large decompositions would be practically useless for the player as a game plan and for algorithmic purposes.

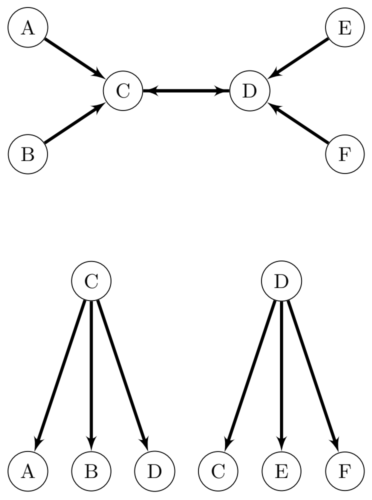


Fig. 1. A simple digraph and its decomposition, showing that DAG-depth decomposition cannot always be done optimally without repetition of vertices (see repeated C,D)

The other extreme solution would be to merge all the copies of one vertex. This cannot always be done, as the graph in Figure 1 shows.

In this graph, the robber can be caught by using two cops. The idea is that if the robber starts on the vertices A or B, the player places the first cop on the vertex C. Then the robber has either stayed on the vertex he was on, or ran to D. Placing the cop on the robber will catch him, since there is no edge between A and D or B and D. Symmetrically, if the robber starts on E or F, the first cop is placed to D and second catches the robber. If the robber starts on C or D, he can not go into any other vertex and covering these two vertices in any order will result into a win.

In the corresponding "decomposition" (Figure 1 bottom), the two copies of the vertex C can not be merged since their merging would create a path of length 2 and therefore the decomposition would not be optimal anymore. For the same reason the copies of D can not be merged, too.

Balancing these two extreme approaches would give us decompositions with some of the repeated vertices merged. Now the core question is, which vertices can be merged and why.

3.1 Basic properties of a DAG-depth decomposition

As argued above, in a decomposition some vertices will be copies of the same original vertex $v \in D$. To properly work with this fact, there is a need to formally distinguish the two vertex sets and map between them.

This is the first difference from an elimination tree of tree-depth where the vertex sets are identical. Formally, this can be done by defining the function $org : V(P) \to V(D)$ which takes a vertex from the decomposition and returns its original from the digraph D. Vertices $x, y \in V(P)$ are copies of the same vertex if and only if $org(x) = org(y)$.

Roots of a DAG are all of its vertices whose indegree is zero. Vertices whose outdegree is zero are called *leafs*.

The *level* of a vertex in a DAG is the maximal length of a directed path from a root to this vertex. The *depth* of a vertex is the maximal length of a path from this vertex to a leaf. The *depth* of a DAG is the maximum depth over its vertices.

Definition 2 (DAG-depth decomposition). *A DAG-depth decomposition of a digraph D is a DAG P and a surjective function $org : V(P) \to V(D)$. Furthermore, a DAG-depth decomposition is called valid if the following Neighbor cover condition holds.*

The Neighbor cover condition states that for every vertex $v' \in V(P)$ such that $org(v') = v$, the following holds:
For every $u \in N_D^+(v)$,

1. *there exists $u' \in V(P)$ such that $org(u') = u$ and u' is a descendant of v' in P, or*
2. *every path from any root of P to v' contains a vertex $u' \in V(P)$ such that $org(u') = u$.*

Let P be a DAG-depth decomposition of some graph D such that the depth of P is equal to the DAG-depth of D. P is then called an *optimal* decomposition. Such decomposition exists for any digraph D, as Theorem 3 shows.

The following example of Figure 2 illustrates how a valid DAG-depth decomposition can be used as a strategy for the cop player to catch the robber.

If the player is to use the decomposition in Figure 2 as a game plan, he starts by covering the vertex E, since its copy is the only root.

Then, if the robber is in the vertex A or B, the player continues by covering the vertex B. If the robber was in this vertex, he has been caught. Otherwise he is in the vertex A, which the player will cover by the third cop and therefore catch the robber.

If the robber was not in the vertex A or B, the player places the second cop in the vertex G, whose copy is on the same level as the copy of B. There are now three possibilities where the robber can be. The first one is that he is in one of

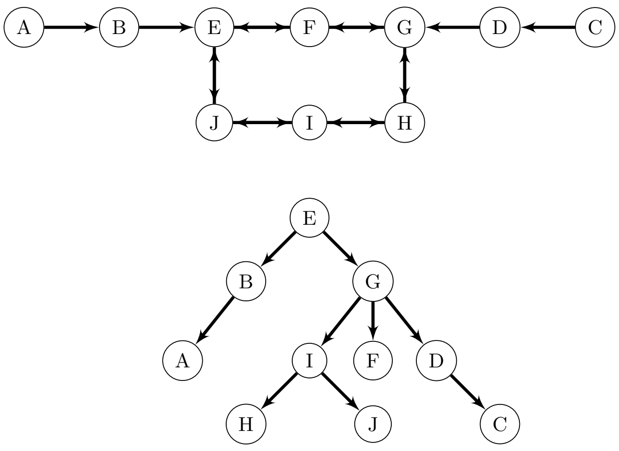


Fig. 2. A simple graph and its valid and optimal decomposition

the vertices C, D. The second one is that he is in the vertex F. The last one is that the robber is hiding in one of the vertices H, I, or J.

If it is the first case, the player continues by covering the vertex D, whose copy is the child of the copy of the last covered vertex. If the robber was here, he is caught, otherwise the cop is placed to C and catches the robber.

If it is the second case and the robber is hiding in the vertex F, the player simply covers the vertex F and catches the robber. If the robber is in one of the vertices H, I, or J, the player covers the vertex I, whose copy is the child of the copy of the last covered vertex G. The robber then escapes either to vertex H or to J. In the last step the player simply covers the vertex robber is in and catches him.

The game rules outlined in this example are formally defined next.

Definition 3. *Given a DAG-depth decomposition (P, org) of a digraph D, the cop player's strategy is as follows. The cops are placed on the vertices of D and every cop is placed "because of" some vertex of P. The following convention is observed: if we say a cop is to be placed to a vertex $v' \in V(P)$, he is actually placed to $v \in V(D)$ such that $org(v') = v$, unless the vertex has been covered by another cop before. In that case, no cop is placed in this step. Then, the strategy based on (P, org) is described by two simple rules:*

1. *The first cop is always placed to one of the roots of P. Each subsequent cop is placed to the out-neighborhood of the previous cop in P.*
2. *Among the possible positions from 1, the actually chosen one must have in P a directed path to a copy of robber's current position.*

The choice of the next vertex to be covered by a cop in Definition 3 is generally non-deterministic, since more vertices can contain robber's position as a descendant.

Theorem 2. *Let D be a digraph and (P, org) its DAG-depth decomposition of depth k. Then the decomposition is valid if and only if every strategy based on (P, org) by Definition 3 is winning for k cops.*

Proof. ($\Leftarrow$) The decomposition is valid if the Neighbor cover condition holds by Definition 2. Suppose the contrary: there exist a pair of vertices $u, v \in V(D)$ such that edge $(u, v) \in E(D)$ exists. Also a vertex $u' \in V(P)$ exists such that $org(u') = u$ and u' does not contain any copy of the vertex v as a descendant. Since none of the conditions in the definition of the Neighbor cover condition holds, there exists a path p from some root to u' such that p does not contain any copy of the vertex v.

Let the player use the decomposition according to rules specified in Definition 3. If the robber started on the vertex $u \in V(D)$, then the player could proceed along the path p, since all of its vertices contain the copy $u' \in V(P)$ as a descendant. When the player covers the vertex u because of u', the robber can escape to the vertex $v \in V(D)$ since the path p does not contain any copy of that vertex and therefore it is not covered by a cop. Since the vertex u' does not contain any copy of v as a descendant, the player playing according the Definition 3 can not cover v and the robber wins. The given decomposition therefore represents a strategy which is not winning.

($\Rightarrow$) The other direction is the subject of subsequent claims and will follow from Theorem 4.

Theorem 3. *If the DAG-depth of a digraph D is k, then there exists a valid DAG-depth decomposition of D of depth k.*

Proof. If $|V(D)| = 1$, then $ddp(D) = 1$. A decomposition with depth one exists, since it consists also of the only vertex.
A decomposition that consists of one vertex is always valid, since the original graph did not contain any edges and the Neighbor cover condition therefore always holds.

If $|V(D)| > 1$ and D consists of the only reachable fragment, then the DAG-depth is computed as $ddp(d) = 1 + \min_{v \in V(D)}(ddp(D - v))$. Such vertex v is then the root of the decomposition and is connected to the roots of the recursive decomposition of the rest of a graph.
Since the vertex v was chosen to be the root, all other vertices are its descendants. Therefore all the vertices of its out-neighborhood are his descendants, and for the rest of the graph the Neighbor covercondition holds by induction. That means the decomposition is valid.

Otherwise, D consists of more reachable fragments. The decomposition of each of them can be computed separately. When a disjoint union of them is made to a single graph, its depth will be equal to the maximum of the decompositions of the fragments. This is in accordance with Definition 1.

Since the decomposition is a disjoint union of the decompositions of the fragments, by induction the decompositions of the fragments are valid. Therefore their disjoint union is a valid decomposition too.

Theorem 4. *Let D be a digraph for which there exists a valid DAG-depth decomposition of depth k. Then, any strategy observing the rules of Definition 3 is a winning strategy for at most k cops.*

Proof. A decomposition (P, org) is valid if the Neighbor cover condition from Definition 2 holds. The cop player wins when the cop is placed on top of the robber to a vertex r and all vertices from $N_D^+(r)$ are already covered by the cops.

Let the last move of the cop be to vertex $u \in V(D)$, because of its copy $u' \in V(P)$ as in Definition 3. We claim that the robber may move only to vertices of D whose copies in P are reachable from u' in P.

Before the robber moves, the previous statement holds because of the rule 2 of Definition 3.

Let the robber be on a vertex $r \in V(D)$ and $r' \in V(P)$ its copy such that it is a descendant of u'. The statement still holds if the robber moves along an edge $(r, v) \in E(D)$ to vertex $v \in V(D)$ which has not yet been covered by a cop. From Definition 2 we know that in the decomposition P every path from a root to r' contains a copy of v or P contains a vertex $v' \in V(P)$ such that it is a descendant of r'. If v' is a descendant of r', it is also a descendant of u' since r' is its descendant. If every path from a root to r' contains a copy of v, such copy must be a descendant of u'. If it was not, then v' would have to lie on some path from a root to u', and v would have already been covered by a cop by Definition 3.

The previous invariant allows the player to always fulfill the rules of Definition 3.

The rules in Definition 3 end with covering a vertex from $V(D)$ because of its copy which is a leaf in decomposition P. That means that all the neighbors of the covered vertex have been covered before and the robber is caught. The decomposition therefore represents a winning strategy.

In every move, the vertex $v \in V(D)$ is covered because of some $v' \in V(P)$. Such vertex v' is always a child of the previous v' and therefore all such vertices create a path in P. If the player used more than k cops, the path would need to be longer than k. Since the depth of P is k, such path can not exist. Therefore, the decomposition represents a strategy for at most k cops.

4 Merging the copies

We now return to the size aspect of a DAG-depth decomposition (say, the one obtained by Theorem 3). We start with an example that it could be exponentially large. To reduce the size of the decomposition, some copies of the same vertex should be merged while preserving validity of the decomposition. Not all vertices with the same original can be merged (recall Figure 1, though.

Theorem 5. *There exists a digraph D such that its only valid and optimal DAG-depth decomposition without merging any vertices is exponentially large.*

Proof. Let D be a digraph such that $V(D) = \{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n\}$ for $n \in \mathbb{N}$ and $E(D) = \{(a_i, a_j) \cup (a_i, b_j) \cup (b_i, a_j) \cup (b_i, b_j)\}$ for every $1 \le i < j \le n$. See Figure 3.

The digraph D consists of two isomorphic reachable fragments – $V(D) \setminus \{a_1\}$ and $V(D) \setminus \{b_1\}$.

In the reachable fragment $V(D) \setminus \{b_1\}$ the only optimal first move of the cop player is placing the cop onto a_1 since if he made any other move, one of the subgraphs $\{a_1, a_2, \ldots, a_n\}$ and $\{a_1, b_2, \ldots, b_n\}$ is left uncovered. These subgraphs each require another n cops to catch the robber, while the DAG-depth of D is $n = 1 + n - 1$.

After covering a_1, the remaining digraph has the vertex set $\{a_2, \ldots, a_n, b_2, \ldots, b_n\}$. Its decomposition can be found by induction.

The reachable fragment $V(D) \setminus \{a_1\}$ is isomorphic to $V(D) \setminus \{b_1\}$ and therefore the only optimal move is to cover b_1 and the remaining digraph is the same as for the first reachable fragment.

The decomposition of D will thus contain decomposition of the remaining digraph two times – as a descendant of a_1 and as a descendant of b_1. The decomposition therefore consists of $\sum_{i=1}^{n} 2^i = 2^{n+1} - 2$ vertices.

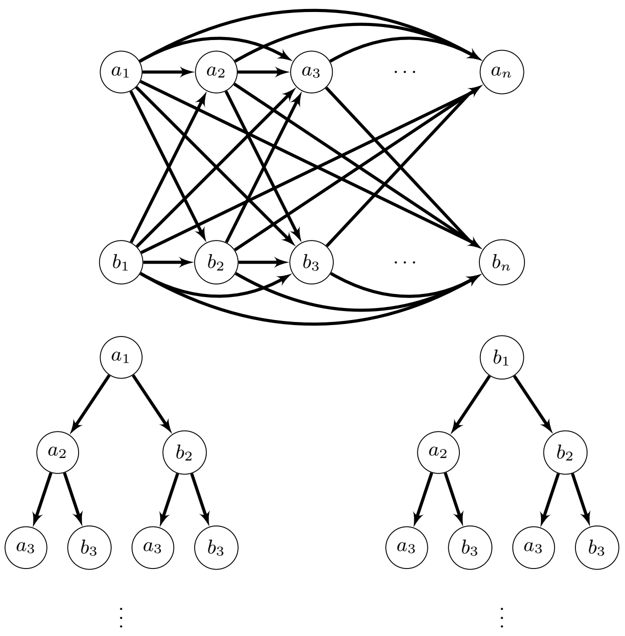


Fig. 3. A digraph and its exponentially large valid and optimal DAG-depth decomposition without merging of any vertices

Definition 4. *Two vertices $u, v \in V(P)$ are mergeable in the DAG-depth decomposition (P, org) if the conditions $1 - 3$ hold. Furthermore, u and v are optimally mergeable if they are mergeable and also the condition 4 holds.*

1. *$org(u) = org(v)$*
2. *After merging u with v, P remains a DAG.*
3. *Merging u with v does not break the Neighbor cover property from Definition 2.*
4. *Merging u with v does not increase the depth of the decomposition.*

For example, all duplicits in the example of Theorem 5 are optimally mergeable.

The following is obvious from the definition:

Proposition 1. *Let (P, org) be a valid and optimal DAG-depth decomposition of some graph and $u, v \in V(P)$ an optimally mergeable pair of vertices of P. Then, after merging u and v, P is still a valid and optimal decomposition.*

The trivial lower bound on the size of a valid decomposition is equal to the number of vertices of the original graph, but such a decomposition may not be optimal. The question of minimizing the size of a valid and optimal decomposition is left for further investigations.

5 Closure of a DAG-depth decomposition

While previous text focused on how to construct a DAG-depth decomposition, or a game plan, for a given digraph, now we look from the other side. Roughly, having a game plan, can we easily say on which digraphs we can win with it?

Recall that in the case of tree-depth this was trivial - already the definition of the tree-depth decomposition worked with the concept of a closure of a rooted forest, which, at the same time, represents the unique maximal graph on which the cop player always wins when following the decomposition as the game plan.

However, in the case of digraphs and DAG-depth, we again face unprecedented complications. A DAG-depth decomposition, unlike an elimination tree, can contain more copies of a single vertex of the original digraph. Therefore a problem, which was trivial in undirected graphs, becomes complex.

In the closure obtained from an elimination tree, each vertex is connected with all of its former ancestors and descendants. In a DAG-depth decomposition, more copies of a vertex can have different ancestors and descendants.

We thus define the following.

Definition 5 (Closure). *A partial closure C is a directed graph obtained from a DAG-depth decomposition (P, org) of some graph D, such that D is a spanning subgraph of C and (P, org) is still a valid DAG-depth decomposition for the digraph C. A closure is a maximal partial closure by inclusion.*

Theorem 6. *For a DAG-depth decomposition (P, org) of a digraph D, we construct a digraph C, such that $V(C) = V(D)$ by iteratively adding edges (u, v) for $u, v \in V(C)$ if for every $u' \in V(P)$ which is a copy of u*

1. *there exists $v' \in V(P)$ such that v' is a copy of v and v' is a descendant of u' in P, or*
2. *every path from a root of P to u' contains a copy of v.*

Then, C is a closure of P, which is thus unique.

Proof. The conditions in this theorem are the same as the Neighbor cover property in Definition 2 and so C is clearly a partial closure. On the other hand, every other edge not in $E(C)$ would, by its own, violate Definition 2 and so C is maximal.

These are some of the informal ideas worth further investigation - see [2] for more details.

6 Conclusion

We have presented a definition of a DAG-depth decomposition which extends the concept of an elimination tree as a winning strategy for the cop player in the lift-free version of the cops-and-robber game to directed graphs. Unlike in the case of an elimination tree, the vertex set of a DAG-depth decomposition is not equal to the vertex set of the original graph. That requires us to deal with the two vertex sets and to find a way to map between them. Since the vertex sets are not equal, the size aspect of the DAG-depth decomposition becomes a problem. In the primitive handling, the size of the decomposition can grow exponentially. To make the decomposition smaller and therefore acceptable as a strategy for the cop player, we provide a definition of mergeable and optimally mergeable vertices.

Secondly, we present a definition of the closure as the largest graph where the given decomposition works as a winning strategy. We also provide a way to deterministically obtain a closure for a given decomposition.

The main direction for possible future improvements and extension of our results is to study the lower bounds on the size of a valid and optimal DAG-depth decomposition of a digraph and a relationship between these bounds and the properties of given digraphs.

References

1. I. Adler, T. Gavenčiak & T. Klimošová (2012): *Hypertree-depth and minors in hypergraphs.* *Theoretical Computer Science* 463, pp. 84–95, doi:10.1016/j.tcs.2012.09.007.
2. M. Bezek (2016): *Characterizing DAG-depth of directed graphs.* Bachelor's thesis, Masaryk University, Faculty of Informatics.
3. B. Courcelle, J. Engelfriet & G. Rozenberg (1993): *Handle-rewriting hypergraph grammars.* *Journal of Computer and System Sciences* 46(2), pp. 218–270, doi:10.1016/0022-0000(93)90004-G.
4. R. Ganian, P. Hliněný, J. Kneis, A. Langer, J. Obdržálek & P. Rossmanith (2014): *Digraph width measures in parametrized algorithmics.* *Discrete applied mathematics* 168, pp. 88–107, doi:10.1016/j.dam.2013.10.038.
5. R. Ganian, P. Hliněný, J. Kneis, D. Meister, J. Obdržálek, P. Rossmanith & S. Sikdar (2016): *Are there any good digraph width measures?* *Journal of Combinatorial Theory, Series B* 116, pp. 250–286, doi:10.1016/j.jctb.2015.09.001.
6. P. Hunter & S. Kreutzer (2008): *Digraph measures: Kelly decompositions, games, and orderings.* *Theoretical Computer Science* 399(3), pp. 206–219, doi:10.1016/j.tcs.2008.02.038.
7. T. Johnson, N. Robertson, P. D. Seymour & R. Thomas (2001): *Directed Tree-Width.* *Journal of Combinatorial Theory, Series B* 81(1), pp. 138–154, doi:10.1006/jctb.2000.2031.
8. A. S. LaPaugh (1993): *Recontamination does not help to search a graph.* *Journal of the ACM* 40(2), pp. 224–245, doi:10.1145/151261.151263.
9. J. Nešetřil & P. Ossona de Mendez (2006): *Tree-depth, subgraph coloring and homomorphism bounds.* *European Journal of Combinatorics* 27(6), pp. 1022–1041, doi:10.1016/j.ejc.2005.01.010.

10. J. Obdržálek (2006): *DAG-width: connectivity measure for directed graphs. Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 814–821, doi:10.1145/1109557.1109647.
11. N. Robertson & P. D. Seymour (1983): *Graph minors. I. Excluding a forest. Journal of Combinatorial Theory, Series B* 35(1), pp. 39–61, doi:10.1016/0095-8956(83)90079-5.
12. N. Robertson & P. D. Seymour (1986): *Graph minors. II. Algorithmic aspects of tree-width. Journal of Algorithms* 7(3), pp. 309–322, doi:10.1016/0196-6774(86)90023-4.
13. P. D. Seymour & R. Thomas (1993): *Graph Searching and a Min-Max Theorem for Tree-Width. Journal of Combinatorial Theory, Series B* 58(1), pp. 22–33, doi:10.1006/jctb.1993.1027.

Sending Money Like Sending E-mails: Cryptoaddresses, The Universal Decentralised Identities

Michal Zima
xzima1@fi.muni.cz

Faculty of Informatics, Masaryk University,
Brno, Czech Republic

Abstract. Sending money in cryptocurrencies is majorly based on public keys or their hashed forms—"addresses." These long random-looking strings are user unfriendly for transferring by other means than via copy-and-paste or QR codes. Replacing such strings with identifiers chosen by users themselves would significantly improve usability of cryptocurrencies. Such identifiers could be memorable, easier to write on paper or to dictate over phone. Main challenge lies in designing a practically usable decentralised system for providing these identifiers. Former solutions have been built as centralised systems or come with nonnegligible limitations. Our solution is reminiscent of a prevalent e-mail system, which is an already user friendly and desirably decentralised system. It is shown that our approach is directly applicable also to other systems that use long cryptographic identifiers.

1 Introduction

Nowadays, the most common way for sending money in cryptocurrencies is to copy-and-paste payee's address—usually a string of over 30 random characters long hash of their public key—or to scan its QR code representation. For instance, in Bitcoin such address looks like this: 1NS17iag9jJgTHD1VXjvLCEnZuQ3rJDE9L. However, it is far from being user friendly in terms of manual transfer—on paper, over phone or in one's memory.

A primary requirement for a system providing mapping of user friendly identities to the original user unfriendly strings is to be decentralised. Centralised systems usually already provide user friendly identities, while decentralised trustless systems often do not. It is crucial for the new system to keep up with the systems it provides identities for, i. e., not to be centralised (and potentially a single point of failure) in a decentralised environment.

Our system of cryptoaddresses, presented in this paper, introduces e-mail-like identifiers, leveraging the existing DNS system. Usage of DNS provides straightforward decentralisation on the level of "domain namespaces," designating a server of user's own choice for each of their domain names. Servers provide resolution of cryptoaddresses to the original cryptographic identifiers. The system is secured both on the level of DNS and communication with servers. The scope of applicability is not limited to cryptocurrencies—the system is universal with variety of possible applications.

The remainder of this paper is organised as follows. Section 2 presents previous work related to the problem. The overall design of the system is discussed in section 3, including definitions of used identifiers. Section 4 elaborates on details of utilisation of the DNS system, followed by a design of our protocol SCAP in section 5. Security considerations are presented in section 6. We discuss possible future work in section 7 and provide final conclusions in section 8.

2 Related Work

Endeavours to make complex addresses more usable began within Bitcoin by a concept of "first bits." First bits make use of implicit information contained in cryptocurrency's database of transactions—blockchain. Using first bits, an address can be identified by a short prefix of only several characters. This prefix is expanded to a full address by looking up in the blockchain the first address matching the prefix [8]. However, first bits have also several disadvantages. Most notably, prefix lookup and obtaining a new prefix are time-expensive operations. Besides, it is not guaranteed to be good-looking, the address has to be already used, i. e., to have some money sent to it (this fact alone goes against a common practice of not reusing addresses in cryptocurrencies), and the prefix cannot be revoked or reassigned.

Later on, many centralised systems arose, e. g., Keybase [5] or Gravatar [9]. Nonetheless, none of them has ever been adopted by the cryptocurrency community.

In 2014, community of a cryptocurrency called Monero created a project OpenAlias [16]. OpenAlias enables translation of domain names to addresses of various cryptocurrencies. This is achieved by specially crafted TXT records in DNS zones of respective domain names. While the record can be secured with DNSSEC against tampering, similarly to first bits, revocation and replacement of the cryptocurrency address included in the record is a subject to expiration period (TTL) of DNS caches, therefore a change in the record may be visible after delay.

All current approaches share a drawback in their design: they are one-to-one mappings. It is not possible to dispense a different cryptocurrency address to every user. Instead, all obtain the same one. This is a nonnegligible flaw in terms of privacy preservation. Not only is publicly known the connection between the identity and the address, but the address itself gets reused, making linkability of different payments to a single entity trivial.

3 Design of Cryptoaddresses

Our scope is a decentralised system for providing users with arbitrary identities. Primary goal of this system is to translate these identities to addresses in any cryptocurrency. We call this system *cryptoaddresses*.

Systems based on a shared database, e. g., first bits, retain impractical properties related to the database use and maintenance. In contrast, systems built upon existing DNS infrastructure can leverage its inherent distribution of responsibility. As with e-mail, every administrator of a domain name designates servers responsible for receiving e-mails for this particular domain name. Thence, cryptoaddresses are made decentralised by utilising DNS, and further employing DNSSEC to secure information stored in and retrieved from the DNS.

3.1 Format of Cryptoaddresses

A cryptoaddress (also called a *Crypto ID*, abbreviated as *CID*) is formed of a *local part*, an *@ symbol* and a *domain part*: local-part@domain-part. A cryptoaddress intentionally resembles an e-mail address, because of familiarity of these addresses to users—the format is nowadays easily recognisable and is known to form an address. The same format is also used in XMPP to form an *XMPP address* (or a *Jabber ID*, *JID*) [15].

Since the local part may also contain @ symbols, always the last one is used as a separator of the local and domain part. Format of the local part is not restricted and may be formed of any valid UTF-8 string. Length of the local part may be up to 1023 bytes; zero length is allowed. This length limit is high enough for any practical use case and at the same time gives a concrete upper bound useful for implementers.

The domain part must conform to the format of a fully-qualified domain name (FQDN) [4], without an ending dot. Internationalised domain names are allowed—conversion to standard ASCII format of domain names is done in a way specified by the IDNA standard [12].

3.2 Identifiers of Addressable Services

Single cryptoaddress may be used as identity for various systems and services at the same time. To internally differentiate one from another, each such addressable service must be uniquely identified. This is especially of a concern within cryptocurrencies as there are cryptocurrencies with ambiguous names and/or codes (or even multiple codes). To provide a guidance for such situations, we propose usage of a hash of their genesis block (i. e., the very first block in the block chain) in hexadecimal encoding, which provides unique identification not only among other cryptocurrencies, but also among other systems. Examples for few cryptocurrencies are shown in table 1.

For other kinds of services, e. g., a communication platform Bitmessage[1], a single identifier needs to be established by them. We do not suggest any con-

[1] `https://www.bitmessage.org/wiki/Main_Page`

Table 1. Examples of service identifiers for several cryptocurrencies

Bitcoin	000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
Litecoin	12a765e31ffd4059bada1e25190f6e98c99d9714d334efa41a195a7e7e04bfe2
Dogecoin	1a91e3dace36e2be3bf030a65679fe821aa1d6ef92e7c9902eb318182c355691

crete identifier for individual services. Nevertheless, it may be suggested to use lowercase ASCII characters for these identifiers, mainly for their ease of use and recognition.

Note that these service identifiers are important for flawless internal functioning of the system while users usually do not encounter them. For details on their usage in the protocol see section 5.2.

3.3 System Components

The system is designed of several components: a client, a cryptoaddress, a SRV DNS record, a cryptoaddress server and a protocol for communication with the server. The cryptoaddress server is specific for the domain given in the cryptoaddress. For a single cryptoaddress there might be more than one such server—for load balancing or high availability purposes, as detailed in section 4.

A client communicates with a cryptoaddress server via a Simple Cryptoaddress Protocol, described in section 5. The communication is encrypted with a shared key derived from server's public key and client's private key using ECDH algorithm [13].

4 DNS Records

An intermediate step in the translation of a cryptoaddress to target data is obtaining an address of the translating server. This address is provided by DNS infrastructure. Out of available DNS record types we choose SRV record. It is a record type dedicated for service discovery, i. e., for this kind of information, and provides great flexibility for designated server's FQDN, L4 protocol, port number, priority and weight within a set of servers designated for the same application protocol [10].

In DNS zone format the record has the following form:

```
_service._proto.name. TTL class SRV priority weight port target.
```

We propose the service identifier `scap` and a TCP port 4332. Nonetheless, a server administrator may choose to use arbitrary port number since clients retrieve information about it in the SRV record.

4.1 FQDN Formation

Traditional approaches for establishing a secure connection to a server include X.509 certificates and trust-on-first-use (TOFU) trust model with server's public key provided by the server itself. Nonetheless, to simplify both the secure key distribution and the communication protocol, we leverage an idea originally found in DNSCurve [7]. Instead of sending the server's public key during the initial protocol handshake, we encode it in the server's name. Therefore, a client obtains the key from server's FQDN which is included in the SRV record and even the very first message sent to the server can be hence encrypted. Security achieved with this approach is higher than with TOFU model due to the secure key distribution. In this section we describe how the key encoding into FQDN is done.

We place the key preceded by a version prefix into a dedicated FQDN label. By "FQDN label" we understand a part of FQDN delimited by dots from other FQDN labels. For encoding of the version prefix and public key we use the same base-32 encoding as DNSCurve. The specification of used base-32 encoding is provided within the DNSCurve specification in [7]. Cryptoaddresses are likely to be used together with DNSCurve, therefore it is advantageous to use single encoding algorithm. Besides, a standard base32 encoding algorithms are not suitable for use in FQDN due to their usage of padding character = (equals sign) [11].

Version prefix is a four-character string `1000` which is formed by a base-32 encoded 2-byte little endian number `1`. Encoding of 2 bytes always results in 4 characters, therefore this FQDN label can be always safely split to a version prefix and a key before performing an actual base-32 decoding. If some further version of cryptoaddresses uses more labels in a FQDN, version prefix must be always present in the leftmost label. Version does not only determine the actual FQDN creation, it also indicates version of the protocol to be used for communication with the server. This way there is no need to indicate version in the messages sent between a client and a server and compatibility can be resolved before initiation of the communication.

After the version prefix a base-32 encoded public key follows. As a cryptographic scheme for the key we use Curve25519, a scheme based on elliptic curves. Its choice is based on its well-founded choice of elliptic curve and constants, availability of high-grade implementations and small size of keys, while providing reasonable security [3]. Although small size of public keys is not necessary, it is still beneficial as it allows keys to be directly used. Still, usage of elliptic curve cryptography makes it possible to leverage ECDH algorithm for establishment of a shared key without any additional communication [13].

4.2 Deployment Considerations

When a cryptoaddress server serves many domains or domains whose DNS zones are out of control of the cryptoaddress server operator, it may be more convenient to use an alias in the domains' SRV records. An alias can be, for instance, *scap1.example.com* pointing via a CNAME DNS record to the canonical name as described above. This provision also makes it easier to rotate keys used by cryptoaddress servers (e. g., when corresponding private keys are exposed during a security breach), as it requires change to only one DNS record per cryptoaddress server.

It is possible to operate several cryptoaddress servers, e. g., to enhance availability of cryptoaddresses within a domain. However, their operator must ensure that they provide equivalent information. In a case one server is missing information about a pair of a certain cryptoaddress and a service, client processes this information as authoritative and does not repeat their request to other servers.

An example of SRV records for a setup of two load-balanced servers and one backup server for a domain example.org follows (each record is broken into 2 lines by "\" in order to fit into this limited space):

```
_scap._tcp.example.org. 86400 IN SRV 10 65 4332 \
    1000vs2nh9b3gz04db4rgpjmzv2cwlnpvh3qzn6xljwyxmnp57j8h0d.example.org.
_scap._tcp.example.org. 86400 IN SRV 10 35 4332 \
    100027q245f6cglhdjyy91vk5btyszk6g5fnhz7mvsc6mtfjh2q0c14.example.org.
_scap._tcp.example.org. 86400 IN SRV 30  0 4332 \
    10009ydzvtccqmbzw6q0zlgumtr227g0kwb2zk8h5rv7yruj7gg6zh3.example.org.
```

The first two servers have priority 10 (which is higher than priority 30) with weights 65 and 35, meaning that requests to these two servers will be statistically divided by ratio 65:35. If both of them are unavailable to clients, a server with the next lowest priority is selected, i. e., the server with priority 30 in this case. All three servers use TCP port 4332. The last part of each record (all starting with *1000...*) is the actual address of the cryptoaddress server with encoded version and public key.

5 Simple Cryptoaddress Protocol

Simple Cryptoaddress Protocol (abbreviated *SCAP*) is designed for communication of clients with cryptoaddress servers. It is a synchronous protocol minimising the number of needed round trips (i. e., request-reply pairs), while maintaining its universality and extensibility. Its security design is inspired by DNSCurve extension of DNS [7]—it shares with DNSCurve the scheme for keys and their exchange, nonce extension technique, or use of cryptographic boxes. Nonetheless, the binary format differs and different is also our choice of algorithm for cryptographic boxes (for authenticated encryption).

Distribution of servers' public keys is carried out by embedding them in FQDN as described in section 4.1, therefore every client has a public key of the server they connects to. A client provides the server with their public key in the first message they sends to it. In order to prevent connection hijacking, the server remembers client's public key for the duration of the session and the client never resends their public key within the session.

For cryptographic boxes we use IETF standard ChaCha20-Poly1305 authenticated encryption algorithm [14].

5.1 General Message Format

Format of messages slightly differs depending on whether a message is sent by a client or a server and whether it is client's first message in the session or not.

First Client's Message

– 32 bytes: client's public key.
– 6 bytes: client's nonce choice.
– Cryptographic box with the following content: `H`

Follow-up Client's Message

– 6 bytes: client's nonce choice.
– 6 bytes: server's nonce extension.
– Cryptographic box with the actual message content.

Server's Message

– 6 bytes: client's nonce.
– 6 bytes: server's nonce extension.
– Cryptographic box with the actual message content.

Nonces A nonce in this context is a 96-bit number that is unique for each message within a single shared key (otherwise the shared key would be exposed). Half of the nonce is chosen by the client and half by the server. While the basic principle for nonce creation is shared with DNSCurve, details differ due to session-oriented nature of the communication in SCAP. Mainly, only the first client's message uses zero server's nonce extension for the cryptographic box—any follow-up client's message repeats the extension from the last server's message. Similarly, server always repeats client's chosen part of the nonce from their last message. Since neither client nor server repeat their part of the nonce, each message is guaranteed to have a unique nonce.

5.2 Protocol Messages

The protocol is client-driven—the server only sends responses to client's requests. Both requests and responses use a format based on the netstring encoding, as defined in [2]. In brief, a netstring has a format `[len]:[string],`, where `[len]` is a decimal representation of length of `[string]` in bytes. Netstrings are usually catenated, but are also allowed to be nested.

The first byte of a message determines a type of the message. We specify the following types of client's messages:

H A hello message, sent right after opening the connection. The message content consists only of this byte. The primary purpose of this message is to provide the server with the client's public key.

Q A query request. It consists of two netstrings (nested inside of the main message netstring): a cryptoaddress to be resolved and a service identifier.

E This message type is reserved for possible use by future protocol extensions. Nonetheless, its exclusive use by extensions is not strictly required—an extension may also define new message types.

X A type reserved for proprietary/non-standard protocol extensions. The message consists of the extension's name (without the leading "X" which would normally signal that it is a non-standard extension) followed by a space (U+0020 in Unicode) and the actual data of the message. Non-standard extensions are required to use this message type.

A server always responds to these messages with a message with one of the following types:

O Message signals that no error occurred and may also contain the data the client requested.

Z Temporary failure. The client should retry their request later.

D Permanent failure.

In a case of a failure the message should include a human-readable description right after the first byte.

More specifically regarding the positive responses, a response to the **H** message is either empty or contains a list of extensions supported by the server delimited with a space (U+0020). An extension's name may be at most 31 bytes long and must not contain a space (U+0020), otherwise any UTF-8 valid byte sequence is allowed, although usage of uppercase ASCII letters and digits is recommended. Proprietary extensions' names must begin with ASCII character "X". Empty extension names are not allowed; this also includes an extension name "X", which is therefore treated as empty.

A positive response to the **Q** message contains the data corresponding to the given cryptoaddress within the specified service.

5.3 Example of a Query

In the following example, "C >: " denotes a message sent by a client and "< S: " denotes a reply sent by a server. Only the content of cryptographic boxes is shown. The client requests a bitcoin address for a cryptoaddress *johndoe@example.com.*

```
C >: 1:H,
< S: 1:O,
C >: 92:Q19:johndoe@example.com,64:000000000019d6689c085ae165831e934ff763
     ae46a2a6c172b3f1b60a8ce26f,,
< S: 35:O1NS17iag9jJgTHD1VXjvLCEnZuQ3rJDE9L,
```

6 Security Considerations

Incorporation of several components potentially opens more possibilities for attacks. We are therefore discussing security impact of each component.

On network level, a connection to and communication with a cryptoaddress server is secured by authenticated encryption. The server's public key is known to the client before initiation of the connection. Therefore, the communication is safe from intercepting and tampering attacks. However, before a client connects to a cryptoaddress server, they queries a DNS server in order to obtain a public key and an IP address of the cryptoaddress server. Unless secured, the DNS traffic is unencrypted and forgeable. Therefore, we put as a requirement to use DNSSEC to sign records leading to the SRV records and associated A/AAAA records, including these. Additionally, we recommend usage of DNSCurve to provide users with confidentiality of their DNS queries [7].

Unless a cryptoaddress server is operated by the same person or company who uses it for their cryptoaddresses, a certain amount of trust in the server's operator is needed. The risk is that the operator could replace user's cryptographic identifier with their own one or one of some colluding party. For static cryptoaddress mappings a monitoring can be setup to detect potential dishonesty of the operator. However, there is no general solution for this issue yet.

Special considerations are needed for cryptoaddresses themselves. Threats related to them can be generalised to spoofing.

6.1 Cryptoaddress Spoofing

An attacker's goal we mainly consider here is mimicking cryptoaddresses in order to gain ability to forge cryptographic identifiers connected to the cryptoaddresses without the need to actually attack the cryptoaddress server.

Primary problem are visually similar characters, e. g., "1" (digit one) and "l" (lowercase L) in ASCII or "c" (Latin lowercase C; U+0063) and "с" (Cyrillic lowercase es; U+0441) in Unicode. There is no straightforward solution to this issue. Indeed, a cryptoaddress formed by the alternative characters (e. g., "john@examp1e.org" with lowercase L replaced by digit one) might be the authentic cryptoaddress and not a forgery.

Local part of a cryptoaddress may be further affected by the permission to include any valid UTF-8 character. Unicode enables some characters to be either formed by a single Unicode character or by a combination of two or more Unicode characters [6]. An example of such a character is the German "ä": this is either a single character U+00E4 or a combination of "a" and the combining diacritical mark umlaut—U+0061 and U+0308.

Furthermore, special or unprintable characters could be inserted into an otherwise legitimate cryptoaddress. Client applications should not interpret these characters and instead signalise their presence in the cryptoaddress. This is already a common practice with today's GUI applications.

Note that the risk of spoofing of cryptoaddresses, e. g., in the context of e-mails or web pages, is comparable with the risk of spoofing of cryptographic identifiers which a particular cryptoaddress represents, although spoofing of the original identifiers might be easier due to their random-looking nature [1].

7 Future Work

In this paper we present the concept of cryptoaddresses in its basic form. Future work will focus on possible extensions, e. g., providing various information to authenticated users. In case of cryptocurrencies this may include information about index of the last issued dynamic address.

Another challenge, which is not yet covered, is the unsecured nature of information provided by a cryptoaddress server. An efficient method of preventing the server from being able to undetectably forge sensitive information, would reduce the level of trust in the server's operator that is currently needed.

8 Conclusion

Many systems whose central features include usage of cryptography to secure their functioning present users with long random-looking identifiers. These identifiers are supposed to be shared and to identify users or their resources within the systems. To improve usability of such systems, we propose in this paper a decentralised system of cryptoaddresses which resemble well-known e-mail addresses. The system enables cryptoaddresses to be mapped back to the original identifiers.

Our proposed system includes leveraging of the existing DNS system secured by its DNSSEC extension; implicit distribution of public keys of servers; and a design of a simple communication protocol between a cryptoaddress client and a server, based on top of a secure communication channel. Although primarily aimed at cryptocurrencies, the presented system is flexible, extensible and can be used for a variety of systems with cryptographic identifiers.

References

1. Andreas M. Antonopoulos (2015): *Mastering Bitcoin*, first edition. O'Reilly, Sebastopol CA.
2. Daniel J. Bernstein (1997): *Quick Mail Transfer Protocol (QMTP).* Available at `https://cr.yp.to/proto/qmtp.txt`.
3. Daniel J. Bernstein (2006): *Curve25519: New Diffie-Hellman Speed Records.* In: *Public Key Cryptography – PKC 2006*, 3958, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 207–228, doi:`10.1007/11745853_14`.
4. Robert Braden (1989): *RFC 1123: Requirements for Internet Hosts – Application and Support.* Available at `https://tools.ietf.org/html/rfc1123`.
5. Daniel Cawrey (2014): *Keybase Project Plans to Make Cryptography as Easy as Twitter.* Available at `http://www.coindesk.com/keybase-project-plans-make-cryptography-easy-twitter/`.

6. Mark Davis, Michel Suignard & et al. (2014): *Unicode Security Considerations.* Technical Report 36. Available at `http://www.unicode.org/reports/tr36/`.
7. Matthew Dempsky (2010): *DNSCurve: Link-Level Security for the Domain Name System.* Available at `https://tools.ietf.org/html/draft-dempsky-dnscurve-01`.
8. Firstbits (2013): *About Firstbits.* Available at `http://firstbits.net/about.php`.
9. Gravatar (2014): *About Crypto-currencies and Your Gravatar Profile.* Available at `http://en.gravatar.com/support/profile-crypto-currency/`.
10. Arnt Gulbrandsen, Paul Vixie & Levon Esibov (2000): *RFC 2782: A DNS RR for specifying the location of services (DNS SRV).* Available at `https://tools.ietf.org/html/rfc2782`.
11. Simon Josefsson (2006): *RFC 4648: The Base16, Base32, and Base64 Data Encodings.* Available at `https://tools.ietf.org/html/rfc4648`.
12. John C Klensin (2010): *RFC 5890: Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework.* Available at `https://tools.ietf.org/html/rfc5890`.
13. Laurie Law, Alfred Menezes, Minghua Qu, Jerry Solinas & Scott Vanstone (1998): *An Efficient Protocol for Authenticated Key Agreement.* Technical Report, Designs, Codes and Cryptography.
14. Yoav Nir & Adam Langley (2015): *RFC 7539: ChaCha20 and Poly1305 for IETF Protocols.* Available at `https://tools.ietf.org/html/rfc7539`.
15. Peter Saint-Andre (2015): *RFC 7622: Extensible Messaging and Presence Protocol (XMPP): Address Format.* Available at `https://tools.ietf.org/html/rfc7622`.
16. The Monero Project (2014): *OpenAlias.* Available at `https://www.openalias.org/`.

A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker

Dirk Beyer[1] and Karlheinz Friedberger[2]

[1] LMU Munich, Germany
[2] University of Passau, Germany

Abstract. Verifying multi-threaded programs is becoming more and more important, because of the strong trend to increase the number of processing units per CPU socket. We introduce a new configurable program analysis for verifying multi-threaded programs with a bounded number of threads. We present a simple and yet efficient implementation as component of the existing program-verification framework CPAchecker. While CPAchecker is already competitive on a large benchmark set of sequential verification tasks, our extension enhances the overall applicability of the framework. Our implementation of handling multiple threads is orthogonal to the abstract domain of the data-flow analysis, and thus, can be combined with several existing analyses in CPAchecker, like value analysis, interval analysis, and BDD analysis. The new analysis is modular and can be used, for example, to verify reachability properties as well as to detect deadlocks in the program. This paper includes an evaluation of the benefit of some optimization steps (e.g., changing the iteration order of the reachability algorithm or applying partial-order reduction) as well as the comparison with other state-of-the-art tools for verifying multi-threaded programs.

1 Introduction

Program verification has successfully been applied to programs to find errors in applications. There exist many approaches to verify single-threaded programs (cf. SV-COMP for an overview [1]), and several of them are already implemented in the open-source program-verification framework CPAchecker [4, 10]. For multi-threaded programs a new dimension of complexity has to be taken into account: the verification tool has to efficiently analyze all possible thread interleavings. CPAchecker did not support the analysis of multi-threaded programs for a long time. Our work focuses on a new, simple configurable program analysis that reuses several existing components of the framework. The approach is sound and can be combined with several steps of optimization to achieve an efficient analysis for multi-threaded programs.

Our analysis is based on a standard state-space exploration using a given control-flow automaton that represents the program. For a program state with several active threads, we compute the succeeding program state for each of those threads, i.e. basically we compute every possible interleaving of the threads. The approach is orthogonal to other data-flow based analyses in CPAchecker, thus it can be combined with algorithms like CEGAR [7] and analyze an potentially infinite state space.

Related Work. A prototypical version of our analysis was already applied for the category of concurrent programs during the SV-COMP'16 [1]. Due to some unsupported features and missing parts of the optimization that where implemented later, the score in this category was low at that time. The experimental results that we report show that the current version of the implementation performs much better.

Just like several other tools [8, 9, 15], we explore possible interleavings of different thread executions and our optimization methods include partial order reduction [12]. In contrast to verification techniques for multi-threaded programs like constraint-based representation [13] that limits the domain to Horn clauses and predicate abstraction or sequentialization [11, 16] that transforms the program on source-code level before starting the analysis, our approach computes the interleaving of threads on-the-fly and is independent from the applied analysis. This makes it possible to integrate our approach easily with data-flow analyses of different abstract domains, such as value analysis [5] and BDD analysis [6].

2 Analysis of Multi-Threaded Programs in CPAchecker

The following section provides an overview of some basic concepts and definitions used for our approach. We describe the program representation and the details of our configurable program analysis.

2.1 Program Representation

A program is represented by a *control-flow automaton* (CFA) $A = (L, l_0, G)$, which consists of a set L of program locations (modeling the program counter), a set $G \subseteq L \times Ops \times L$ (modeling the control flow with assignment and assumption operations from Ops), and an initial program location l_0 (entry point of the main function).

Let V be the set of variables in the program. The *concrete data state for a program location* assigns a value to each variable from the set V; the set C contains all concrete data states. For every edge $g \in G$, the transition relation is defined by $\xrightarrow{g} \subseteq C \times \{g\} \times C$. The union of all edges defines the complete transfer relation $\rightarrow = \bigcup_{g \in G} \xrightarrow{g}$. If there exists a chain of concrete data states $\langle c_0, c_1, ..., c_n \rangle$ with $\forall c_i$: there exists a program location l_i for which c_i is a concrete data state and $\forall i : 1 \leq i \leq n \Rightarrow \forall i : 1 \leq i \leq n \Rightarrow \exists g : c_{i-1} \xrightarrow{g} c_i \wedge (l_{i-1}, g, l_i) \in G$, then the state c_n is *reachable* from c_0 for l_0.

Our analysis is a reachability analysis and unrolls the program lazily [14] into an *abstract reachability graph* (ARG) [2]. The ARG is a directed acyclic graph that consists of abstract states (representing the abstract program state, e.g., including program location and variable assignments) and edges modeling the transfer relation that leads from one abstract state to the next one.

2.2 ThreadingCPA

CPAchecker is based on the concept of *configurable program analysis* (CPA) [3]. Thus, different aspects of a program are analyzed by different components (denoted as CPAs). A default analysis in CPAchecker [4] uses the LocationCPA to track the program location (program counter) and the CallstackCPA to track call stacks (function calls and their corresponding return location in the CFA). Thus, for the analysis of sequential programs, each abstract state that is reached during an analysis consists of exactly one program location and one call stack.

For the analysis of multi-threaded programs we have developed a new ThreadingCPA that replaces both the LocationCPA and the CallstackCPA and explores the state space of a multi-threaded program on-the-fly. The benefit of the ThreadingCPA is that it is able to track several program locations (one per thread) together with their call stacks (also one per thread). For simplicity of the definition we ignore the handling of call stacks in the next section. The reader can simply assume that for each program location there is also a call stack. The ThreadingCPA has to handle multiple call stacks (one per thread), whereas the CallstackCPA only handles a single call stack.

The definition of the ThreadingCPA $\mathbb{T} = (D_{\mathbb{T}}, \rightsquigarrow_{\mathbb{T}}, merge_{\mathbb{T}}, stop_{\mathbb{T}})$ follows the structure of a configurable program analysis:
Domain: The abstract domain $D_{\mathcal{T}} = (C, \mathcal{T}, [[\cdot]])$ is a triple of the set C of concrete states, the flat semi-lattice $\mathcal{T} = (T, \sqsubseteq, \sqcup, \top)$, and the concretization function $[[\cdot]] : \mathcal{T} \to 2^C$. Let I be the set of all possible thread identifiers, e.g., a set of names used to identify threads in the program. The type of abstract states $T : I \nrightarrow \mathcal{L}$ consists of all assignments of thread identifiers $t \in I$ to program locations $l \in \mathcal{L} = L \cup \{\top_L\}$. The special program location $\top_L$ represents an unknown program location. The top element $\top \in T$, with $\top(t) = \top_L$ for all $t \in I$, is the abstract state that holds no specific program location for any thread identifier. Each abstract threading state $s \in T$ is represented by the assignments $\{t_1 \mapsto l^{t_1}, t_2 \mapsto l^{t_2}, ...\}$ of thread identifiers to their current program location. The partial order $\sqsubseteq$ induces a semi-lattice for the abstract states. The join operator $\sqcup$ yields the least upper bound of given abstract states. The top element $\top$ of the semi-lattice is defined as $\top = \sqcup T$.
Merge: The ThreadingCPA uses the merge operator $merge_{sep}$, which does not combine different elements.
Stop: The ThreadingCPA uses the termination operator $stop_{sep}$, which defines coverage only in case of equal abstract states.
Transfer: The transfer relation $\rightsquigarrow_{\mathbb{T}}$ determines the syntactic successor for the current state and is based on the transfer relation of the LocationCPA. The implementation is simple: The transfer relation returns all possible successors for all threads that are active in an abstract state, i.e., it applies the transfer relation of the LocationCPA for each active thread. Additionally, thread-management-related operations are included, such that creating or joining threads (when calling *pthread_create* or *pthread_join*) is defined. It is in theory sufficient to only handle these two function calls, because other thread-related function calls do not change the number of threads or the progress of the state-space exploration.

The transfer relation $\rightsquigarrow_{\mathbb{T}}$ has the transfer $s \overset{g}{\rightsquigarrow} s'$ for two abstract states $s = \{t_1 \mapsto l^{t_1}, t_2 \mapsto l^{t_2}, ..., t_N \mapsto l^{t_N}\}$ and $s' = \{t_1 \mapsto l'^{t_1}, t_2 \mapsto l'^{t_2}, ..., t_N \mapsto l'^{t_M}\}$ and $g = (l^{t_i}, op, l'^{t_i})$ if

1. the operation *op* matches the *pthread_create* statement for t_i that is in program location l^{t_i} and creates a new thread t_{new} starting from a CFA node $l_0^{t_{new}} \in L$:
$$s' = s \setminus \{t_i \mapsto l^{t_i}\} \cup \{t_{new} \mapsto l_0^{t_{new}}\} \cup \{t_i \mapsto l'^{t_i}\}$$
i.e., an existing thread t_i matches the program location l^{t_i} and moves along the edge g towards program location l', and the initial program location $l_0^{t_{new}}$ of the new thread t_{new} is added to the current abstract state.
2. the operation *op* matches the *pthread_join* statement for t_i that is in program location l^{t_i} and waits for a thread t_{exit} to exit, t_{exit} exits at program location $l_E^{t_{exit}}$, and $t_{exit} \mapsto l_E^{t_{exit}} \in s$:
$$s' = s \setminus \{t_i \mapsto l^{t_i}\} \setminus \{t_{exit} \mapsto l_E^{t_{exit}}\} \cup \{t_i \mapsto l'^{t_i}\}$$
i.e., an existing thread t_i matches the program location l^{t_i} and moves along the edge g towards program location l'^{t_i}, and the program location $l_E^{t_{exit}}$ of the thread t_{exit} is removed from the current abstract state, if the thread t_{exit} has already been at this program location.
3. otherwise, if the operation *op* is not related to thread management:
$$s' = s \setminus \{t_i \mapsto l^{t_i}\} \cup \{t_i \mapsto l'^{t_i}\}$$
i.e., thread t_i matches the program location l^{t_i} and moves along the edge towards l'^{t_i}.

For a basic analysis for multi-threaded programs the handling of the operations *pthread_create* and *pthread_join* is sufficient. Additional thread management like mutex locks (details in Section 4.3) can be applied on top of this transfer relation. We assume C statements as atomic statements, i.e., interleaving of threads is considered to happen on statement level (matching the encoding of the program as CFA). This might be insufficient for real-world programs, but is good enough for several examples and in theory the CFA could be inflated with read and write operations for memory registers.

2.3 Example

The following example applies our new ThreadingCPA to a given program. In contrast to the simplified illustration below, a real-world analysis would combine the ThreadingCPA with another analysis, e.g., to track assignments, such as value analysis or BDD analysis.

The example program (cf. Fig. 1 for the source code) creates two additional threads that change the value of global variables. Afterwards, the main method checks the assignment of a global variable. In this example, the property holds.

```
pthread_t id1, id2;
int i=1, j=1;

void main() {
  pthread_create(&id1, 0, t1, 0);
  pthread_create(&id2, 0, t2, 0);

  pthread_join(id1, 0);
  pthread_join(id2, 0);

  assert(j <= 8);
}

void t1() {
  i+=j;
  i+=j;
}

void t2() {
  j+=i;
  j+=i;
}
```

Fig. 1: Program with concurrent threads

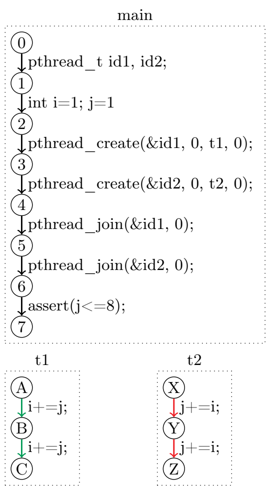


Fig. 2: CFA for the functions of the program

The program's functions are represented as CFAs in Figure 2. The ThreadingCPA produces the ARG in Fig. 3, where each abstract state is labeled with the indices of the program locations of all active threads.

The analysis starts at entry location l_0 of the main function and analyzes all possible interleavings. After reaching the statement *pthread_create*, an additional program location is tracked for the newly created thread, e.g., when reaching program location l_3 in the main function, the abstract state is enriched with the initial program location l_A of the newly created thread.

As the ThreadingCPA merges its abstract states when reaching the same program locations via different execution paths, the diamond-like structure in the ARG is the result of interleaved thread-execution of two (or more) threads. When exploring the statement *pthread_join*, the program-exit location of the exiting thread is removed from the abstract state. This is visible in Fig. 3 for each abstract state with an outgoing edge leading from program location l_4 towards program location l_5, because the program-exit location l_C of the joining thread t_1 (identified by id 1) is removed from the abstract state.

3 Optimization

The simple definition of the ThreadingCPA allows (and needs) a wide range of optimization to gain competitive efficiency. In the following, we define some approaches and show how fluently they match existing concepts in CPAchecker.

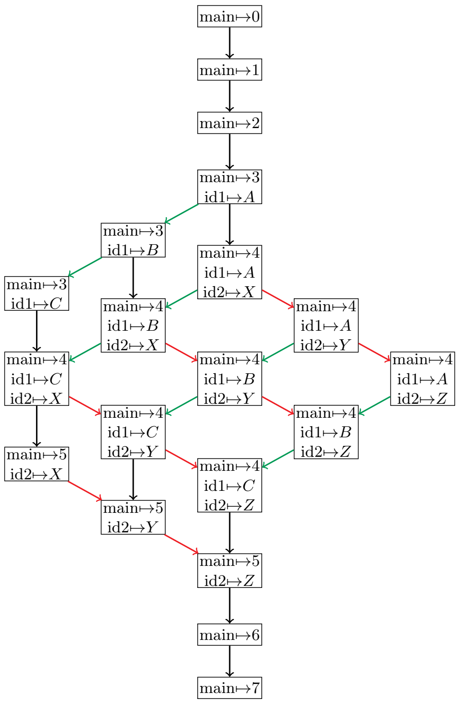


Fig. 3: ARG of the interleaved threads of the program

3.1 Partitioning of Reached Abstract States

The reachability algorithm [3] has two important operators *merge* and *stop* that are defined as operations on sets of reached abstract states. These operations can merge abstract states and combine their information into a new abstract state or detect coverage, i.e., an abstract state is implied by another one and thus the exploration can stop at that point. In each iteration of the reachability algorithm, these operators are by default applied to *all* combinations of new explored abstract states and previously reached abstract states. However, applying such an operator to *all* previously reached abstract states is inefficient, because most of the abstract states are irrelevant for a concrete application of these operators. For example, comparing abstract states from different program locations is useless, because there will not be any important relation between them.

Partitioning the set of abstract states makes it possible to perform both operations much more efficiently, as only a (small) subset of the previously reached abstract states has to be considered in the computation. This basic optimization is also applied for verifying single-threaded programs. Each partition is identified by a constant key that is based on the program location of the abstract state, as only states from equal program locations are considered for merging or coverage. We extended the existing partitioning of abstract states, such that it uses the tuple of program locations for all threads in an abstract state. This new partitioning can also be combined with partitionings provided by other CPAs.

3.2 Waitlist Order

For finding property violations it is often sufficient to only analyze interleavings with a low number of thread interleavings. As the exploration algorithm in CPAchecker analyzes the reachable state space state by state, there exists the possibility to *prioritize abstract states* during the exploration: The abstract states waiting to be analyzed are simply sorted by some criteria. This optimization is a heuristic depending on the internal structure of the analyzed program and the executed analysis. For a bug-free program this heuristic does not bring any benefit. however an existing error path in a faulty program might be found sooner.

The most-often used orderings of abstract states cause the state-space exploration to perform either depth-first search (DFS) or breadth-first search (BFS), i.e., the list of waiting abstract states is ordered in the same manner as abstract states are explored (BFS) or reverse (DFS). For multi-threaded programs, we added a new ordering of this list based on the number of active threads, such that states with fewer active threads are considered first. The new ordering can also be combined with existing orderings, i.e., the first criteria for ordering is based on the number of active threads, the second criteria uses the exploration order.

3.3 Partial-Order Reduction

With multi-threaded programs, the most common form of optimization is *partial-order reduction* (POR) [12, 18, 19]. POR aims to avoid unnecessary interleavings of threads and improves the performance of the analysis by reducing the explored state space. However, its application depends on the property to be verified, because all necessary program paths must remain reachable.

In our case (reachability analysis), we started with a simple separation of program operations (modeled as CFA edges) into *thread-local* and *global* operations. We conservatively apply a static analysis for all program variables and memory accesses, on whether they are declared and used in *global* scope or only *locally* in the context of a thread. Because CPAchecker uses several dummy operations (e.g., for temporary variables or function returns), a majority of CFA edges is marked as *thread-local*.

If a statement is *thread-local* for a thread, we do not simulate any interleaving after analyzing this operation, but the analysis executes the current thread further, until a global operation (in the same thread) is reached. This behavior is *sound*, because no interaction between threads is possible, due to the definition of *thread-local* operations. Thus, we only need to synchronize all available threads after the next *global* transition.

Our approach can analyze program with loops as well, because we execute both paths, i.e., the loop and the concurrent thread, and none of them disables the other path. Thus, any possible interaction between CFA edges of the loop and other threads is considered. Our approach handles loops implicitly, thus we do not have to actively check for loops, but simply apply the reachability algorithm combined with the described POR technique.

4 Extensions

During our work on the analysis of multi-threaded programs, we explored some assumptions in CPAchecker that need to be considered when integrating such a basic analysis as the ThreadingCPA. We also noticed several features that can also be specified or implemented for the analysis of multi-threaded programs. In the following, we describe the extensions that we have developed in order to use the full potential of the framework.

4.1 Cloning for CFAs

CPAchecker has a modular structure, such that many components can be combined without knowing (and depending on) details about each other. As the analysis of multi-threaded programs should fit into this design, we decided not to modify each analysis that should be combined with our new approach, but use an approach that allows us to re-use as much existing code as possible.

The basic problem with the existing components of CPAchecker is that many of them rely on knowing only their current function scope, and solely identify a variable by its name combined with the name of the function scope it was declared in. For example, many analyses (including value analysis and BDD analysis) use the identifier $f{::}x$ for a variable x declared in function f. This identifier is used in the internal data structures whenever the variable is used during the program analysis. In a multi-threaded program, the same function f might be called in different threads, such that $f{::}x$ is not unique for one variable any more at a certain point in the program's execution. The existing analyses do not know about two variables with the same identifier and would, e.g., assign a wrong value to one of them.

Our solution is simple: We use different function names for each thread by *cloning* the function and inserting the corresponding indexed function name. For a function f we create a clone f' by copying the corresponding CFA nodes from L and edges from G, while renaming all appearances of the function's identifier in the clone. Cloning functions causes all function-local variables to be unique for different threads in the later applied analysis, e.g., the identifier $f{::}x$ is distinct from $f'{::}x$. An analysis using the identifier does not even have to know whether the function is cloned and can simply assume uniqueness of identifiers for all variables.

4.2 Deadlock Detection

A deadlock [17] is defined as an abstract state where two (or more) competing actions wait for each other to finish, and thus neither ever does. CPAchecker allows the user to define the goal of an analysis by giving a specification in form of an automaton. Detecting deadlocks in the program can be done by giving an observer automaton that monitors the abstract states of the ThreadingCPA and reports deadlocks. This approach is independent of any further analysis and can be combined with, e.g., value analysis or BDD analysis.

4.3 Mutex Locks

Mutex locks are commonly used to synchronize threads, e.g., to manage access to shared memory. In our implementation, mutex locks are stored as part of the abstract state of the ThreadingCPA. If a mutex lock is requested along a CFA edge, but not available in the preceding abstract state, the transfer relation does not yield a successive abstract state for the CFA edge.

Additionally, we use mutex locks for more use cases: We simulate atomic sequences of statements and some aspects of partial order reduction as mutex locks in the ThreadingCPA. Entering an atomic sequence requires an *atomic mutex lock*, which is released after leaving the atomic sequence. Consecutive CFA edges containing only thread-local operations (see Section 3.3) are modeled and analyzed as atomic sequence.

5 Evaluation

In this section we evaluate different configurations of the ThreadingCPA and compare it with other state-of-the-art tools. The evaluation is performed on machines with a 2.6 GHz Intel Xeon E5-2650 v2 CPU running Ubuntu 16.04 (Linux 4.4.0). Each single verification run is limited to 15 min of run time and 15 GB of memory. The 1 016 benchmark tasks are taken from the category of multi-threaded programs at SV-COMP'16 [3]. The tasks are C programs, where reaching a specific function call is considered as property violation. We use CPAchecker [4] 1.6.1 in revision 23 011.

5.1 Optimization Steps

First, we show the effect of applying each optimization step from Section 3 successively, i.e., on top of the previous optimization. Starting with a plain (non-optimized) configuration of the ThreadingCPA combined with the value analysis, we step-wise apply optimization in form of

– reached-set partitioning (see Section 3.1) based on the abstract states,
– waitlist ordering (see Section 3.2) based on the number of threads, and
– POR (see Section 3.3) based on *local-scope* and *global* statements.

The optimization steps are independent of the value analysis and can also be applied to any other analysis like BDD analysis and interval analysis, where

[3] `https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp16`
[4] `https://cpachecker.sosy-lab.org/`

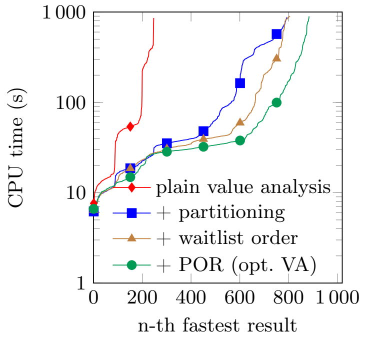


Fig. 4: Quantile plot for different configurations of the value analysis, corresponding to step-wise applied optimization steps

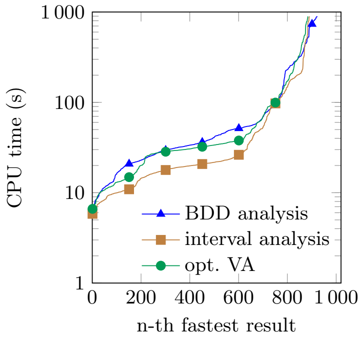


Fig. 5: Quantile plot for different abstract domains using the ThreadingCPA within CPAchecker

the same benefit will be visible. Figure 4 shows a quantile plot containing the run time of correctly solved verification tasks. The evaluation shows that the verification process benefits from each of the optimization steps. For small tasks that can be verified within a few second, e.g., because of only a few thread interleavings in the program, the benefit of optimization is small. For tasks that need more run time the benefit becomes visible.

We noticed that the heuristic of ordering the waiting abstract states is beneficial in two ways: first, some property violations are found earlier (some property violations need only a small number of interleavings); second, some unsupported operations (like assigning several thread instances to the same thread identifier) are reached earlier and the analysis can abort immediately without wasting time.

Compared to the plain value analysis, partitioning the reached set improves the performance and reduces the run time of the analysis by more than an order of magnitude. Additionally changing the waitlist order improves run time in several cases, mostly for tasks with a property violation. However, in our benchmark this optimization step does not lead to more correctly solved tasks. POR causes a lower number of explored abstract states, and thus the performance increases.

5.2 Abstract Domains

Second, we combine the ThreadingCPA with different analyses, such as value analysis, interval analysis, and BDD analysis, which are already implemented in the CPAchecker framework and are normally used for the analysis of single-threaded programs. We only evaluate the optimized version of each combination. The analyses could also be combined with CEGAR [7], however the current benchmark does not benefit from it, and thus we just execute a reachability algorithm to verify the specification. We show that we can verify the majority of benchmark programs and discuss strengths and weaknesses of the analyses. As all compared analyses use the same framework (parser, algorithm, ...), we expect our evaluation to be fair for all implemented approaches and allow a precise comparison. Figure 5 shows the quantile plot of correct results for the combinations of the ThreadingCPA with other analyses.

The BDD analysis is optimized for BFS in the reachability algorithm, whereas value analysis and interval analysis use DFS as basic order for the list of waiting abstract states during the exploration algorithm (see Section 3.2). Thus, the state-space exploration traverses program locations and thread interleavings in another order and finds the corresponding abstract states in a different order, too. Depending on the verification task, this can result in an in- or decreased performance compared to the value analysis.

5.3 Other Tools

Third, we compare the (optimized) value analysis with two other state-of-the-art verification tools, namely CBMC [5] and VVT [6]. Both tools are executed as in the SV-COMP'16 and are chosen, because they do not apply special approaches like sequentialisation, but rely on a similar state-space exploration technique as our approach in CPAchecker. Figure 6 shows the quantile plot of correct results for CBMC, VVT, and CPAchecker (using the optimized value analysis). The ThreadingCPA (combined with value analysis) is competitive with the other tools. The plot for CPAchecker matches the trend of the other tools with only some differences. At the left side of the plot the initial start-up time of a few seconds for CPAchecker is visible, whereas other tools already solve some of the given instance within this time.

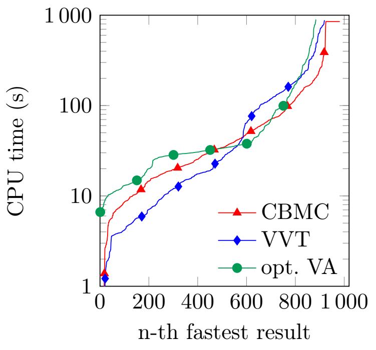


Fig. 6: Quantile plot for comparison of other verifiers with support for multi-threaded programs

Due to the missing support for pointer aliasing and array computations in the value analysis as well as due to our simple kind of POR, CPAchecker can not solve as many verification tasks as other tools within the time limit.

[5] `http://www.cprover.org/cbmc/` [6] `https://vvt.forsyte.at/`

6 Conclusion

This paper presents a basic approach to support the analysis of multi-threaded programs in CPAchecker. We formally defined a new ThreadingCPA in the framework and demonstrated that several core components can be reused. Reusing existing analyses is possible without any further overhead. Due to our simple approach, there are a few limitations that have to be considered when verifying multi-threaded programs with CPAchecker. Our approach for partial order reduction is simple and can be extended with more advanced techniques to further reduce the number of explored abstract states. The maximum number of threads is bounded, because of possible conflicts in function names. To avoid naming conflicts, we clone each function's CFA several times before starting the analysis. The number of clones cannot be changed afterwards. If we run out of clones during the analysis and would need more due to a naming conflict, we abort the analysis and report an insufficient number of threads.

As the ThreadingCPA identifies each thread only by the variable it is assigned to, we currently can not analyze more complex thread management such as pointer aliasing for the thread identifier or more complex locking mechanisms. Our framework already contains a mechanism for exchanging information between abstract states on a state-level during the analysis. The analysis of multi-threaded programs could be extended to exchange information about thread management with another analysis capable of such data, such that we could analyze more difficult thread management with the ThreadingCPA.

Possible ideas for optimization have been implemented and evaluated. The evaluation shows that the results of different analyses based on the ThreadingCPA are competitive with other state-of-the-art tools.

References

1. D. Beyer (2016): *Reliable and Reproducible Competition Results with* BenchExec *and Witnesses.* In: *Proc. TACAS*, Springer, pp. 887–904, doi:10.1007/978-3-662-49674-9_55. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2016-TACAS.Reliable_and_Reproducible_Competition_Results_with_BenchExec_and_Witnesses.pdf`.
2. D. Beyer, T. A. Henzinger, R. Jhala & R. Majumdar (2007): *The Software Model Checker* Blast. *Int. J. Softw. Tools Technol. Transfer* 9(5-6), pp. 505–525, doi:10.1007/s10009-007-0044-z. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2007-STTT.The_Software_Model_Checker_BLAST.pdf`.
3. D. Beyer, T. A. Henzinger & G. Théoduloz (2007): *Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis.* In: *Proc. CAV*, LNCS 4590, Springer, pp. 504–518, doi:10.1007/978-3-540-73368-3_51. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2007-CAV.Configurable_Software_Verification.pdf`.
4. D. Beyer & M. E. Keremoglu (2011): CPAchecker*: A Tool for Configurable Software Verification.* In: *Proc. CAV*, LNCS 6806, Springer, pp. 184–190, doi:10.1007/978-3-642-22110-1_16. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2011-CAV.CPAchecker_A_Tool_for_Configurable_Software_Verification.pdf`.

5. D. Beyer & S. Löwe (2013): *Explicit-State Software Model Checking Based on CEGAR and Interpolation.* In: *Proc. FASE*, LNCS 7793, Springer, pp. 146–162, doi:10.1007/978-3-642-37057-1_11. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2013-FASE.Explicit-State_Software_Model_Checking_Based_on_CEGAR_and_Interpolation.pdf`.
6. D. Beyer & A. Stahlbauer (2013): *BDD-Based Software Model Checking with* CPAchecker. In: *Proc. MEMICS*, LNCS 7721, Springer, pp. 1–11, doi:10.1007/978-3-642-36046-6_1. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2013-MEMICS.BDD-Based_Software_Model_Checking_with_CPAchecker.pdf`.
7. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu & H. Veith (2003): *Counterexample-guided abstraction refinement for symbolic model checking. J. ACM* 50(5), pp. 752–794, doi:10.1145/876638.876643.
8. E. M. Clarke, D. Kröning, N. Sharygina & K. Yorav (2005): SatAbs*: SAT-Based Predicate Abstraction for ANSI-C.* In: *Proc. TACAS*, LNCS 3440, Springer, pp. 570–574, doi:10.1007/978-3-540-31980-1_40.
9. L. Cordeiro & B. Fischer (2011): *Verifying multi-threaded software using smt-based context-bounded model checking.* In: *Proc. ICSE*, ACM, pp. 331–340, doi:10.1145/1985793.1985839.
10. Matthias Dangl, Stefan Löwe & Philipp Wendler (2015): CPAchecker *with Support for Recursive Programs and Floating-Point Arithmetic.* In: *Proc. TACAS*, LNCS 9035, Springer, pp. 423–425, doi:10.1007/978-3-662-46681-0_34.
11. B. Fischer, O. Inverso & G. Parlato (2013): *CSeq: A Sequentialization Tool for C (Competition Contribution).* In: *Proc. TACAS*, LNCS 7795, Springer, pp. 616–618, doi:10.1007/978-3-642-36742-7_46.
12. Patrice Godefroid (1996): *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* LNCS 1032, Springer, doi:10.1007/3-540-60761-7.
13. A. Gupta, C. Popeea & A. Rybalchenko (2011): *Threader: A Constraint-Based Verifier for Multi-threaded Programs.* In: *CAV*, LNCS 6806, Springer, pp. 412–417, doi:10.1007/978-3-642-22110-1_32.
14. T. A. Henzinger, R. Jhala, R. Majumdar & G. Sutre (2002): *Lazy abstraction.* In: *Proc. POPL*, ACM, pp. 58–70, doi:10.1145/503272.503279.
15. G. J. Holzmann (1997): *The* Spin *Model Checker. IEEE Trans. Softw. Eng.* 23(5), pp. 279–295, doi:10.1109/32.588521.
16. Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre & Gennaro Parlato (2015): *Lazy-CSeq: A Context-Bounded Model Checking Tool for Multi-threaded C-Programs.* In: *Proc. ASE*, ACM, pp. 807–812, doi:10.1109/ASE.2015.108.
17. Sreekaanth S. Isloor & T. Anthony Marsland (1980): *The Deadlock Problem: An Overview. IEEE Computer* 13(9), pp. 58–78, doi:10.1109/MC.1980.1653786.
18. Doron A. Peled (1993): *All from One, One for All: on Model Checking Using Representatives.* In: *Proc. CAV*, LNCS 697, Springer, pp. 409–423, doi:10.1007/3-540-56922-7_34.
19. Antti Valmari (1989): *Stubborn sets for reduced state space generation.* In: *Proc. Petri Nets*, LNCS 483, Springer, pp. 491–515, doi:10.1007/3-540-53863-1_36.

Anti-Path Cover on Sparse Graph Classes[⋆]

Pavel Dvořák[1], Dušan Knop[2⋆⋆], and Tomáš Masařík[3⋆⋆⋆]

[1] Computer Science Institute of Charles University
Charles University, Prague, Czech Republic
koblich@iuuk.mff.cuni.cz
[2] Department of Applied Mathematics
Charles University, Prague, Czech Republic
knop@kam.mff.cuni.cz
[3] Department of Applied Mathematics
Charles University, Prague, Czech Republic
masarik@kam.mff.cuni.cz

Abstract. **Abstract.** We show that it is possible to use Bondy-Chvátal closure to design an FPT algorithm that decides whether or not it is possible to cover vertices of an input graph by at most k vertex disjoint paths in the complement of the input graph.
More precisely, we show that if a graph has tree-width at most w and its complement is closed under Bondy-Chvátal closure, then it is possible to bound neighborhood diversity of the complement by a function of w only.
A simpler proof where tree-depth is used instead of tree-width is also presented.

1 Introduction

Graph Hamiltonian properties are studied especially in connection with graph connectivity properties. A graph is called *Hamiltonian* if there is a path passing through all its vertices in that graph. In this work we are interested in sparse graph setting for which this question was already solved e.g. using the famous theorem of Courcelle [4]. It is possible to express the Hamiltonian property by an MSO_2 formula and thus resolve the question by an FPT algorithm. For a graph G and a positive integer k we say that G is *k-path coverable* if there exists a collection of at most k vertex disjoint paths in G such that the vertices of G are the union of vertices of all paths in the collection (that is, each vertex belongs to exactly one path in the collection). We give a simple, but interesting, twist to the question to rise a new problem:

⋆ The work was supported by the project SVV-2016-260332.
⋆⋆ Author was supported by the project GAUK 1784214 and by the project CE-ITI P202/12/G061
⋆⋆⋆ Author was supported by the project GAUK 338216 and by the project CE-ITI P202/12/G061.

k-Anti-Path Cover	
Input:	A graph G and positive integer k.
Question:	Is the complement of graph G k-path coverable?

In this notation Hamiltonian Anti-Path problem is exactly the 1-Anti-Path Cover problem. It is not hard to see that we may focus on solving the Hamiltonian Anti-Path as the k-Anti-Path Cover problem is reducible to the Hamiltonian Anti-Path by addition of k isolated vertices (apex vertices in the complement graph). Note that this does not affect tree-width nor tree-depth. Indeed, for unrestricted graphs this question is solved on complement of the graph. However, this is not possible if input graphs are restricted so that some specified parameter is bounded. Two interesting examples are tree-width and tree-depth. Note that tree-width of a complement of a graph cannot be bounded by a function of the tree-width of the graph. On the contrary, there are graph parameters in whose this question was already solved, as these graph parameters are (for each constant) closed under taking complements – neighborhood diversity and modular width, to name just a few. On both these parameters k-Path Cover was one of the first considered problems which was showed to be in the FPT class [6], [8] respectively.

There is a strong connection between the $L(2,1)$-labeling problem and Hamiltonian Anti-Path. $L(2,1)$-labeling is a labeling of vertices of a graph G so that labels of vertices in distance 1 differ by at least 2, while labels of vertices in distance 2 differ by at least 1. The labels are taken from set $\{0,\dots,\lambda\}$ and λ is called a span. It not hard to see that a graph G with an apex vertex added admits $L(2,1)$-labeling with span $\lambda = n$ if and only if G has Hamiltonian Anti-Path [2].

Our Contribution

Theorem 1. *Let G be a graph of tree-width at most w. The problem of k-Anti-Path Cover admits an FPT algorithm parameterized by tree-width.*

The proof uses the famous closure theorem of Bondy and Chvátal. Most notably we prove the following theorem from which we can derive the previous theorem using e.g. known FPT algorithm of Lampis [8]. This exploit an unexpected relation between tree-width and neighborhood diversity. Thus, it naturally rises many questions – whether similar approach is admissible for other problems besides k-Anti-Path Cover problem.

Theorem 2. *Let G be a graph of tree-width w and further complement $\overline{G}$ closed under Bondy-Chvátal closure. It follows that neighborhood diversity of $\overline{G}$ is bounded by $2^k + k$ where $k = 2(w^2 + w)$.*

Furthermore, we give a natural specializations of the theorems above in Section 3. The purpose of Section 3 is twofold – first as tree-depth is more restrictive parameter than tree-width the proof is simpler and second, the analysis of a particular application of Bondy-Chvátal theorem gives more light on the structure of the complement of a graph G that is closed under this closure operator.

2 Preliminaries

One of the basic graph operations is taking the complement of a graph. A *complement* of a graph $G = (V, E)$ is denoted by $\overline{G}$ and it is the graph $(V, \binom{V}{2} \setminus E)$. Throughout the paper we denote by n the number of vertices in the input graph. By the distance between two vertices u, v we mean the length of the shortest path between them in the assumed graph G. We denote the distance by $\mathrm{d}_G(u, v)$ and omit the subscript if the graph is clear from the context. We extend the notion to sets of vertices in a straightforward manner, that is $\mathrm{d}_G(U, W) = \min\{\mathrm{d}_G(u, w) \colon u \in U, w \in W\}$. For further graph related notation we refer reader to the monograph by Diestel [5].

In our approach we repeatedly use the closure theorem of Bondy and Chvátal to increase the number of edges in the complement of a graph (i.e. to reduce the number of edges in the given graph). Note that this operation does not increase the tree-width of the input graph.

Theorem 3 (Bondy-Chvátal closure [3]). *Let $G = (V, E)$ be a graph of order $|V| \geq 3$ and suppose that u and v are distinct non-adjacent vertices such that* $\deg(u) + \deg(v) \geq |V|$. *Now G has a Hamiltonian path if and only if $(V, E \cup \{u, v\})$ has a Hamiltonian path.*

The notion of *tree-width* was introduced by Bertelé and Brioshi [1].

Definition 1 (Tree decomposition). *A* tree decomposition *of a graph G is a pair (T, X), where $T = (I, F)$ is a tree, and $X = \{X_i \mid i \in I\}$ is a family of subsets of $V(G)$ (called bags) such that:*

– *the union of all X_i, $i \in I$ equals V,*
– *for all edges $\{v, w\} \in E$, there exists $i \in I$, such that $v, w \in X_i$ and*
– *for all $v \in V$ the set of nodes $\{i \in I \mid v \in X_i\}$ forms a subtree of T.*

The *width* of the tree decomposition is $\max(|X_i| - 1)$. The *tree-width* of a graph $\mathrm{tw}(G)$ is the minimum width over all possible tree decompositions of the graph G.

Proposition 1 ([7]). *Let G be a graph with n vertices. There exists an optimal tree decomposition with $O(n)$ bags. Moreover, there is an* FPT *algorithm that finds such a decomposition.*

Definition 2 (Tree-depth [9]). *The closure $Clos(F)$ of a forest F is the graph obtained from F by making every vertex adjacent to all of its ancestors. The* tree-depth*, denoted as* $\mathrm{td}(G)$*, of a graph G is one more than the minimum height of a rooted forest F such that $G \subseteq Clos(F)$.*

The last graph parameter needed in this work is the *neighborhood diversity* introduced by Lampis [8].

Definition 3 (Neighborhood diversity). *The* neighborhood diversity *of a graph G is denoted by* $\mathrm{nd}(G)$ *and it is the minimum size of a partition of vertices into classes such that all vertices in the same class have the same neighborhood, i.e. $N(v) \setminus \{v'\} = N(v') \setminus \{v\}$, whenever v, v' are in the same class.*

It can be easily verified that every class of a neighborhood diversity partition is either a clique or an independent set. Moreover, for every two distinct classes C, C', either every vertex in C is adjacent to every vertex in C', or there is no edge between C and C'. If classes C and C' are connected by edges, we refer to such classes as *adjacent*.

It is possible to find the optimal neighborhood diversity decomposition of a given graph in polynomial time [8].

3 Tree-depth

We show that using the Bondy-Chvátal closure it is possible on input G and k either decide that $\overline{G}$ is k-path coverable or return graph $\overline{H}$ that is equivalent (for the k-path coverability) to graph $\overline{G}$ with $\mathrm{nd}(\overline{H}) \leq 2^{2d} + 2d$. Furthermore, it is possible to use the FPT algorithm of Lampis [8] on the resulting graph to decide whether it is k-path coverable or not. This in turn gives an FPT algorithm for k-Anti-Path Cover with respect to tree-depth.

Applying Bondy-Chvátal. We will apply the Bondy-Chvátal closure from the leaves of a tree-depth decomposition of the input graph G in order to reduce the number of edges in G and either resolve the given question (in the case G becomes an edgeless graph) or impose a structure on the complement of G (after the removal of several edges). Note that every leaf of the decomposition has at most $\mathrm{td}(G)$ neighbors and that the set of leaves spans an edgeless subgraph of G (a clique in $\overline{G}$). We apply the Bondy-Chvátal closure to a vertex v and all leaves beneath v (denote these as L) in the tree-depth decomposition tree. We choose v such that the distance between L and v is 1. We denote ℓ the number of nodes, that is $\ell = |L|$. We denote a *height of vertex v* in the tree-depth decomposition as $h(v)$ and define it as follows. Height of a root is set to 0 and for a vertex v let u denote the closest ancestor of v in the tree-depth decomposition we set $h(v) = h(u) + 1$. Observe that it is possible to add all edges between L and v to $\overline{G}$ if

$$n - h(v) - 1 + n - h(v) - \ell \geq n.$$

That is equivalent to $n > 2h(v) + \ell$.

Lemma 1. *Denote H the graph after application of the closure. We claim that* $\mathrm{nd}(\overline{H}) \leq 2^{2d} + 2d$*, where* $d = \mathrm{td}(G)$*.*

Proof. It follows that if the application process stops, then $n \leq 2h(v) + \ell$ must hold for all nonleaf vertices. Take $h = \max\{h(v) \colon v \text{ nonleaf}\}$. Note that all (actual) leaves of H form a clique in $\overline{H}$ and thus, $\overline{H}$ is a graph on $n \leq 2h + \ell$ vertices with K_ℓ as a subgraph. This in turn yields that the distance to clique (the number of vertices to delete such that the resulting graph is a clique) of $\overline{H}$ is at most $2h$. Thus, the neighborhood diversity of $\overline{H}$ is at most $2^{2h} + 2h$. This follows from the fact that there are at most 2^{2d} different neighborhoods from the point of view of a clique vertex. This together with trivial fact $h \leq \mathrm{td}(G)$ completes the proof.

Lemma 1 yield the following corollary when known algorithms for finding Hamiltonian path are applied to the resulting graph $\overline{H}$.

Theorem 4. *The k-Anti-Path Cover problem admits an* FPT *algorithm with respect to parameterization by the tree-depth of the input graph.* □

4 Tree-width

In this section we restate and prove Theorem 2.

Theorem 5 (Restate Theorem 2). *Let G be a graph of tree-width w and further complement $\overline{G}$ closed under Bondy-Chvátal closure. It follows that neighborhood diversity of $\overline{G}$ is bounded by $2^k + k$ where $k = 2(w^2 + w)$.*

Proof. We will prove the graph $\overline{G}$ has a clique C of size at least $n - 2w^2 - w$. Thus, the graph $\overline{G}$ has at most $2(w^2+w)$ vertices which are not in C. The bound of $\mathrm{nd}(\overline{G})$ follows.

Since $\overline{G}$ is closed under Bondy-Chvátal closure, for every edge $\{u,v\} = e \in E(G)$ holds that $\deg_G(u) + \deg_G(v) \geq n$. Otherwise, it would holds $\deg_{\overline{G}}(u) + \deg_{\overline{G}}(v) \geq n$ and we could add the edge e into $E(\overline{G})$. Thus, for every edge $e \in E(G)$ there exists a vertex $v \in e$ such that $\deg_G(v) \geq \frac{n}{2}$. Let $f : E(G) \to V(G)$ be a function such that for every $v = f(e)$ holds that $v \in e$ and $\deg_G(v) \geq \frac{n}{2}$. Let $V_1 = \{f(e) | e \in E(G)\}$. Note that V_1 is a vertex cover of the graph G. Thus, if we remove the set V_1 from the graph $\overline{G}$ we obtain a clique. Moreover, $V_1 \subseteq V_2 = \{v \in V(G) | \deg_G(v) \geq \frac{n}{2}\}$.

It remains to prove that $|V_2| \leq 2(w^2 + w)$. Let $\mathcal{T} = (T, X)$ be a tree decomposition of G such that width of $\mathcal{T}$ is w and T has n nodes. Let p be a number of all ordered pairs (v, X_i) where $v \in V(G), X_i \in X$ and $v \in X_i$. We use double counting for p. Since T has at most n nodes and all bags in X contains at most $w + 1$ vertices of G, we have

$$p \leq n(w + 1). \tag{1}$$

Let $v \in V_2$ and $E_v = \{e \in E(G) | v \in e\}$. Note that $|E_v| = \deg_G(v) \geq \frac{n}{2}$. Every edge of G has to be in some bag in X. However, there can be only w edges from E_v in one bag in X. Thus, edges from E_v and also the vertex v have to be in at least $\frac{n}{2w}$ bags from X. Therefore, we have lower bound

$$|V_2| \frac{n}{2w} \leq p. \tag{2}$$

When we join Inequality 1 and Inequality 2 we get the right upper bound for V_1 and V_2

$$|V_1| \leq |V_2| \leq 2(w^2 + w).$$

5 Conclusions

We have proven that even through apparently there is no structure in terms of neighborhood diversity on the complements of sparse graphs (having bounded tree-width or tree-depth), the structure after exhaustive application of Bondy-Chvátal closure can be exploited – the complement has bounded neighborhood diversity.

We would like to ask several vague questions here.

- Is it possible to use other graph closure operators to show a connection between tree-width and neighborhood diversity or modular width?
- Is it possible to exhibit closer connection between tree-width and modular width trough graph complements?
- Does any other non-MSO_2 problem besides k-Anti-Path Cover admit an FPT algorithm on a graph with bounded tree-width?
- When one assumes parameterization by the tree-width of an input graph it is convenient to approach the problem by the famous theorem of Courcelle [4]. Is it possible to extend the theorem for MSO_2 for the complementary setting – i.e. to allow quantification over sets of non-edges?

References

1. Umberto Bertelè & Francesco Brioschi (1973): *On non-serial dynamic programming.* *Journal of Combinatorial Theory, Series A* 14(2), p. 137–148, doi:10.1016/0097-3165(73)90016-2.
2. Hans L. Bodlaender, Ton Kloks, Richard B. Tan & Jan van Leeuwen (2004): *Approximations for lambda-Colorings of Graphs.* *Comput. J.* 47(2), pp. 193–204, doi:10.1093/comjnl/47.2.193.
3. J.A. Bondy & V. Chvatal (1976): *A method in graph theory.* *Discrete Mathematics* 15(2), pp. 111 – 135, doi:10.1016/0012-365X(76)90078-9.
4. Bruno Courcelle (1990): *The Monadic Second-Order Logic of Graphs. I. Recognizable Sets of Finite Graphs.* *Inf. Comput.* 85(1), pp. 12–75, doi:10.1016/0890-5401(90)90043-H.
5. Reinhard Diestel (2010): *Graph Theory.* Springer Berlin Heidelberg, doi:10.1007/978-3-642-14279-6.
6. Jakub Gajarský, Michael Lampis & Sebastian Ordyniak (2013): *Parameterized Algorithms for Modular-Width.* In: *Parameterized and Exact Computation - 8th International Symposium, IPEC 2013, Sophia Antipolis, France, September 4-6, 2013, Revised Selected Papers*, pp. 163–176, doi:10.1007/978-3-319-03898-8_15.
7. Ton Kloks (1994): *Treewidth, Computations and Approximations.* *Lecture Notes in Computer Science* 842, Springer, doi:10.1007/BFb0045375.
8. Michael Lampis (2012): *Algorithmic Meta-theorems for Restrictions of Treewidth.* *Algorithmica* 64(1), pp. 19–37, doi:10.1007/s00453-011-9554-x.
9. Jaroslav Nesetril & Patrice Ossona de Mendez (2012): *Sparsity - Graphs, Structures, and Algorithms.* *Algorithms and combinatorics* 28, Springer, doi:10.1007/978-3-642-27875-4.

Hades: Microprocessor Hazard Analysis via Formal Verification of Parameterized Systems

Lukáš Charvát Aleš Smrčka Tomáš Vojnar

Brno University of Technology, FIT, IT4Innovations Centre of Excellence
Božetěchova 2, 612 66 Brno, Czech Republic
{icharvat,smrcka,vojnar}@fit.vutbr.cz

Abstract. HADES[1] is a fully automated verification tool for pipeline-based microprocessors that aims at flaws caused by improperly handled data hazards. It focuses on single-pipeline microprocessors designed at the register transfer level (RTL) and deals with read-after-write, write-after-write, and write-after-read hazards. HADES combines several techniques, including data-flow analysis, error pattern matching, SMT solving, and abstract regular model checking. It has been successfully tested on several microprocessors for embedded applications.

1 Introduction

Implementation of pipeline-based execution of instructions in purpose-specific microprocessors, often used, e.g., in embedded applications, is an error-prone task, which implies a need of proper verification of the resulting designs. Formal verification of such microprocessors—despite they are much simpler than common processors for mainstream computing—is a very challenging task. One way how to deal with it is to develop a set of verification techniques specialised in checking absence of a certain kind of errors in such microprocessors. Here, the main idea is that, this way, a high degree of automation and scalability can be achieved since only parts of a design related to a specific error are to be investigated. The above idea has been followed, e.g., in the works [6,7] that proposed fully automated approaches for (1) checking correctness of individual execution of processor instructions and (2) for verifying absence of read-after-write (RAW) hazards when the instructions are pipelined. In [8], the approach was extended by covering write-after-write (WAW) and write-after-read (WAR) hazards.

To be more precise, an *RAW hazard* arises when an instruction writes to a storage that some later instruction reads, but it is possible for the later instruction to read an old value being rewritten by the earlier instruction. A *WAW hazard* refers to a situation when an instruction writes to a storage and rewrites a result stored by some later instruction which already finished its execution. A *WAR hazard* arises when a later instruction write to a destination before it is read by the previous instruction. There are also non-data hazards. *Structural hazards* deal with sharing resources by instructions in a pipeline. *Control hazards*

[1] www.fit.vutbr.cz/research/groups/verifit/tools/hades/

arise when an instruction is executed improperly due to an unfinished update of a program counter. This paper, however, concentrates on data hazards only.

In particular, the paper presents the HADES tool, developed by VeriFIT research group at FIT BUT, that implements a slightly improved version of the approaches proposed in [7,8]. Namely, after briefly discussing related works, we specify how the input of HADES looks like, we describe its architecture and implementation, and provide experimental results on a larger set of microprocessors than in [7,8]. Moreover, we include a more detailed discussion of the needed verification time and its decomposition to the computing times needed by the different analysis phases implemented in HADES. We close the paper by a discussion of possible future improvements of the HADES tool.

Related Work. Verifying that there are no hazards in a pipelined microprocessor is quite crucial. Hence, it has become a native part of checking conformance between an RTL design and a formally encoded description of an instruction set architecture (ISA), and many approaches with formal roots have been proposed for this purpose. Among them, one can find, e.g., the following approaches [5,13,1,14,15,20,12]. However, these methods typically require a significant manual user intervention—either in a form of specifying the consistent state of the microprocessor or defining predicates describing pipeline behaviour. Compared with such approaches, HADES does not aim at full conformance checking of RTL and ISA implementations. Instead, it addresses one specific property—namely, absence of problems caused by pipeline hazards. On the other hand, HADES is almost fully automated—the user is required to identify the architectural resources (such as registers and memory ports) and the program counter only.

2 Input Models

HADES focuses on microprocessors with a single pipeline and in-order execution. The tool expects storages (registers and memories) to have a unit write and zero read delay. Multicycle delay storages can be easily simulated by a chain of unit storages. The tool also assumes that pipeline internal registers which carry data interchanged between programmer visible storages are controlled by stall and clear signals.

The tool expects the processor under verification to be described by a so-called *processor structure graph* (PSG in short) which represents the internal structure of the processor. A PSG is an oriented graph that consists of vertices (storages or boolean circuits) and edges (control and data connections). An example of a simple PSG is depicted in Figure 1. It shows a part of a simple microprocessor with an accumulator architecture with two architectural registers: X (a memory index register) and A (an accumulator). For the sake of brevity, the PSG does not exhibit control connections of pipeline registers. In the CPU, an instruction fetched from the memory is stored into the storage *id_ir* representing the instruction register. The decoder determines the type of the operation of arithmetic logic unit and identifies its destination by activating the appropriate

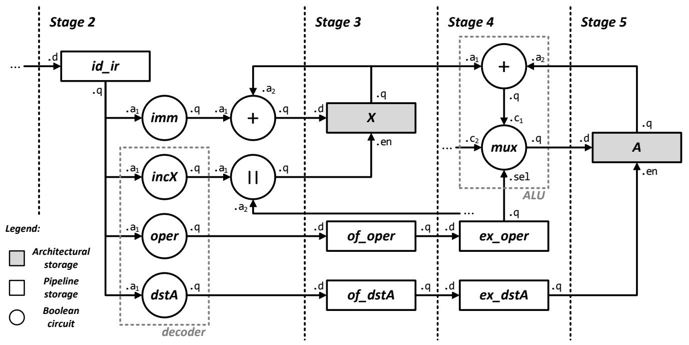


Fig. 1: A processor structure graph of a part of a CPU with an accumulator architecture.

enable connection (en) of the X or A register. An early auto-increment of register X can be performed in stage 3. Such a feature allows the CPU to execute sequences of instructions working with juxtaposed data in the memory without a penalty (brought, e.g., by unnecessary stalls of the pipeline) which would be present if the update of X was done in a later stage.

A design of a processor on the register transfer level (RTL) written in a common hardware design language like VHDL or Verilog can be easily converted into a PSG.

3 The Verification Approach of Hades

The verification approach of Hades was proposed in [7,8]. It leverages the current advances in SMT solvers for bit-vector logic and in formal verification of systems with a parameterized number of processes—for short, referred to as *parameterized systems* (PSs) below. The main idea is to reduce the problem of finding hazards that may arise when executing an in advance unknown number of in advance unknown instructions[2] to a parametric verification problem where the successive instructions are modelled by processes, which gradually pass through the processor. In particular, it turns out that one can use the common notion of PSs operating on a linear topology where the processes (i.e., instructions being executed) may perform local transitions or universally/existentially guarded global transitions [9,18,2].

More precisely, the approach consists of the following steps: (1) a data-flow analysis intended to distinguish particular stages of the pipeline, (2) a consistency check of a correct implementation of the particular pipeline stages,

[2] Note that one cannot simply restrict the checking to a number of instructions given by the number of pipeline stages since the processor can get to different internal states after having processed some number of instructions of some kind.

(3) a static analysis identifying constraints over data-paths of instructions that can potentially cause data hazards, (4) generation of a PS modelling mutual interaction between potentially conflicting instructions, and (5) an analysis of the constructed parameterized system.

Identification of Pipeline Stages. A simple data flow analysis is used to derive the number of pipeline stages implemented in a given processor and to assign storages and logic functions into the pipeline stages. A *pipeline stage* is defined as a sub-graph of the PSG responsible for executing a single-cycle step of an instruction. The pipeline stage of a PSG vertex (representing some storage or function) is given by the minimum number of cycles needed to propagate data from the input of the program counter (assumed to be in a fictive stage 0) to the output of the given vertex.

Consistency Checking. The second step of the method is consistency checking that checks whether the flow logic assures a correct in-order execution of all instructions through all the identified pipeline stages. This step checks whether the flow logic obeys a set of rules that express how the control connections (i.e., enable, stall, and clear signals) of storages in adjacent pipeline stages should be set. In short, the rules require that an instruction carried by a pipeline stage cannot be fragmented, duplicated, or lost. In particular, a strengthened variant of the rules proposed in [16] is used.

Static Detection of Potential Hazards. Next, a static hazard analysis over the PSG with annotated pipeline stages is performed to identify a finite set of so-called *hazard cases*. Each hazard case describes one possible source of a hazard. A hazard case consists of a programmer visible source storage (i.e., a register or a writing port of the memory), target storage, reading and writing stages, and an influence path describing how data propagate between the stages. Since the definition of a hazard case speaks about storages, their access stages, and the path along which the problematic data is transferred, it is not related to a single instruction only but to an entire class of instructions.

Generation of PSs Modelling the Possible Hazards. In this stage, a PS for each identified hazard case is generated. The main component of the PS is a finite automaton whose instances represent instructions passing the pipeline. A state of the automaton identifies the class of instructions that the particular instance represents[3], the execution stage into which the instruction got, and the conditions that must hold for the instruction to proceed such that a flow of data along the path associated with the given hazard case is caused. The transitions of the automata can be guarded by referring to the states of the automata representing instructions that surround the given instruction in the pipeline. Their generation is pruned by checking whether the conditions behind the states of the involved

[3] Three classes are distinguished—write instructions, read instructions, and other instructions.

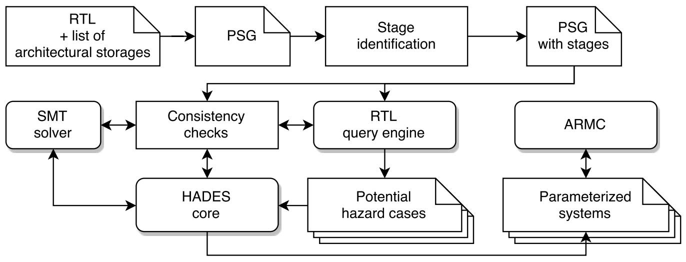


Fig. 2: HADES architecture.

automata do not exclude each other. Further, regular sets of initial and bad configurations are generated. Initial configurations represent simply an arbitrary sequence of instructions waiting for entry into the pipeline. Bad configurations are specified separately for the different types of hazards considered—e.g., for RAW hazards, they say that a later instruction finished reading before an earlier instruction committed writing.

Analysis of the Generated PS. As the last step, it is verified that the bad configurations are not reachable from the initial configurations in the generated PS. For that, *abstract regular model checking* can be used [4].

4 Hades Implementation

The HADES tool implements the above sketched approach and consists of several components depicted in Figure 2. HADES reads in an RTL description of the processor to be verified and converts it into its internal PSG representation. Currently, HADES supports the RTL format of CodAl which is an architectural description language for processor design [10]. For other RTL languages like VHDL and Verilog where architectural storages are not explicitly identified, a list of architectural storages with an explicit identification of the program counter must be provided.

The input PSG is normalized and simplified (conditional branching is replaced by multiplexors, value propagation is applied, redundant nodes and edges are removed, etc.). For that, the *RTL query engine* of HADES, which allows one to search for data-paths and substitute parts of the RTL design described by a PSG, is used. The engine uses a LISP-like syntax both for queries and their output, and it can handle basic RTL constructs like signals, registers, logic gates, as well as memory and its ports.

Subsequently, pipeline stages are identified by a simple data-flow analysis. Intuitively, the analysis propagates so far computed stages forward through the PSG, always taking the minimum of values incoming to a vertex and adding one whenever a storage other than a read port (which has a zero delay) is passed.

Table 1: Experimental results.

Processor / variant		Simpl. Time [s]	Data Flow Analysis [s]	Consistency Checking [s]			Parameterized System Generation and Verification [s]				Total Time [s]	Hazard Cases [#]
				rtl	smt	core	rtl	smt	armc	core		
TinyCPU	S	0.05	0.01	<0.01	0.25	0.49	0.01	0.38	5.44	6.71	13.34	5
	SA	0.06	0.02	<0.01	0.33	0.60	0.02	1.00	11.58	20.84	34.45	8
	B	0.05	0.01	<0.01	0.25	0.44	0.01	0.38	5.08	5.95	12.17	5
	BA	0.07	0.02	<0.01	0.33	0.63	0.03	1.03	11.02	18.28	31.41	8
	SF	0.06	0.02	<0.01	0.30	0.51	0.02	0.77	10.82	13.89	26.39	11
	SFA	0.07	0.02	<0.01	0.34	0.68	0.04	1.88	20.42	43.09	66.54	18
SPP8	S	0.27	0.04	0.01	0.43	0.85	0.05	2.02	20.81	36.24	60.72	27
	B	0.25	0.03	0.01	0.40	0.82	0.07	2.16	20.35	43.19	67.28	27
SPP16	S	0.27	0.05	0.01	0.44	0.90	0.04	1.90	19.99	36.33	59.93	27
	B	0.30	0.05	0.01	0.43	0.88	0.07	2.16	19.75	42.29	65.94	27
Codea2	SF	0.81	0.13	0.01	0.59	1.04	0.94	32.91	224.73	527.34	788.49	239
CompAcc	SFA	0.27	0.04	0.01	0.54	1.06	0.11	5.60	65.83	98.05	171.87	38
	BFA	0.28	0.05	0.01	0.55	1.04	0.28	7.74	66.03	158.56	234.94	53
DLX5	S	0.47	0.08	0.01	1.09	2.23	0.22	8.96	140.40	205.69	359.15	25
	SA	0.54	0.10	0.01	1.12	2.44	0.37	17.54	250.78	460.75	733.65	59
	B	0.62	0.12	0.01	1.07	2.40	0.33	9.47	138.55	316.08	468.65	25
	BA	0.65	0.12	0.01	1.15	2.69	0.48	19.28	247.98	745.16	1017.52	59

S Stalling Logic B Bypassing Logic F Flag Register(s) A Auto-increment Logic

Next, instances of the consistency rules for the particular design are derived using RTL query engine. The rules are checked using an SMT solver for bit-vector logic. Hades is compatible with all SMT solvers accepting SMT2 formula description. In particular, for the below experiments, Z3 [17] was used.

Further, given a PSG with annotated stages, the Hades core repeatedly utilizes the RTL query engine (written in C++) and the SMT solver to extract potential hazard cases and to generate the appropriate PSs for them. The generated PSs are then checked using the abstract regular model checker of [3] (implemented in OCaml over the Timbuk tree automata library [11], however, tree automata degenerated to word automata are used only).

Note that the different hazard cases are are independent, and hence, in the future, the generation of the PSs and their verification can be run in parallel.

5 Experimental Evaluation

We have tested Hades on five processors: *TinyCPU* is a small 8-bit processor, mainly used for testing new verification methods. *SPP8* is an 8-bit ipcore with 3 pipeline stages, 16 general-purpose registers, and a RISC instruction set with 9 instructions. *SPP16* is a 16-bit variant of SPP8 with a more complex memory model. *Codea2* is a 16-bit processor for signal processing applications. It is equipped with 16 general-purpose registers, 15 special registers, a flag register, and an instruction set including 41 instructions where each may use up to 4 available addressing modes. *CompAcc* is an 8-bit processor based on an accumulator architecture. Finally, *DLX5* is a 5-staged 32-bit processor able to execute a subset of the instruction set of the DLX architecture [19] (with no floating point support).

Compared with [6,7,8], we enriched the number of variants for the above introduced processors, which gave us 17 unique test cases in total. The variants of the particular processors differ in the following aspects: (i) the way how data hazards are avoided (pipeline stalling and clearing, data bypassing), (ii) the presence of flag / status registers, and (iii) utilization of so-called *auto-increment* (AI) logic. The AI logic is a feature allowing for an early incrementation[4] of the value of a register for memory addressing just before (pre-increment) or right after (post-increment) it is read. The AI feature usually brings a more efficient execution of sequences of instructions accessing the processor's memory (e.g., computation upon long arrays in cyber-security CPUs), but it also introduces potential WAW and WAR hazards that must be handled properly.

Besides the modifications in our test cases, we improved the Hades tool as well. This includes an addition of dynamic programming techniques (e.g., paths found in PSG are hashed and reused) and a faster pipe-based communication (instead of previously used file-based) between the Hades core and the RTL query engine.

We conducted a series of experiments on a PC with Intel Core i7-3770K @ 3.50GHz and 16 GB RAM with results shown in Table 1. The first columns give the verified processor, its variant, the time needed for the PSG simplification and its data flow analysis. The next columns give the duration of the consistency checking and the time spent by verification of the PSs that are created for each hazard case. The times are split to the times consumed by the different parts of the Hades architecture.

The following column gives the overall verification time, which remains in the order of minutes even for complex designs. Moreover, Hades also scales well with the growing size of the processor data-path as can be seen by comparing the times obtained for *SPP8* and *SPP16*. It should be noted that the amount of time consumed by the tool's core can be reduced by using a direct API of the SMT solver used instead of the current implementation that relies on exporting (potentially large) formulas in the `smt2` file format. (On the other hand, the current implementation does not depend on any particular SMT solver.) Finally, the last column represents the number of hazard cases that had to be generated and checked. This number differs from the one computed in [7,8] due to Hades newly does not include hazard cases on the program counter among data hazards. These cases will be treated in separate control hazard detection phase, which is currently under implementation. Note that each hazard case represents a separate task so the part of generation and verification of PSs can be parallelized in the future.

During the experiments, we identified a flaw in a RAW hazard resolution when accessing the data memory in a development version of the *SPP8* processor.

6 Conclusions and Future Work

We have presented the main ideas, architecture, and evaluation of Hades—a tool for fully-automated discovery of data hazards in pipelined microprocessors.

[4] The incrementation typically takes place in an execution stage of the processor's pipeline.

In the future, we plan to extend HADES with methods for verification of other processor features, such as control hazards. We also plan to parallize some parts of HADES and extend it with a compiler from VHDL and Verilog IP cores to the HADES input format.

Acknowledgement. The work was supported by the Czech Science Foundation project 14-11384S, the IT4IXS: IT4Innovations Excellence in Science project (LQ1602), and the internal BUT project FIT-S-14-2486.

References

1. M. D. Aagaard (2003): *A Hazards-Based Correctness Statement for Pipelined Circuits.* In: *Proc. of CHARME'03*, *LNCS* 2860, Springer, pp. 66–80, doi:10.1007/978-3-540-39724-3_8.
2. P. A. Abdulla, F. Haziza. & L. Holik (2013): *All for the Price of Few (Parameterized Verification through View Abstraction).* In: *Proc. of VMCAI'13*, *LNCS* 7737, Springer, pp. 476–495, doi:10.1007/s10009-015-0406-x.
3. A. Bouajjani, P. Habermehl, L. Holiḱ, T. Touili & T. Vojnar (2008): *Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata.* In: *Proc. of CIAA'08*, LNCS 5148, Springer, doi:10.1007/978-3-540-70844-5_7.
4. A. Bouajjani, P. Habermehl & T. Vojnar (2004): *Abstract Regular Model Checking.* In: *Proc. of CAV'04*, *LNCS* 3114, Springer, pp. 197–202, doi:10.1007/978-3-540-30579-8_19.
5. J. R. Burch & D. L. Dill (1994): *Automatic Verification of Pipelined Microprocessor Control.* In: *Proc. of CAV'94*, *LNCS* 818, Springer, pp. 68–80, doi:10.1007/3-540-58179-0_44.
6. L. Charvat, A. Smrcka & T. Vojnar (2012): *Automatic Formal Correspondence Checking of ISA and RTL Microprocessor Description.* In: *Proc. of MTV'12*, IEEE, pp. 6–12, doi:10.1109/mtv.2012.19.
7. L. Charvat, A. Smrcka & T. Vojnar (2014): *Using Formal Verification of Parameterized Systems in RAW Hazard Analysis in Microprocessors.* In: *Proc. of MTV'14*, IEEE, pp. 83–89, doi:10.1109/mtv.2014.21.
8. L. Charvát, A. Smrčka & T. Vojnar (2015): *Microprocessor Hazard Analysis via Formal Verification of Parameterized Systems.* In: *Proc. of EUROCAST'15*, *LNCS* 9520, Springer, pp. 605–614, doi:10.1007/978-3-319-27340-2_75.
9. E. Clarke, M. Talupur & H. Veith (2006): *Environment abstraction for parameterized verification.* In: *Proc. of VMCAI'06*, *LNCS* 3855, Springer, pp. 126–141, doi:10.1007/11609773_9.
10. Codasip Ltd. (2013): *CodAL Architecture Description Language.*
11. T. Genet: *Timbuk: A Tree Automata Library.* http://www.irisa.fr/lande/genet/timbuk.
12. K. Hao, S. Ray & F. Xie (2014): *Equivalence Checking for Function Pipelining in Behavioral Synthesis.* In: *Proc. of DATE'14*, IEEE, pp. 1–6, doi:10.7873/date.2014.163.
13. R. B. Jones, C. H. Seger & D. L. Dill (1996): *Self-Consistency Checking.* In: *Proc. of FMCAD'96*, *LNCS* 1166, Springer, pp. 159–171, doi:10.1007/bfb0031806.

14. A. Koelbl, R. Jacoby, H. Jain & C. Pixley (2009): *Solver Technology for System-level to RTL Equivalence Checking.* In: *Proc. of DATE'09*, IEEE, pp. 196–201, doi:10.1109/date.2009.5090657.
15. U. Kuhne, S. Beyer, J. Bormann & J. Barstow (2010): *Automated Formal Verification of Processors Based on Architectural Models.* In: *Proc. of FMCAD'10*, IEEE, pp. 129–136.
16. P. Mishra, H. Tomiyama, N. Dutt & A. Nicolau (2002): *Automatic Verification of In-Order Execution in Microprocessors with Fragmented Pipelines and Multicycle Functional Units.* In: *Proc. of DATE'02*, IEEE, pp. 36–43, doi:10.1109/date.2002.998247.
17. L. De Moura & N. Bjorner (2008): *Z3: An Efficient SMT Solver.* In: *Proc. of TACAS'08*, *LNCS* 4963, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.
18. K. S. Namjoshi (2007): *Symmetry and completeness in the analysis of parameterized systems.* In: *Proc. of VMCAI'07*, *LNCS* 4349, Springer, pp. 299–313, doi:10.1007/978-3-540-69738-1_22.
19. D. A. Patterson & J. L. Hennessy (2012): *Computer Organization and Design: The Hardware / Software Interface*, fourth edition. Morgan Kaufmann, Boston, doi:10.1016/c2013-0-08305-3.
20. M. N. Velev & P. Gao (2011): *Automatic Formal Verification of Multi-threaded Pipelined Microprocessors.* In: *Proc. of ICCAD'11*, IEEE, pp. 679–686, doi:10.1109/iccad.2011.6105403.

A note on one less known class of generated residual implications

Vojtěch Havlena[1] and Dana Hliněná[2]

[1] Brno University of Technology,
Faculty of Information Technology,
Brno, Czech Republic
`xhavle03@stud.fit.vutbr.cz`,
[2] Brno University of Technology,
Faculty of Electrical Engineering and Communication,
Brno, Czech Republic
`hlinena@feec.vutbr.cz`

Abstract. This paper builds on our contribution [4] which studied modelling of the conjunction in human language. We have discussed three different ways of constructing a conjunction. We have dealt with generated t-norms, generated means and Choquet integral.
In this paper we construct the residual operators based on the above conjunctions. The only operator based on a t-norm is an implication. We show that this implication belongs to the class of generated implications I_N^g which was introduced in [8] and studied in [3]. We study its properties. Moreover, we investigate this class of generated implications. Some important properties, including relations between some classes of implications, are given.

1 Introduction

In [4], we studied modelling of the conjunction in human language. We have experimentally rated simple statements and their conjunctions. Then we have tried, on the basis of measured data, to find a suitable function, which corresponds to human conjunction. We have discussed three different ways of constructing a conjunction. We have dealt with generated t-norms, generated means and Choquet integral. Now we are interested in a construction of the implications based on the above conjuctions. One of the possible ways to construct the implications is the following transformation

$$\forall x, y, u \in [0,1]; C(x,u) \leq y \iff R_C(x,y) \geq u.$$

This transformation produces the residual operator R_C based on the given conjunction C. For some conjunctions we can get, in this way, a residual operator which is an implication.

For better understanding we recall basic definitions and statements used in the paper. We deal with multivalued (MV for short) logical connectives, which are monotone extensions of the classical connectives on the unit interval $[0,1]$. We turn our attention to the conjunctions in MV-logic. Usually, the triangular norms are used to interpret the conjunctions in MV-logic.

Definition 1. *[7] A triangular norm (t-norm for short) is a binary operation on the unit interval* $[0,1]$*, i.e., a function* $T : [0,1]^2 \to [0,1]$*, such that for all* $x, y, z \in [0,1]$ *the following four axioms are satisfied:*

– *(T1) Commutativity*
$$T(x,y) = T(y,x),$$
– *(T2) Associativity*
$$T\left(T(x,y),z\right) = T\left(x,T(y,z)\right),$$
– *(T3) Monotonicity*
$$T(x,y) \le T(x,z) \text{ whenever } y \le z,$$
– *(T4) Boundary Condition*
$$T(x,1) = x.$$

The four basic t-norms are:

– the minimum t-norm $T_M(x,y) = \min\{x,y\}$,
– the product t-norm $T_P(x,y) = x \cdot y$,
– the Łukasiewicz t-norm $T_L(x,y) = \max\{0, x+y-1\}$,
– the drastic product $T_D(x,y) = \begin{cases} 0 & \text{if } \max\{x,y\} < 1, \\ \min\{x,y\} & \text{otherwise.} \end{cases}$

We deal only with such continuous t-norms, that are generated by a unary function (the generator). One possibility is to generate by an additive generator, which is a strictly decreasing function f from the unit interval $[0,1]$ to $[0,+\infty]$ such that $f(1) = 0$ and $f(x) + f(y) \in H(f) \cup [f(0^+), +\infty]$ for all $x, y \in [0,1]$, where $H(f)$ is range of f. Then the generated t-norm is given as follows
$$T(x,y) = f^{(-1)}\left(f(x) + f(y)\right),$$
where $f^{(-1)} : [0,+\infty] \to [0,1]$ and $f^{(-1)}(y) = \sup\{x \in [0,1] \,|\, f(x) > y\}$. Note, that $f^{(-1)}$ is a pseudo-inverse, which is a monotone extension of the ordinary inverse function. For an illustration, we give the following example of parametric class of t-norms and their additive generators.

The family of Yager t-norms, introduced by Ronald R. Yager, is given for $0 \le p \le +\infty$ by
$$T_p^Y(x,y) = \begin{cases} T_D(x,y) & \text{if } p = 0, \\ T_M(x,y) & \text{if } p = +\infty, \\ \max\left\{0, 1 - \left((1-x)^p + (1-y)^p\right)^{\frac{1}{p}}\right\} & \text{if } 0 < p < +\infty. \end{cases}$$

The additive generator of T_p^Y for $0 < p < +\infty$ is

$$f_p^Y(x) = (1-x)^p.$$

Because of associativity, we can extend t-norms to the n-variete case as:

$$x_T^{(n)} = \begin{cases} x & \text{if } n = 1, \\ T(x, x_T^{(n-1)}) & \text{if } n > 1. \end{cases}$$

A t-norm T is called Archimedean if for each x, y in the open interval $]0,1[$ there is a natural number n such that $x_T^{(n)} \leq y$. It is sufficient to investigate Archimedean t-norms, because every non-Archimedean t-norm can be approximated arbitrarily well with Archimedean t-norms, [6, 5].

Remark 1. If T is a t-norm, then the dual function $S : [0,1]^2 \to [0,1]$ defined by $S(x,y) = 1 - T(1-x, 1-y)$ is called a t-conorm. Its neutral element is 0 instead of 1, and all other conditions remain unchanged. Analogously to the case of t-norms, some classes of t-conorms can be generated by additive generators. The additive generator for a t-conorm is a strictly increasing function g from the unit interval $[0,1]$ to $[0,+\infty]$ such that $g(0) = 0$ and $g(x)+g(y) \in H(g) \cup [g(1^-), +\infty]$ for all $x, y \in [0,1]$. The generated t-conorm is given by

$$S(x,y) = g^{(-1)}\left(g(x) + g(y)\right),$$

where $g^{(-1)}(y) = \sup\{x \in [0,1] \,|\, g(x) < y\}$. Note that t-conorms are usually used for modelling fuzzy disjunctions.

Now, we continue with definitions and properties of fuzzy negations.

Definition 2. *(see e.g. in [2]) A function $N : [0,1] \to [0,1]$ is called a* fuzzy negation *if, for each $a, b \in [0,1]$, it satisfies the following conditions*

- *(i)* $a < b \Rightarrow N(b) \leq N(a)$,
- *(ii)* $N(0) = 1, N(1) = 0$.

Remark 2. A *dual negation* $N^d : [0,1] \to [0,1]$ based on a negation N, is given by $N^d(x) = 1 - N(1-x)$. A fuzzy negation N is called *strict* if N is strictly decreasing and continuous for arbitrary $x, y \in [0,1]$. In classical logic we have that $(\boldsymbol{A}')' = \boldsymbol{A}$. In multivalued logic this equality is not satisfied for every negation. The negations with this equality are called *involutive negations.* The strict negation is *strong* if and only if it is involutive. The most important and most widely used strong negation is the standard negation $N_S(x) = 1 - x$.

In the literature, one can find several different definitions of fuzzy implications. In this paper we will use the following one, which is equivalent to the definition introduced by Fodor and Roubens in [2].

Definition 3. *A function $I : [0,1]^2 \to [0,1]$ is called a* fuzzy implication *if it satisfies the following conditions:*

(I1) I is non-increasing in its first variable,
(I2) I is non-decreasing in its second variable,
(I3) $I(1,0) = 0$, $I(0,0) = I(1,1) = 1$.

We recall definitions of some important properties of fuzzy implications which we will investigate.

Definition 4. *A fuzzy implication $I : [0,1]^2 \to [0,1]$ satisfies:*

(NP) the left neutrality property if

$$I(1,y) = y \quad \textit{for all } y \in [0,1],$$

(EP) the exchange principle if

$$I(x, I(y,z)) = I(y, I(x,z)) \quad \textit{for all } x,y,z \in [0,1],$$

(IP) the identity principle if

$$I(x,x) = 1 \quad \textit{for all } x \in [0,1],$$

(OP) the ordering property if

$$x \leq y \iff I(x,y) = 1 \quad \textit{for all } x,y \in [0,1],$$

(CP) the contrapositive symmetry with respect to a given fuzzy negation N if

$$I(x,y) = I(N(y), N(x)) \quad \textit{for all } x,y \in [0,1].$$

Definition 5. *Let $I : [0,1]^2 \to [0,1]$ be a fuzzy implication. The function N_I defined by $N_I(x) = I(x,0)$ for all $x \in [0,1]$, is called the natural negation of I.*

(S,N)-implications which are based on t-conorms and fuzzy negations form one of the well-known classes of fuzzy implications.

Definition 6. *A function $I : [0,1]^2 \to [0,1]$ is called an (S,N)-implication if there exist a t-conorm S and a fuzzy negation N such that*

$$I(x,y) = S(N(x), y), \quad x,y \in [0,1].$$

If N is a strong negation then I is called a strong implication.

The following characterization of (S,N)-implications is from [1].

Theorem 1. *(Baczyński and Jayaram [1], Theorem 5.1) For a function I : $[0,1]^2 \to [0,1]$, the following statements are equivalent:*

- *I is an (S,N)-implication generated from some t-conorm and some continuous (strict, strong) fuzzy negation N.*
- *I satisfies (I2), (EP), and N_I is a continuous (strict, strong) fuzzy negation.*

Another way of extending the classical binary implication to the unit interval $[0,1]$ is based on the residual operator with respect to a left-continuous triangular norm T

$$I_T(x,y) = \max\{z \in [0,1] \mid T(x,z) \le y\}.$$

Elements of this class are known as R-implications. The following characterization of R-implications is from [2].

Theorem 2. *(Fodor and Roubens [2], Theorem 1.14) For a function $I : [0,1]^2 \to [0,1]$, the following statements are equivalent:*

- *I is an R-implication based on some left-continuous t-norm T.*
- *I satisfies (I2), (OP), (EP), and $I(x,.)$ is right-continuous for any $x \in [0,1]$.*

At last we introduce a characterization of implications based on Φ-conjugate from [1].

Definition 7. *We denote by Φ the family of all increasing bijections on the unit interval $[0,1]$. We say that implications $I_1, I_2 : [0,1]^2 \to [0,1]$ are Φ-conjugate if there exists a bijection $\varphi \in \Phi$ such that $I_2 = (I_1)_\varphi$, where*

$$(I_1)_\varphi(x,y) = \varphi^{-1}(I_1(\varphi(x), \varphi(y))),$$

for all $x, y \in [0,1]$.

Theorem 3. *(Baczyński and Jayaram [1], Theorem 2.4.20) Let $I : [0,1]^2 \to [0,1]$ be a function. Then I is a continuous function satisfying (OP), (EP), if and only if, I is Φ-conjugate with the Łukasiewicz implication.*

It is well-known that it is possible to generate t-norms from one variable functions. Therefore the question whether something similar is possible in the case of fuzzy implications is very interesting. In [9] Yager introduced two new classes of fuzzy implications: f-implications and g-implications where their generators f are continuous additive generators of continuous Archimedean t-norms and generators g are continuous additive generators of continuous Archimedean t-conorms.

In this paper we deal with some of less known classes of generated fuzzy implications which were introduced in [8] and studied in [3].

The first class of generated implications is based on strictly increasing functions g.

Theorem 4. *[8] Let $g : [0,1] \to [0,\infty]$ be a strictly increasing function such that $g(0) = 0$. Then the function $I^g : [0,1]^2 \to [0,1]$ which is given by*

$$I^g(x,y) = g^{(-1)}(g(1-x) + g(y)), \tag{1}$$

is a fuzzy implication.

The fuzzy implication I^g can be generalized. This generalization is based on replacing the standard negation by an arbitrary one.

Theorem 5. *[8] Let $g : [0,1] \to [0,\infty]$ be a strictly increasing function such that $g(0) = 0$ and N be a fuzzy negation. Then the function I_N^g*

$$I_N^g(x,y) = g^{(-1)}(g(N(x)) + g(y)), \tag{2}$$

is a fuzzy implication.

2 The residual operators based on the considered conjunctions

As mentioned in the first section, we have found residual operators of conjunctions which were based on empirically measured data.

The first conjunction was the t-norm T_2^Y which is given by

$$T_2^Y(x,y) = \max\left\{0, 1 - \left((1-x)^2 + (1-y)^2\right)^{\frac{1}{2}}\right\}.$$

It is Yager's t-norm with parameter $p = 2$. The corresponding residual operator (Fig. 1a) is given by

$$I_{T_2^Y}(x,y) = 1 - (\max((1-y)^2 - (1-x)^2), 0)^{\frac{1}{2}}. \tag{3}$$

In general, residual implications which are based on Yager t-norms T_p^Y are given by:

$$I_{T_p^Y}(x,y) = 1 - (\max((1-y)^p - (1-x)^p), 0)^{\frac{1}{p}}. \tag{4}$$

Now, we will investigate properties of implications $I_{T_p^Y}$ and their membership in the classes of implications. We turn our attention to the class of I^g implications. The boundary conditions for I^g implications are given by

$$I^g(x,0) = g^{(-1)} \circ g(1-x) = 1-x,$$

$$I^g(1,y) = g^{(-1)} \circ g(y) = y.$$

On the other hand, residual implication $I_{T_p^Y}$ satisfies the following equality

$$I_{T_p^Y}(x,0) = 1 - (\max(1 - (1-x)^p), 0)^{\frac{1}{p}} = 1 - (1 - (1-x)^p)^{\frac{1}{p}}.$$

Therefore the implication $I_{T_p^Y}$ can not be expressed as I^g, but as I_N^g. The function

$$N_p(x) = I_{T_p^Y}(x,0) = 1 - (1 - (1-x)^p)^{\frac{1}{p}}$$

is a negation (particularly, for $p = 2$ we get $N_2(x) = 1 - \sqrt{x(2-x)}$) and since $I_N^g(x,0) = g^{(-1)}(g(N(x)), g(0)) = N(x)$, the implication $I_{T_p^Y}$ is expressed by the function I_N^g with negation $N = N_p$.

Furthermore, we consider the function $g_p(x) = 1 - (1-x)^p$, where $g_p^{-1}(x) = 1 - (1-x)^{\frac{1}{p}}$. Then the function $I_{N_p}^{g_p}$ is given by

$$\begin{aligned} I_{N_p}^{g_p}(x,y) &= g_p^{-1}(\min(g_p(N_p(x)) + g_p(y), g_p(1))) \\ &= g_p^{-1}(\min((1-x)^p + 1 - (1-y)^p, 1)) \\ &= 1 - (1 - \min((1-x)^p + 1 - (1-y)^p, 1))^{\frac{1}{p}}. \end{aligned}$$

Since $1 - \min(1-x, 1-y) = \max(x,y)$ we have

$$I_{N_p}^{g_p}(x,y) = 1 - (\max((1-y)^p - (1-x)^p), 0)^{\frac{1}{p}} = I_{T_p^Y}(x,y).$$

Let $p > 0$. Directly from Definition 4 we get that the implications $I_{T_p^Y}$ satisfy properties (IP) and (NP). Since the implications $I_{T_p^Y}$ are residual operators based on the left-continuous t-norms T_p^Y, and due to Theorem 2, properties (EP) and (OP) are satisfied for these implications. Additionally

$$I_{T_p^Y}(N_p(y), N_p(x)) = 1 - (\max(1 - (1-x)^p - (1-(1-y))^p), 0)^{\frac{1}{p}} = I_{T_p^Y}(x,y),$$

which is the property (CP) with respect to the negations N_p.

The next conjunction is a quasi-arithmetic mean M (for more details see [4]). Its residual operator is given by formula

$$\begin{aligned} M_r(x,y) &= \sup\{t \in [0,1] \mid M(x,t) \le y\} = \sup\left\{t \in [0,1] \;\middle|\; \frac{1}{2}(x^2 + t^2) \le y^2\right\} \\ &= (\min\{\max\{2y^2 - x^2, 0\}, 1\})^{\frac{1}{2}}. \end{aligned}$$

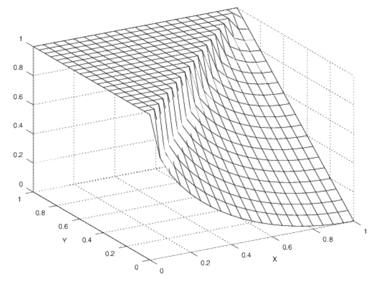
This operator is not an implication, since the boundary condition $I(0,0) = 1$ is violated (Fig. 1c). The same problem occurs with residual operator of the last conjunction, which is Choquet integral (Fig. 1b). Therefore we will not discuss these operators.

3 Properties of I^g and I_N^g implications

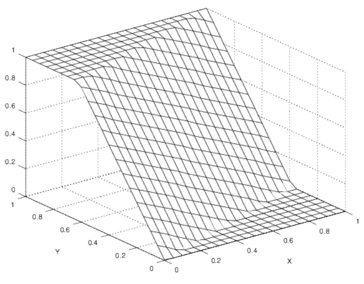
In this section we investigate properties of generated implications I^g and I_N^g. We focus on relations between these generated implications and some well known classes of implications.

In the following text we denote by $\mathbb{I}^g$ the class of I^g implications and by $\mathbb{I}_N^g$ the class of I_N^g implications. Further we denote by $\mathbb{I}_{\mathbb{T}_{\mathbb{LC}}}$ the class of R-implications based on left-continuous t-norm and by $\mathbb{I}_{\mathbb{S},\mathbb{N}}$ the class of (S,N)-implications. With the subscript c we denote a continuous function (we use it in the context of continuous functions g and N).

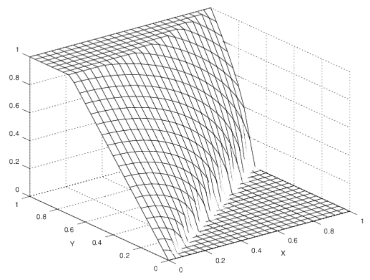
Two of the best known classes of implications are R-implications and (S,N)-implications. In the first part we focus on the relation of the classes $\mathbb{I}_N^g$ and $\mathbb{I}_{\mathbb{S},\mathbb{N}}$. We are interested in two questions – whether the class $\mathbb{I}_N^g$ is a proper subclass of $\mathbb{I}_{\mathbb{S},\mathbb{N}}$ and if not, find a subclass C of $\mathbb{I}_N^g$ satisfying $C \subseteq \mathbb{I}_{\mathbb{S},\mathbb{N}}$.



(a) Residual operator of t-norm T_2^Y.



(b) Residual operator of Choquet integral.



(c) Residual operator of quasi arithmetic mean M_r.

Fig. 1: Residual operators based on the found conjunctions.

Lemma 1. *Let $I : [0,1]^2 \to [0,1]$ be an implication. If $I \in \mathbb{I}_N^{g_c}$ then $I \in \mathbb{I}_{\mathbb{S},\mathbb{N}}$.*

Proof. We deal with I_N^g, where N is an arbitrary negation and g is a continuous generator. Since g is a strictly increasing continuous function with $g(0) = 0$, it holds

$$g^{(-1)}(g(x) + g(y)) = S_g(x, y),$$

where S_g is t-conorm generated by g. Accordingly

$$I(x, y) = I_N^g(x, y) = g^{(-1)}(g(N(x)) + g(y)) = S_g(N(x), y)$$

and thus $I \in \mathbb{I}_{\mathbb{S},\mathbb{N}}$.

For illustration we provide the following example:

Example 1. Let $g : [0,1] \to [0,\infty]$ be a function given by the following formula

$$g(x) = -\ln(1 - x).$$

The function g is strictly increasing and continuous. Its pseudoinverse function $g^{(-1)}$ is given by

$$g^{(-1)}(x) = 1 - e^{-x} \quad \text{for } x \in [0, \infty].$$

Then for the function g we get the following implication

$$I^g(x, y) = 1 - e^{\ln(x(1-y))} = 1 - x + xy,$$

which is $S_P(1 - x, y)$, where S_P is dual operator to the product t-norm and I^g is thus an (S, N)-implication with negation $N(x) = 1 - x$.

Lemma 2. *For the classes $\mathbb{I}_N^g$ and $\mathbb{I}_{\mathbb{S},\mathbb{N}}$, it holds $\mathbb{I}_N^g \setminus \mathbb{I}_{\mathbb{S},\mathbb{N}} \neq \emptyset$.*

Proof. We assume $\mathbb{I}_N^g \setminus \mathbb{I}_{\mathbb{S},\mathbb{N}} = \emptyset$.

We turn our attention to the following example: We consider the strictly increasing function $f : [0, 1] \to [0, \infty]$ which is given by formula

$$f(x) = \begin{cases} x & \text{if } x \leq 0.5, \\ 0.5 + 0.5x & \text{otherwise.} \end{cases}$$

Its pseudoinverse function is given by

$$f^{(-1)}(x) = \begin{cases} x & \text{if } x \leq 0.5, \\ 0.5 & \text{if } 0.5 < x \leq 0.75, \\ 2x - 1 & \text{if } 0.75 < x \leq 1, \\ 1 & \text{if } 1 < x. \end{cases}$$

Finally, for implication based on the function f we get

$$I^f(x, y) = \begin{cases} 1 - x + y & \text{if } x \geq 0.5, y \leq 0.5, x - y \geq 0.5, \\ 0.5 & \text{if } x \geq 0.5, y \leq 0.5, 0.25 \leq x - y < 0.5, \\ 1 - 2x + 2y & \text{if } x \geq 0.5, y \leq 0.5, x - y < 0.25, \\ \min(1 - x + 2y, 1) & \text{if } x < 0.5, y \leq 0.5, \\ \min(2 - 2x + y, 1) & \text{if } x \geq 0.5, y > 0.5, \\ 1 & \text{if } x < 0.5, y > 0.5. \end{cases}$$

Now we will construct a negation N and a t-conorm S such that $I^f(x, y) = S(N(x), y)$. From the boundary condition we get

$$I^f(x, 0) = f^{(-1)} \circ f(1 - x) = 1 - x = S(N(x), 0) = N(x)$$

and therefore $S(x, y) = I^f(1 - x, y)$ is a t-conorm. But

$$\begin{aligned} S(0.3, S(0.35, 0.2)) &= S(0.3, 0.5) = 1 - 1.4 + 1 = 0.6 \\ S(S(0.3, 0.35), 0.2) &= S(0.5, 0.2) = 0.5 \end{aligned}$$

and thus S is not associative, which is a contradiction.

Theorem 6. *For the classes* $\mathbb{I}^{g_c}, \mathbb{I}^{g_c}_{N_c}$ *and* $\mathbb{I}_{\mathbb{S},\mathbb{N}}$, *it holds* $\mathbb{I}^{g_c} \subset \mathbb{I}^{g_c}_{N_c} \subset \mathbb{I}_{\mathbb{S},\mathbb{N}}$.

Proof. Apparently $\mathbb{I}^{g_c} \subseteq \mathbb{I}^{g_c}_{N_c}$ holds true and the implication $I_{T_2^Y}$ from the previous section forms an example of an implication in $\mathbb{I}^{g_c}_{N_c} \setminus \mathbb{I}^{g_c}$. From Lemma 1 we get $\mathbb{I}^{g_c}_{N_c} \subseteq \mathbb{I}_{\mathbb{S},\mathbb{N}}$. If we consider the (S,N)-implication $I(x,y) = \max\{1-x, y\}$ and try to express this implication as I^g_N, we obtain $I(x,y) = \max\{1-x,y\} = g^{(-1)}(g(1-x) + g(y))$, which is an expresion via additive generator, but the t-conorm $\max\{x,y\}$ has no additive generator. Therefore $\mathbb{I}_{\mathbb{S},\mathbb{N}} \setminus \mathbb{I}^{g_c}_{N_c} \neq \emptyset$.

The second part is devoted to the relation of a subclass of $\mathbb{I}^g_N$, with continuous generator g and continuous negation N, and $\mathbb{I}_{\mathbb{T}_{\mathbb{LC}}}$, which is explained in the following assertion.

Lemma 3. *Let* $I : [0,1]^2 \to [0,1]$ *be an implication such that* $I \in \mathbb{I}^{g_c}_{N_c}$. *Then* I *is an* R*-implication based on left-continuous t-norm if and only if* I *is* Φ*-conjugate with the Łukasiewicz implication.*

Proof. ($\Rightarrow$) We assume that $I = I^g_N$ for some continuous g and N. According to Lemma 1 we get $I(x,y) = S_g(N(x),y)$. Since both g and N are continuous functions, also S_g is continuous and therefore I is continuous, too. By the assumption, I is an R-implication based on left-continuous t-norm. From Theorem 2 we directly get that, I satisfying properties (OP) and (EP) and from Theorem 3 we finally obtain that I is Φ-conjugate with the Łukasiewicz implication.

($\Leftarrow$) Since I is Φ-conjugate with the Łukasiewicz implication, according to Theorem 3, I is a continuous implication satisfying (OP), (EP) and from Theorem 2 we get that I is an R-implication based on a left-continuous t-norm.

Theorem 7. *Let* $I : [0,1]^2 \to [0,1]$ *be an implication such that* $I \in \mathbb{I}^{g_c}_{N_c}$. *Then* I *is an* R*-implication based on a left-continuous t-norm if and only if* $I = I^{\varphi}_{N_\varphi}$, *where* $N_\varphi(x) = \varphi^{-1}(1-\varphi(x))$ *for some* $\varphi \in \Phi$.

Proof. ($\Rightarrow$) Since I is an R-implication based on a left-continuous t-norm, from Lemma 3 we get that I is Φ-conjugate with the Łukasiewicz implication, and thus for all $x,y \in [0,1]$,

$$I(x,y) = (I_{\mathbf{LK}}(x,y))_\varphi = \varphi^{-1}(\min\{1-\varphi(x)+\varphi(y), 1\}) = I^{\varphi}_{N_\varphi}(x,y),$$

where $I_{\mathbf{LK}}$ is the Łukasiewicz implication given by $I_{\mathbf{LK}}(x,y) = \min\{1-x+y, 1\}$. The last equality holds because, for all $x,y \in [0,1]$

$$I^{\varphi}_{N_\varphi}(x,y) = \varphi^{-1}(\min\{\varphi(N_\varphi(x)) + \varphi(y), \varphi(1)\}) = \varphi^{-1}(\min\{1-\varphi(x)+\varphi(y), 1\}).$$

($\Leftarrow$) This directly follows from Lemma 3 and equality $(I_{\mathbf{LK}})_\varphi = I^{\varphi}_{N_\varphi}$.

Directly from previous theorem we get what are the intersection of $\mathbb{I}_{\mathbb{T}_{\mathbb{LC}}}$ and $\mathbb{I}^{g_c}_{N_c}$, $\mathbb{I}^{g_c}$ respectively. (Fig. 2).

Corollary 1. $\mathbb{I}_{\mathbb{T}_{\mathbb{LC}}} \cap \mathbb{I}^{g_c}_{N_c} = \mathbb{I}^{g_\varphi}_{N_\varphi}$, *where* $\mathbb{I}^{g_\varphi}_{N_\varphi} = \{I^{\varphi}_{N_\varphi} \mid \varphi \in \Phi\}$.

Corollary 2. $\mathbb{I}_{\mathbb{T}_{\mathbb{LC}}} \cap \mathbb{I}^{g_c} = \mathbb{I}^{g_\varphi}$, *where* $\mathbb{I}^{g_\varphi} = \{I^\varphi \mid \varphi \in \Phi, \varphi(x) + \varphi(1-x) = 1, x \in [0,1]\}$.

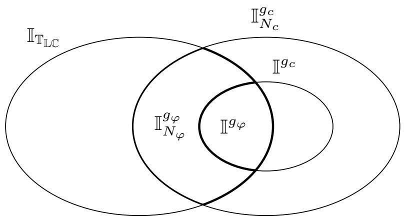

$\mathbb{I}_{\mathbb{T}_{\mathrm{LC}}}$
$\mathbb{I}^{g_c}_{N_c}$
$\mathbb{I}^{g_c}$
$\mathbb{I}^{g_\varphi}_{N_\varphi}$
$\mathbb{I}^{g_\varphi}$

Fig. 2: Intersection of the class of R-implications based on left-continuous t-norm and the class of I^g_N implications with continuous generator g and negation N.

4 Conclusion

We have investigated the residual operator of the conjunction. This conjunction was based on empirical data. It turned out that the only operator based on generated t-norm is an implication and it belongs to the less known class of generated implications I^g_N where $N(x) \neq N_S(x)$. We have studied the properties of I^g_N-implications. We showed that although the classes I^g_N and (S,N)-implications are similar, they are not identical. And also, we examined the relationship between classes I^g_N and R-implications based on left-continuous t-norms. In the future we plan to model implications in human language via fitting residual operators to empirical data.

Acknowledgement. The work was supported by the BUT project FIT-S-14-2486.

References

1. M. Baczynski & B. Jayaram (2008): *Fuzzy Implications. Studies in Fuzziness and Soft Computing* 231, Springer, Berlin, doi:10.1007/978-3-540-69082-5.
2. J. C. Fodor & M. R. Roubens (1994): *Fuzzy Preference Modelling and Multicriteria Decision Support.* Theory and Decision Library D:, Kluwer Academic Publishers, Dordrecht, doi:10.1007/978-94-017-1648-2.
3. D. Hliněná & V. Biba (2012): *Generated fuzzy implications and fuzzy preference structures. Kybernetika* 48(3), pp. 453–464.
4. D. Hliněná & V. Havlena (2016): *Fitting Aggregation Operators.* In: *Mathematical and Engineering Methods in Computer Science, Lecture Notes in Computer Science, Vol. 9548*, Springer International Publishing, pp. 42 – 53, doi:10.1007/978-3-319-29817-7_5.
5. S. Jenei (1998): *On Archimedean triangular norms. Fuzzy Sets and Systems* 99(2), pp. 179 – 186, doi:10.1016/S0165-0114(97)00021-3.
6. S. Jenei & J. C. Fodor (1998): *On continuous triangular norms. Fuzzy Sets and Systems* 100(13), pp. 273 – 282, doi:10.1016/S0165-0114(97)00063-8.
7. E. P. Klement, R. Mesiar & E. Pap (2000): *Triangular Norms.* Kluwer Academic Publishers, Boston, doi:10.1007/978-94-015-9540-7.

8. D. Smutná (1999): *On many valued conjunstions and implications*. *Journal of Electrical Engineering* 1999(50), p. 8.
9. R. R. Yager (2004): *On Some New Classes of Implication Operators and Their Role in Approximate Reasoning*. *Information Sciences* 167(1-4), pp. 193–216, doi:10.1016/j.ins.2003.04.001.

Avalanche Effect in Improperly Initialized CAESAR Candidates

Martin Ukrop and Petr Švenda

Centre for Research on Cryptography and Security,
Faculty of Informatics,
Masaryk University, Brno, Czech Republic
`mukrop@mail.muni.cz, svenda@fi.muni.cz`

Abstract. Cryptoprimitives rely on thorough theoretical background, but often lack basic usability features making them prone to unintentional misuse by developers. We argue that this is true even for the state-of-the-art designs. Analyzing 52 candidates of the current CAESAR competition has shown none of them have avalanche effect in authentication tag strong enough to work properly when partially misconfigured. Although not directly decreasing their security profile, this hints at their security usability being less than perfect.[1]

1 Introduction

Nowadays, experts realize that having cryptography attack-resistant from the theoretical point of view is not sufficient since many attacks are caused by improper use of otherwise sound cryptographic primitives. Developers routinely produce horrendous implementations (at least from the point of security) when they neglect to properly set initialization vectors or ignore the requirement of unique sequence numbers. Recently, Cairns and Steel outlined their vision for developer-resistant cryptography [5] with designs that cannot be misused by the programmer.

The question the security-optimist would ask is: *Is that not the case only for old primitives, old protocols and old designs? Are new designs also prone to developer misuse?* We argue the problem is still open – we tested 52 participants of the current state-of-the-art cryptographic competition by checking the avalanche effect of the candidates in settings simulating partial misconfiguration.

It is long known that cryptographic primitives such as ciphers, hash functions and message authentication codes should produce seemingly random outputs. Further requirements ask for outputs to change unpredictably with respect to changes in the input. The strict avalanche criterion, as introduced by Webster and Tavares in 1985 [23], is one way to formalize this. It is satisfied if, whenever a single input bit is complemented, each of the output bits changes with a 50% probability. It is commonly used for assessing the security of hash functions, though using it as a randomness test has also been done before [6].

[1] Paper details available at `crcs.cz/papers/memics2016`

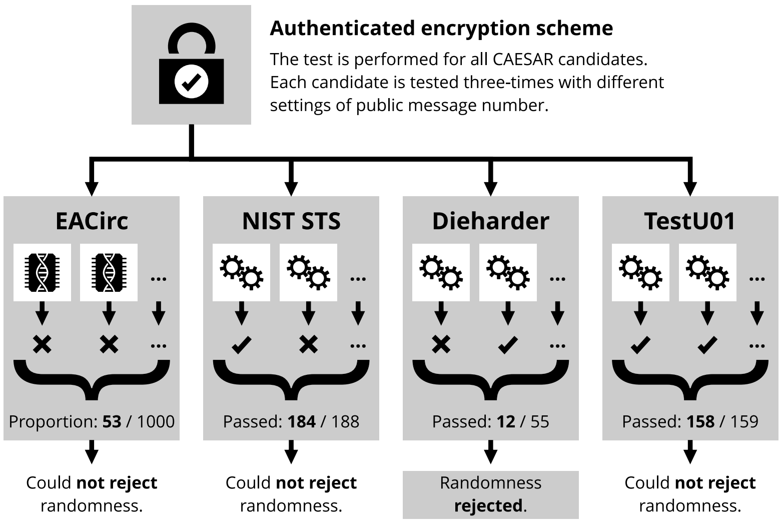

Authenticated encryption scheme
The test is performed for all CAESAR candidates. Each candidate is tested three-times with different settings of public message number.
EACirc
NIST STS
Dieharder
TestU01
Proportion: 53 / 1000
Passed: 184 / 188
Passed: 12 / 55
Passed: 158 / 159
Could not reject randomness.
Could not reject randomness.
Randomness rejected.
Could not reject randomness.

Fig. 1. High-level overview of the performed experiments. Firstly, a CAESAR cipher is used to generate a stream of authentication tags. The randomness of this stream is then assessed by 4 tools (EACirc and 3 statistical testing suites). If the design is good, it should exhibit an avalanche effect strong enough for the stream to look random.

In this paper, we scrutinize submissions of the ongoing CAESAR competition (*Competition for Authenticated Encryption: Security, Applicability, and Robustness*) [4]. The authentication tags produced by all candidates are examined using four different software tools: three standard statistical batteries (NIST STS [14], Dieharder [3] and TestU01 [11]) and a novel genetically-inspired framework (EACirc [22]). The overview of the main experiment idea is depicted in Figure 1.

The analysis was done separately for three different settings of the public message number (fixing it to zero, using a counter and generating unique random value each time). It turned out that none of the tested CAESAR candidates had an avalanche effect strong enough to produce random-looking tags in the most seriously misconfigured case with zero public message numbers (thus avalanching from only a very few changed bits in plaintext). In the case of counter-based and random public message numbers, the ciphers fared much better.

Firstly, in Section 2, the paper gives an overview of the related research. Then the basics of authenticated encryption are explained along with the essentials of CAESAR competition (Section 3). The following sections summarize the way of generating the tested data (Section 4) and the tools used for the analysis (Section 5). Lastly, the results and their interpretation are given in Section 6.

2 Related research

As CAESAR competition is an on-going initiative with many submissions, there are still not many publications thoroughly examining the security of all the proposed algorithms. F. Abed et al. [1] give an excellent overview of the candidates along with a classification with regard to their core primitives. K. Hakju and K. Kwangjo [8] discuss the features of authenticated encryption and predict the essential characteristics of the submissions to survive the CAESAR competition.

Probably the most comprehensive competition-wide analysis so far has been done by M. Saarinen [15] using the BRUTUS automatic cryptanalytic framework. Deeper analysis exists only on a per-candidate basis. For example, R. Ankele in his Ph.D. thesis [2] analyses the *COPA* authenticated encryption composition scheme used in several CAESAR candidates. M. Nandi in his 2014 paper [13] demonstrates a forging attack on *COBRA* and *POET* ciphers.

Numerous works tackled the problem of assessing randomness of outputs from other cryptoprimitives. E. Simion [16] gave a nice overview of statistical requirements for cryptographic primitives in his work. The Ph.D. thesis of K. Jakobsson [9] gives both a good theoretical background and a comparison of commonly available tools for random number testing. Its results are based on assessing a variety of pseudo-random and quantum random number generators.

Cryptographic competitions are often the target of these analyses since the unified function API allows for effortless evaluation of a high number of schemes. M. Turan et al. [19] performed a detailed examination of eStream phase 2 candidates (both full and reduced-round) with NIST STS and structural randomness tests, finding six ciphers deviating from expected values. In 2010, Doganaksoy et al. [7] applied the same battery, but only a subset of tests to SHA-3 candidates with a reduced number of rounds as well as only to their compression functions.

A different strategy is employed in the EACirc framework – it uses a genetically-inspired process to find a successful distinguisher (function capable for differencing between cipher output and random stream). The framework has been used for assessing the randomness of outputs produced by the round-limited eSTREAM and SHA-3 candidates [22, 18]. Although still falling behind in some cases, this approach surpasses NIST STS in a few instances.

3 Authenticated encryption

A cryptosystem for authenticated encryption simultaneously provides confidentiality, integrity, and authenticity assurances on data – decryption is combined in a single step with integrity verification. Authenticated ciphers are often built as various combinations of block ciphers, stream ciphers, message authentication codes, and hash functions. There are many examples commonly used today, such as the *Galois/counter mode* (GCM) [12] based on block ciphers.

Combining confidentiality and integrity assurances into a single scheme has tremendous advantages as combining a confidentiality mode with an authentication mode could be error prone and difficult[2]. Therefore, following a long tradition of cryptography competitions, CAESAR [4] aims to create a portfolio of authenticated encryption systems intended for wide public adoption.

Each submission in CAESAR specifies a family of authenticated ciphers. Family members differ only in parameters (e.g. key length, the number of internal rounds). There were 56 different designs submitted to the first round. Taking into account all possible parameter sets, this amounts to 172 independent schemes. Till the announcement of the second-round candidates, 9 ciphers were withdrawn by their authors. On July 7th 2015, 29 ciphers were chosen for the second round. Later, on August 15th 2016, 15 ciphers out of these were selected for the third round.

Our goal was to test as many authenticated encryption schemes as possible. Using CAESAR candidates enabled us to test many ciphers and many configurations automatically due to the shared API. All the candidate source codes were taken from the 1st-round SUPERCOP repository managed by eBACS [21].

In the end, there were 168 different ciphers tested in all performed experiments. From 172 submitted independent schemes (56 designs with different parameter sets), 6 were not tested. Firstly, we could not get the *AVALANCHE* candidates working properly (segmentation fault while running). Secondly, *Julius* did not compile due to problems with the inclusion of the external AES routines provider. Thirdly, *POLAWIS* seemed not to have followed the prescribed API. Lastly, the implementation of *PAES* is probably faulty, since it did not pass our encrypt-decrypt sanity test. We might have been able to fix most of these cases, but doing so would require extensive interventions in the code increasing the possibility of error. Apart from the submitted candidates, we tested 2 versions of *AES/GCM* referenced by the CAESAR committee as a design baseline.

4 Tested data streams

The aim of the performed experiments is to assess randomness of authentication tags produced by many authenticated encryption schemes. The same analysis could also be performed on produced ciphertext, but that it out of scope of this paper. In particular, we inspect tags provided by CAESAR candidates in three independent scenarios differing in public message number setting. An overview of tag generation is given below and in Figure 2.

The cipher has 5 inputs: plaintext (encrypted and authenticated user input), associated data (authenticated user input), key, secret message number (secret nonce) and public message number (public nonce). The produced tag (extra ciphertext bytes when compared to the plaintext length) is determined by the cipher design. In the majority of the cases, this means 128 bits (16 bytes), but some candidates produce shorter tags (2, 4, 8 or 12 bytes). These tags were concatenated to form a continuous stream suitable for randomness assessment.

[2] *"It is very easy to accidentally combine secure encryption schemes with secure MACs and still get insecure authenticated encryption schemes."* [10]

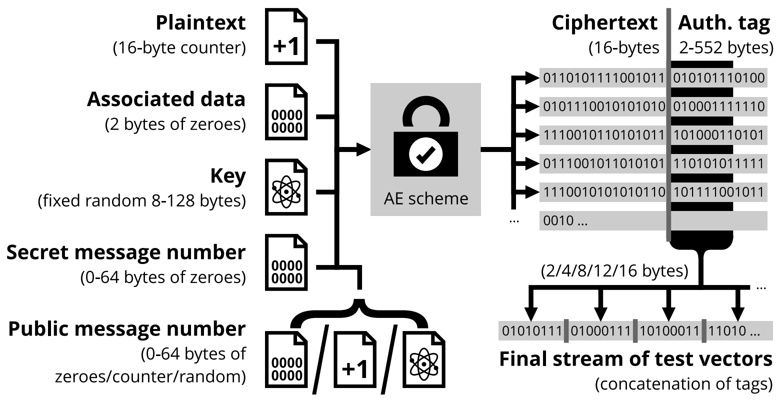


Fig. 2. The process of creating the tested data streams by individual CAESAR candidates. The cipher is initialized as depicted in the diagram. The produced authentication tags are then concatenated to form a continuous stream suitable for randomness analysis.

From the nature of the arguments prescribed by the CAESAR API, the public message number is probably the argument to be most easily (unintentionally) misused. Security requirements for keys are well known, secret message numbers are usually not used, plaintext and associated data are mostly self-explanatory. Public message numbers are sometimes required to be unique (to have properties of nonces), but sometimes this is not necessary. In a way, we deem testing different modes of public message numbers as examining the robustness of the cipher design. The fields were initialized as follows:

- **Key**
  The key value was taken randomly but was fixed. For EACirc (one of the used tools), 1 000 independent runs used different keys to allow for variation (otherwise, the same numerical results would be produced).
- **Associated data, secret message number**
  We used two bytes of associated data; the length of the secret message number was determined by the cipher or the parameter set. Both fields' values were fixed to binary zeros. Note that only three ciphers used secret message numbers.
- **Public message number**
  This was the only parameter explored in different settings:
  - Fixed to a string of binary zeros for the whole time.
  - Increasing as a counter – each value unique but similar to others.
  - Having each value completely random.
- **Plaintext**
  The plaintext was 16 bytes long, formatted as a single counter starting from

zero. We could not use fixed-value plaintext, because, in the case of fixed-value public message numbers, the produced tags would be identical (considering settings of the other arguments). A plaintext of binary zeros would have been possible in the other two modes for public message numbers, but we refrained from doing so to keep the experiments as comparable as possible (with as similar settings as possible).

In summary, if we denote the cipher as a function $F(plain, adata, key, smn, pmn)$ producing the authentication tag (the ciphertext is not used in our analysis, but inspecting it would also be interesting), the final analyzed stream in the scenario with random public message numbers looks as follows:

$$\begin{aligned}\text{Stream} = {} & F(0, 0, rand_A, 0, rand_1) \,||\, F(1, 0, rand_A, 0, rand_2) \,||\\ & F(2, 0, rand_A, 0, rand_3) \,||\, F(3, 0, rand_A, 0, rand_4) \,||\, ...\end{aligned}$$

5 Randomness testing tools

The most common way of testing randomness is using statistical testing. From the multitude of available batteries, we used the following three: NIST STS (older, yet still commonly used and a valid NIST standard), Dieharder (modern framework reimplementing other suites as well as adding brand new tests) and TestU01 (another modular framework implementing many tests).

Although the p-value of a randomness test focusing on a single characteristic has a clear statistical interpretation, the interpretation of results produced by testing suites is somewhat problematic. We need to determine what number of failed tests allows us to reject randomness of the assessed sequence while respecting the chosen significance level. For this, we use the methods proposed in 2015 by M. Sýs et al. [17].

For all experiments, we chose the significance level of $\alpha = 1\%$, which is the default value for NIST STS [14]. This keeps the type I. error (false positives) reasonably low while preventing the type II. error (false negatives) to reach too high values.

We used NIST STS version 2.1.1 with the default parameters (block lengths) for all tests. To comply with the minimal required stream length for individual tests [14], we tested 100 independent 1 000 000 bit long sequences for each candidate. In summary, NIST STS used about 12 MiB (about 700 000 tags) of data from each candidate for each test.

Dieharder version 3.31.1 was used. The two parametrizable tests were configured with recommended values. The length of the input stream processed by Dieharder varies from test to test. The humblest (Diehard 3D-sphere test) required about 48 kiB, while the greediest one (Bit distribution test) took about 9.2 MiB. To ensure the best possible comparability with the other test suites, we again analyzed 100 independent samples of the input. In summary, Dieharder tests used between 4.7 MiB (about 300 000 tags) and 916 MiB (about 60 000 000 tags) of input data for each candidate (depending on the test).

TestU01 was used in version 1.2.3. The most relevant sub-batteries are Rabbit, Alphabit and BlockAlphabit. These are intended for testing finite binary sequences. The length of the input stream taken by TestU01 can be set arbitrarily. To have an amount of data comparable to the other used batteries, we chose to process 2^{30} bits for each test. In summary, TestU01 thus used about 128 MiB (about 8 400 000 tags) of input data for each test.

EACirc represents a completely different approach to testing data randomness: The main idea is to use supervised learning techniques based on evolutionary algorithms to design and further optimize a successful distinguisher – a test determining whether its input comes from a truly random source or not. The distinguisher is represented as a hardware-like circuit consisting of simple interconnected functions. The used settings cause EACirc to process approximately 2.24 MiB of data produced by the tested cryptoprimitive for a single EACirc run. This amounts to about 2.24 GiB (about 150 000 000 tags) of data for a single experiment with 1 000 runs.

6 Results and interpretation

A selection of the numerical results can be seen in Table 1. The table aims for a representative selection of the interesting cases including all categories from the reference schemes to the algorithms that passed to the third (currently last) round. For the complete numerical results and detailed reasoning, see [20].

Firstly, let us compare the outcomes for the three inspected public message number modes. We expected the random-valued to perform the best, followed by counter-based and then by zero-fixed public message numbers. We reasoned that the more differences there will be among the used values, the easier it will be for the cipher to produce a random-looking tag (since it has more entropy to start from). As stated in the submission call, the ciphers were allowed to lose all security in case of reused (public message number, private message number)-pair under the same key. Nevertheless, we expected some (albeit not many) ciphers will be able to retain the apparent randomness of the produced tag – even though it would require an adamant avalanche effect (all arguments are identical apart from a few bits in plaintext).

From the conducted experiments we see that the primary hypothesis (random values performing better than a counter and much better than zeros) was confirmed. However, none out of the tested candidates passed with the public message numbers fixed to zero. The single bit change in plaintext with all other arguments fixed might not have been enough to cause the avalanche effect needed to produce a tag looking sufficiently random.

Secondly, let us inspect the results for the individual candidates (see Table 1). Tags of just five ciphers (*AES/GCM*, *Marble*, *AEC-CMCC*, *AES-CPFB*, *Raviyoyla*) were distinguishable from random streams with counter-valued public message numbers. Three of these ciphers (*Marble*, *AES-CMCC*, *AES-CPFB*) also failed in the random-valued scenario. The evidence is still too weak to deem the designs insecure – it may merely be the case they produce a constant delim-

Category (CAESAR round)	**Cipher** (official name)	**Candidate ID** (as used in SUPERCOP [21])	PMN fixed to zero				PMN counter-based				PMN truly random			
			EACirc (proportion)	**NIST STS** (x/188)	**Dieharder** (x/55)	**TestU01** (x/159)	**EACirc** (proportion)	**NIST STS** (x/188)	**Dieharder** (x/55)	**TestU01** (x/159)	**EACirc** (proportion)	**NIST STS** (x/188)	**Dieharder** (x/55)	**TestU01** (x/159)
Reference candidates	AES-GCM	aes128gcmv1	1.000	23	1	11	0.014	187	52	157	0.019	187	52	158
	AES-GCM	aes256gcmv1	1.000	71	1	13	0.013	188	51	156	0.007	188	54	158
Withdrawn candidates	Calico	calicov8	0.013	128	3	8	0.009	186	55	156	0.015	188	53	158
	Marble	aes128marble4rv1	0.016	160	16	6	0.010	168	14	8	0.010	160	16	6
First-round candidates	AES-CMCC	cmcc22v1	0.008	53	1	4	0.011	46	2	8	0.008	187	54	156
	AES-CMCC	cmcc24v1	0.005	43	3	3	0.008	188	50	155	0.023	186	50	152
	AES-CMCC	cmcc42v1	0.011	87	2	3	0.008	86	2	4	0.015	182	54	154
	AES-CMCC	cmcc44v1	0.008	85	4	5	0.013	183	52	155	0.007	188	53	152
	AES-CMCC	cmcc84v1	0.014	147	7	4	0.009	184	53	156	0.007	182	48	158
	Enchilada	enchilada128v1	1.000	71	2	15	0.017	187	53	157	0.010	186	52	155
	Enchilada	enchilada256v1	1.000	77	1	11	0.013	188	54	156	0.016	188	53	155
	Raviyoyla	raviyoylav1	1.000	22	2	7	1.000	148	28	24	0.295	186	51	144
Second-round candidates	TriviA-ck	trivia0v1	0.999	140	5	8	0.015	186	52	157	0.005	187	55	154
	TriviA-ck	trivia128v1	0.993	158	12	8	0.017	188	53	158	0.009	188	54	157
Third-round candidates	AEZ	aezv1	0.014	169	15	9	0.015	187	52	155	0.010	188	52	157
	AEZ	aezv3	0.016	164	13	6	0.011	188	53	157	0.009	185	50	156

Table 1. The selection of ciphers with interesting results from different categories (from reference schemes to 3rd round candidates). The numbers in columns of statistical batteries represent the number of passed tests (should be close to all for random stream), while EACirc displays the ratio of runs rejecting randomness (should be around 0.010 for random stream). For the ease of comprehension, if the result rejects randomness of the particular stream, the cell is gray-colored. Note that this is only a subset of the tested candidates (most of the omitted ones have results similar to those of *AEZ*). For complete numerical results and threshold values, see [20].

iter between the ciphertext and tag, violating the statistical randomness of the created tag. To draw any conclusions, a detailed inspection of the ciphers would need to be performed. It is, however, worth mentioning that no candidates failing in either counter- or random-valued scenario were selected by the CAESAR committee to the second round of the competition.

Apart from the findings for the CAESAR candidates, the results allow us to gain insights into the capabilities of the used randomness testing tools. Based on the previous works [18], we expected the randomness distinguishing abilities of EACirc and NIST STS will be similar while both will be surpassed by Dieharder and TestU01. On the one hand, the observed results showed many deficiencies of EACirc – it performed worse than NIST STS in given tested scenarios. On the other, all three statistical batteries achieved comparable results. However, before any conclusions on the quality of the batteries are drawn, one has to be aware there are many domains in which these tools remain incomparable. They inspect different amounts of data and have different modes of operation (batteries see the stream as a whole, EACirc processes short, distinct test vectors).

There is one case contrary to the general behavior observed above (see Table 1): *Raviyoyla* with randomly initialized public message numbers for each test vector seems to be successfully rejected from the random stream by EACirc although none of the statistical batteries support such result. It appears very promising but also requires additional inspection and enhanced testing to announce a case of EACirc surpassing all tested statistical batteries.

The results lead us to several interesting hypotheses requiring further inspection. The candidates failing in randomness tests would deserve a deeper manual inspection to prove their potential (in)security. The used statistical testing suites themselves would be an interesting target for further research. It turned out that interpretation of test suites is quite difficult, and thorough research on test interdependence is necessary. Another perspective direction would be weakening the cipher designs (e.g. by limiting the number of internal rounds) to achieve a fine-grained comparison of the used tools.

7 Summary

We have set off to examine modern authenticated encryption systems from the point of resistance against common developer misconfiguration. In the end, we assessed outputs from 168 distinct schemes (all but six CAESAR submissions) in three different configurations using multiple software tools (NIST STS, Dieharder, TestU01 and EACirc).

We examined a scenario with random (but fixed) keys, counter-based plaintext and three different settings of public message numbers. As expected, tags produced in configurations with random public message numbers fared better than the ones from counter-based configurations. Both did better than tags from fixed-value public message numbers – no submission had an avalanche effect strong enough to produce random-looking tags in the scenario where all test vectors had the same public message numbers.

Only three CAESAR submissions (*Marble*, *AEC-CMCC*, *Raviyoyla*) failed to produce seemingly random tags with counter-based public message numbers. For entirely random public message numbers, only *Marble* failed convincingly. *AEC-CMCC* achieved a borderline value in Dieharder and passed in other tools. *Raviyoyla* seems to have failed according to EACirc – this case is suspicious and worth of further investigation, since it is the only case where EACirc surpassed the other tools. Importantly, none of these candidates made it to the second round of the competition (indirectly supporting our results).

Regarding the tools used for tag evaluation, EACirc seemed to be the least suitable for the given task, being beaten by all the statistical batteries. The batteries themselves (NIST STS, Dieharder and TestU01) produced comparable results. The only exception is the case of *Raviyoyla*, in which EACirc seems to have outperformed all the other tools. However, when making comparisons, one has to take into account the amount of data inspected by each tool and their different modes of operation.

All in all, not even the state-of-the-art authenticated encryption designs do not have avalanche effect strong enough in the case with zero-fixed public message numbers. Although not forming a direct practical attack on the ciphers, it breaks semantic security of the scheme, since the attacker is able to distinguish two messages based on the leakage present in inspected scenarios. A security-optimist from the introduction may see this as an interesting area for further improvements.

Acknowledgments We acknowledge the support of Czech Science Foundation, project GA16-08565S. Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum provided under the programme Projects of Projects of Large Research, Development, and Innovations Infrastructures (CESNET LM2015042), is greatly appreciated.

References

1. Farzaneh Abed, Christian Forler & Stefan Lucks (2014): *General Overview of the Authenticated Schemes for the First Round of the CAESAR Competition. Cryptology ePrint Archive*. Available at `http://ia.cr/2014/792`.
2. Robin Ankele (2015): *Provable Security of Submissions to the CAESAR Cryptographic Competition*. Master thesis, Graz University of Technology. Available at `https://securewww.esat.kuleuven.be/cosic/publications/thesis-263.pdf`.
3. Robert G. Brown (2004): *Dieharder: A Random Number Test Suite*. `http://www.phy.duke.edu/\%7Ergb/General/dieharder.php`.
4. CAESAR committee (2013): *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*. http://competitions.cr.yp.to/caesar-call.html.
5. Kelsey Cairns & Graham Steel (2014): *Developer-resistant cryptography*. In: *A W3C/IAB workshop on Strengthening the Internet Against Pervasive Monitoring (STRINT)*.

6. Julio Cesar Hernandez Castro, José María Sierra, Andre Seznec, Antonio Izquierdo & Arturo Ribagorda (2005): *The strict avalanche criterion randomness test. Mathematics and Computers in Simulation* 68(1), pp. 1–7, doi:`10.1016/j.matcom.2004.09.001`.
7. Ali Doganaksoy, Baris Ege, Onur Koçak & Fatih Sulak (2010): *Statistical Analysis of Reduced Round Compression Functions of SHA-3 Second Round Candidates. IACR Cryptology ePrint Archive*. Available at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.187.814\&rep=rep1\&type=pdf`.
8. Kim Hakju & Kim Kwangjo (2014): *Who can survive in CAESAR competition at round-zero.* In: *The 31th Symposium on Cryptography and Information Security Kagoshima*, pp. 21–24. Available at `http://caislab.kaist.ac.kr/publication/paper_files/2014/SCIS2014_HJ.pdf`.
9. Krister Sune Jakobsson (2014): *Theory, Methods and Tools for Statistical Testing of Pseudo and Quantum Random Number Generators.* Ph.D. thesis, Linköpings universitet, Sweden. Available at `http://liu.diva-portal.org/smash/record.jsf?pid=diva2\%3A740158\&dswid=9282`.
10. Tadayoshi Kohno, John Viega & Doug Whiting (2003): *The CWC authenticated encryption (associated data) mode. ePrint Archives.* Available at `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/cwc/cwc-spec.pdf`.
11. Pierre L'Ecuyer & Richard Simard (2007): *TestU01: A C Library for Empirical Testing of Random Number Generators. ACM Transactions on Mathematical Software* 33(4), doi:`10.1145/1268776.1268777`.
12. David McGrew & John Viega (2004): *The Galois/Counter Mode of Operation (GCM). Submission to NIST.* Available at `http://siswg.net/docs/gcm_spec.pdf`.
13. Mridul Nandi (2014): *Forging Attacks on Two Authenticated Encryption Schemes COBRA and POET.* In: *Advances in Cryptology – ASIACRYPT 2014*, 8873, Springer Berlin Heidelberg, pp. 126–140, doi:`10.1007/978-3-662-45611-8_7`.
14. Andrew Rukhin et al. (2000): *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.* Technical Report, National Institute of Standards and Technology (NIST). Available at `http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf`.
15. Markku-Juhani O. Saarinen (2015): *The BRUTUS automatic cryptanalytic framework. Journal of Cryptographic Engineering* 6(1), pp. 75–82, doi:`10.1007/s13389-015-0114-1`.
16. Emil Simion (2015): *The Relevance of Statistical Tests in Cryptography. IEEE Security & Privacy*, pp. 66–70, doi:`10.1109/MSP.2015.16`.
17. Marek Sýs, Zdenk Říha, Václav Matyáš, Kinga Márton & Alin Suciu (2015): *On the Interpretation of Results from the NIST Statistical Test Suite. Romanian Journal of Information Science and Technology* 18(1), pp. 18–32.
18. Marek Sýs, Petr Švenda, Martin Ukrop & Vashek Matyáš (2014): *Constructing empirical tests of randomness.* In: *SECRYPT 2014 Proceedings of the 11th International Conference on Security and Cryptography*, SCITEPRESS Science and Technology Publications, pp. 229–237, doi:`10.5220/0005023902290237`.
19. Meltem Sonmez Turan, Ali Doganaksoy & Çagdas Çalik (2008): *On Statistical Analysis of Synchronous Stream Ciphers.* Ph.D. thesis, The Middle East Technical University. Available at `http://etd.lib.metu.edu.tr/upload/12609581/index.pdf`.
20. Martin Ukrop (2016): *Randomness analysis in authenticated encryption systems.* Master thesis, Faculty of Informatics, Masaryk University. Available at `http://is.muni.cz/th/374297/fi_m/`.

21. Virtual Applications and Implementations Research Lab (2008): *SUPERCOP: System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives.* Available at `http://bench.cr.yp.to/supercop.html`.
22. Petr Švenda, Martin Ukrop & Vashek Matyáš (2014): *Determining cryptographic distinguishers for eStream and SHA-3 candidate functions with evolutionary circuits.* In: *E-Business and Telecommunications*, 456, Springer Berlin Heidelberg, pp. 290–305, doi:`10.1007/978-3-662-44788-8_17`.
23. A. F. Webster & S. E. Tavares (1986): *On the Design of S-Boxes*, pp. 523–534. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:`10.1007/3-540-39799-X_41`.

Using the Context of User Feedback in Recommender Systems

Ladislav Peska

Faculty of Mathematics and Physics
Charles University
Malostranske namesti 25, Prague, Czech Republic
peska@ksi.mff.cuni.cz

Abstract. Our work is generally focused on recommending for small or medium-sized e-commerce portals, where explicit feedback is absent and thus the usage of implicit feedback is necessary. Nonetheless, for some implicit feedback features, the *presentation context* may be of high importance. In this paper, we present a model of relevant contextual features affecting user feedback, propose methods leveraging those features, publish a dataset of real e-commerce users containing multiple user feedback indicators as well as its context and finally present results of purchase prediction and recommendation experiments. Off-line experiments with real users of a Czech travel agency website corroborated the importance of leveraging *presentation context* in both purchase prediction and recommendation tasks.

1 Introduction

We face continuous growth of information on the web. The volume of products, services, offers or user-generated content rise every day and the amount of data on the web is virtually impossible to process directly by a human. Automation of web content processing is necessary. Recommender systems aim to learn specific preferences of each distinct user and then present them surprising, unknown, but interesting and relevant items. Users do not have to specify their queries directly as in a search engine. Instead, their preferences are learned from their ratings (explicit feedback) or browsing behavior (implicit feedback).

However, some domains, e.g., small or medium-sized e-commerce enterprises, introduce specific problems and obstacles making the deployment of recommender systems more challenging. Let us list some of the obstacles:

– High concurrency has a negative impact on user loyalty. Typical sessions are very short, users quickly leave to other vendors, if their early experience is not satisfactory enough. Only a fraction of users ever returns.
– For those single-time visitors, it is not sensible to provide any unnecessary information such as ratings, reviews, registration details etc.
– Consumption rate is low, users often visit only a handful of objects.

All the mentioned factors contribute to the data sparsity problem. Although the total number of users can be relatively large (hundreds or thousands per day), explicit feedback is very scarce and implicit feedback is also available only for a fraction of objects. Furthermore, as the space of potential implicit feedback features is quite large, it might be challenging to select the right approach to utilize them. In general, some rapidly learning algorithms, capable to recommend from only a limited feedback are needed.

Despite these obstacles, the potential benefit of deploying recommender systems is considerable, it can contribute towards better user experience, increased user loyalty and consumption and thus also improve vendors key success metrics.

Our work within this framework aims to bridge the data sparsity problem and the lack of relevant feedback by modelling and utilizing novel/enhanced sources of information, foremost implicit user feedback features.

More specifically, the work presented in this paper focuses on the question how to define and collect user preference [1] in scenarios, where we cannot invasively ask users to provide it (i.e., there is no explicit feedback), but we can interfere with the website source code (and thus observe any type of user actions).

1.1 Main contributions

Main contributions of this paper are:

- Model of user feedback features enriched by the context of the page and device.
- Methods interpreting this model of user feedback as a proxy of user engagement.
- Experiments on real users of a Czech travel agency.

We also provide datasets of user feedback, contextual features and objects attributes for the sake of repeatability and further experiments.

2 Related Work

2.1 Implicit Feedback in Recommender Systems

Contrary to explicit feedback, implicit feedback approach merely monitors user behavior without intruding it. Implicit feedback features varies from simple user visit or play counts to more sophisticated ones like scrolling or mouse movement tracking [5, 16]. Due to its effortlessness, data are obtained in much larger quantities for each user. On the other hand, they are inherently noisy and harder to interpret [4].

Our work lies a bit further from the mainstream of the implicit feedback research. To the best of our knowledge, vast majority of researchers focus on

[1] Please note that we will freely interchange *user preference* and *user engagement* concepts.

interpreting single type of implicit feedback, e.g., [17], or proposing various recommending algorithms while using predefined implicit feedback, e.g., [3, 4, 13, 14].

Our research goes towards modelling users preference and engagement based on multiple types of implicit feedback. We can trace such efforts also in the literature. One of the first paper mentioning implicit feedback was Claypool et al. [1] comparing several implicit preference indicators against explicit user rating. This paper was our original motivation to collect and analyze various types of user behavior to estimate user preference. More recently Yang et al. [16] analyzed several types of user behavior on YouTube. Authors described both positive and negative implicit indicators of preference and proposed a linear model to combine them.

In our previous work, we defined a complex set of potentially relevant set of implicit user feedback features with respect to the e-commerce domain and provided software component collecting it [7]. We also show that using multiple types of feedback features provides significant improvements over using single feedback feature in purchase prediction task [9, 10]. However, in our previous works we used feedback features in its raw form without any respect to the context of the currently visited page or users browsing device, which can potentially affect user behavior.

2.2 Context Awareness

In this paper, we focus on the *presentation context* (we will also refer to it as a *context of page and device*) rather than more commonly utilized context of the user. We follow the hypothesis that if the same information is presented in a different form, the users response might differ as well. We can trace some notions of *presentation context* in the literature. For example, Radlinski et al. [12] and Fang et al. [3] considered object position as a relevant context for clickstream events. Also Eckhardt et al. [2] proposed to consider user ratings in the context of other objects available on the current page.

Closest to our work is the approach by Yi et al. [17], proposing to use *dwell time* as an indicator of user engagement. Authors discussed the role of several contextual features, e.g., content type, device type or article length on extitdwell time feedback. Nonetheless, there are several substantial differences between our approaches. First, Yi et al. focused solely on the *dwell time* and considered normalized *dwell time* directly as a proxy to the user engagement. Our approach is to integrate multiple indicators of user preference by using machine learning methods. Furthermore, the list of proposed contextual features are different as both the domains and data acquisition methods differ. We introduced, e.g., features based on page and browser window dimensions, not used in Yi et al. Last, Yi et al. proposed to utilize context merely to normalize *dwell time*, however we include context in the feature engineering process.

3 Materials and Methods

3.1 Outline of Our Approach

As already mentioned, the key part of our work aims on implicit user feedback, user preference and its usage in recommender systems. In traditional recommender systems, user u rates some small sample S of all objects O, which is commonly referred as user preference $r_{u,o} : o \in S \subset O$. The task of traditional recommender systems is to build suitable user model, capable to predict ratings $\hat{r}_{u,o'}$ of all objects $o' \in O$. If there are no explicit feedback, user preference must be inferred from implicit feedback. We denote this as inferred preference $\overline{r}_{u,o}, o \in S$. If there is a single feedback feature $f_{u,o}$, the preference is usually inferred directly $f_{u,o} \approx \overline{r}_{u,o}$ [4]. However, some more elaborated approaches are necessary, if there are multiple feedback features $[f_1, ..., f_i]$.

Our approach is based on the hypothesis that purchases represent fully positive user preference:

$$r_{u,o} := \begin{cases} 1 \text{ IF } u \text{ bought } o \\ 0 \text{ OTHERWISE} \end{cases} \tag{1}$$

In another words, we promote *purchases* to the level of explicit user ratings. Unfortunately, the density of *purchases* is very low in the e-commerce [2] , so it is impractical to base recommendations directly on *purchases*. We also suppose that other visited, but not purchased objects reflect some level of user engagement, which can be inferred from other implicit feedback features. Our aim is to show that such inferred user preference provides better source information of a recommender system than binary visits or purchases. Our approach (see Fig. 1) is divided into three steps.

In the first step, feature engineering (Fig. 1c), we combined raw feedback features $F : [f_1, ..., f_i]$, presentation context features $C : [c_1, ..., c_j]$ and user statistics into a set of derived feedback features $\overline{F} : [\overline{f}_1, ..., \overline{f}_i]$. Details of this procedure can be found in Section 3.2.

In the second step, the set of derived feedback features is transformed into the inferred user preference $[\overline{f}_1, ..., \overline{f}_i]_{u,o} \rightarrow \overline{r}_{u,o}$ (Fig. 1d). The transformation is made via machine learning methods aiming to predict, whether the object o was purchased by the user u, given the feedback $[\overline{f}_1, ..., \overline{f}_i]_{u,o}$. More details can be found in Section 3.3.

Finally, we use $\overline{r}_{u,o}$ as an input of recommender systems to provide user with the list top-k objects (Fig. 1e). The description of used recommending algorithms can be found in Section 3.4.

3.2 Implicit User Feedback and Presentation Context

In this section, we will describe the model of implicit feedback F, presentation context C and feature engineering steps transforming it into derived feature set$\overline{F}$. Raw feedback features and presentation context features are listed in Table 1

[2] Less than 0.4% of the visited objects were purchased in our dataset.

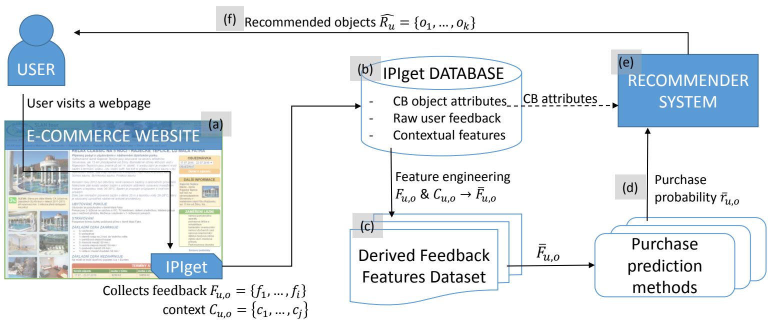

(f) Recommended objects $\widehat{R_u} = \{o_1, \ldots, o_k\}$
USER
User visits a webpage
(a)
E-COMMERCE WEBSITE
IPIget
Collects feedback $F_{u,o} = \{f_1, \ldots, f_i\}$
context $C_{u,o} = \{c_1, \ldots, c_j\}$
(b) IPIget DATABASE
- CB object attributes
- Raw user feedback
- Contextual features
CB attributes
(e) RECOMMENDER SYSTEM
Feature engineering
$F_{u,o}$ & $C_{u,o} \rightarrow \bar{F}_{u,o}$
(c) Derived Feedback Features Dataset
$\bar{F}_{u,o}$
(d) Purchase probability $\bar{r}_{u,o}$
Purchase prediction methods

Fig. 1. Outline of our approach on utilizing complex user feedback and presentation context in recommender systems. Implicit feedback features and presentation context are collected by the IPIget tool (a) and stored in a database (b). Feature engineering process results into the set of derived feedback features (c) used for purchase prediction. The resulting purchase probability (d) serves as an input of a recommender system (e), which provides recommendations to the user (f).

Table 1. Description of the raw user feedback features.

Feature	***Description***
f_1 View Count	The number of visits of the object
f_2 Dwell Time	Total time spent on the object
f_3 Mouse Distance	Approximate distance travelled by the mouse cursor
f_4 Mouse Time	Total time, the mouse cursor was in motion
f_5 Scroll Distance	Total scrolled distance
f_6 Scroll Time	Total time, the user spent by scrolling
r **Purchase**	Binary information whether user bought this object.

and Table 2. All features were collected with respect to the current user u and object o.

Let us now describe some features in more detail. Raw feedback features contain volumes of interaction generated by common user devices (mouse, keyboard etc.), or triggered by some GUI component. All the raw indicators have lower bounds equal to zero (i.e., no interaction was recorded) and except for purchases, they have no upper bound. We consider purchases as a golden standard for the user preference on e-commerce domains.

Comparison between page and browser dimensions is crucial to determine necessity of scrolling the page and also serves as natural rate of the scrolled distance. Number of images, links, text and page sizes serve as a proxy to the page complexity, which should affect the volumes of user actions needed to fully evaluate the page. For example the page with higher amount of text usually takes longer time to read.

Table 2. Description of the presentation context features.

Feature	***Description***
c_1 Number of links	Total number of links presented on the page
c_2 Number of images	Total number of images displayed on the page
c_3 Text size	Total length of the text presented on the page
c_4 Page dimensions	Width and height of the webpage
c_5 Browser window dimensions	Width and height of the browser window
c_6 Visible area ratio	Ratio between browser and page dimensions
c_7 Hand-held device	Binary indicator, whether a cellphone or tablet is used

Table 3. Description of the features introduced in the feature engineering step.

Feature	***Description***
f_i^u Relative User Feedback	The ratio between raw feedback and its per-user average value.
f_{sc} Scrolled area	The percentage of the page which have been presented in the browser visible area.
f_{hb} Hit bottom	Binary indicator whether the user scrolled up to the bottom of the page.
$f_{i,j}$ Feedback vs. Page complexity	The ratio between raw feedback (e.g., dwell time) and page complexity feature (e.g., number of links).

Derived feedback features were composed as follows. First, we defined relative per-user feedback features f_i^u to be able to distinguish specific users browsing patterns (2), where $avg_u(f_i)$ denotes average of feature f_i with respect to all records of user u.

$$f_i^u := f_i / avg_u(f_{i'} : u \text{ visited } i') \tag{2}$$

Next, we defined two features, *scrolled area* f_{sc} and *hit bottom page* f_{hb}, utilizing scrolling behavior and page dimensions. While the *hit bottom* is a simple indicator, whether the page was fully scrolled, the *scrolled area* represents the fraction of the page being visible for the user. Finally we aimed to relate volume of collected feedback with the page complexity context (*number of links*, *images* and *text*, *page dimensions* and *visible area ratio*). As there is no single measure of the page complexity, we opted for the Cartesian product of feedback and reciprocal page complexity features $f_{i,j}$ (3).

$$f_{i,j} := f_i / c_j; \text{ where } f_i \in \{f_1, ..., f_6, f_1^u, .., f_6^u\} \text{ and } c_j \in \{c_1, c_2, c_3, c_4, c_6\} \tag{3}$$

In our future work, we plan to investigate the experimental results with respect to the page complexity problem in order to deliver single page complexity metric. Table 3 lists all new features.

As our main aim is to evaluate contribution of presentation context to the recommendation quality, we defined and evaluated derived feedback datasets as follows:

- *Dwell Time* dataset follows the recommendation from [17] on using dwell time as a proxy towards user engagement. It contains only features f_2 and f_2^u.
- *Raw Feedback* dataset contains all f_i, f_i^u feedback features but no context.
- *Raw + Context* dataset contains f_i, f_i^u, c_i, f_{sc} and f_{hb} features, but not $f_{i,j}$.
- Finally, *all features* dataset contains all described features (f_i, f_i^u, c_i, f_{sc}, f_{hb} and $f_{i,j}$).

3.3 Predicting Purchase Probability from User Feedback

In order to predict *purchases* from derived feedback features, we have selected five machine learning techniques. For each technique, we used its R implementation from the caret package [3]. As the *purchase* indicator is a binary attribute, classification would be a natural option. However, our primary goal is not exactly predict purchased items. We need some better refined approximation for the user engagement as an input of the recommender system. Thus we need to focus either on classification methods class probabilities, or consider purchase prediction as a regression task. We will further refer to both purchase probability (classification methods) and expected value of dependent variable (regression methods) as purchase probability $\overline{r}_{u,o}$.

A potential advantage of regression techniques is the capability of providing negative preferences, i.e., infer user preference < 0, but learning regression function from binary training data could be highly biased. Based on the previous discussion, we decided to evaluate following classification and regression methods in this task.

Linear Regression (LinReg) is a simple regression method aiming to learn coefficients $\boldsymbol{A}$, b with the minimal square loss of the linear function $y = \boldsymbol{AX} + b$, where y is a dependent variable and $\boldsymbol{X}$ is a vector of independent variables (r and $\overline{F}$ in our case).

Lasso Regression (Lasso): The least absolute shrinkage and selection operator is a regression method that performs feature selection, which makes it capable to deal with higher dimensional datasets. The LASSOs objective is to find the parameter vector $\boldsymbol{A}$ that minimizes the sum of squared errors plus the regularization term $\lambda \|\boldsymbol{A}\|_1$, where λ is a hyperparameter controlling the regularization.

AdaBoost regression (Ada LinReg): Adaptive boosting is a meta-algorithm based on the principle of using weak learning algorithm iteratively over partially changed train sets. AdaBoost increases the weights of instances poorly predicted in previous iterations, thus although the individual learners are weak, the final model converge to a strong learner. In this case, linear regression was used.

Decision tree classification (J48): Specifically, the J48 implementation of the C4.5 algorithm was used. The C4.5 algorithm selects attributes on each node based on the normalized information gain. After the tree construction, it performs pruning, controlled by the hyperparameter c.

[3] http://topepo.github.io/caret/

AdaBoost classification (Ada Tree): The algorithm is in principle the same as Ada LinReg, except that the decision stump was used as a weak learner in this case.

3.4 Recommending based on Purchase Probability

The final step of our approach is to use purchase probability $\overline{r}_{u,o}$ in recommender systems. Our previous work [11] shown that purely collaborative algorithms are not suitable for small e-commerce enterprises, so we decided to evaluate one content-based and one hybrid recommending algorithm.

Vector Space Model (VSM) is well-known content-based algorithm brought from information retrieval. We use the variant described in [6] with binarized content-based attributes serving as document vector, TF-IDF weighting and cosine similarity as objects similarity measure. The algorithm recommends top-k objects most similar to the user profile.

For the purpose of content-based recommendation, the dataset of objects (travel agency tours) attributes was used. The dataset contains approximately 20 attributes, such as type of the tour, accommodation quality, destination countries and regions, price per night, discount etc. For more information, please refer to [11].

Popular from similar categories recommender (Popular SimCat). Popular SimCat, is a simple hybrid approach based on collaborative similarity of product categories. There are two motivations for this algorithm.

First, in our early experiments on a Travel Agency website [8], recommending objects from currently visited category turns out to be quite a good baseline. However, some categories were very narrow, containing only a handful of objects, sometimes even less than the intended size of the recommended objects list. For such a narrow category, it might be useful to also recommend objects from categories similar to the current one. Furthermore, there are substantially fewer categories than objects in the dataset (and the list of categories is much more stable), so it is possible to use collaborative similarity of categories.

Second, one of the most successful non-personalized recommendation approach is simply recommending the most popular objects.

Putting both motivations together, the algorithm in training phase computes categories similarity and objects popularity: Categories similarity is defined as Jaccard similarity, based on the users covisiting both categories (4), where U_{c1} and U_{c2} are sets of users who visited category c_1 and c_2 respectively.

$$Sim(c_1, c_2) := \frac{|U_{c1} \cap U_{c2}|}{|U_{c1} \cup U_{c2}|} \tag{4}$$

The objects popularity is defined as the logarithm of the number of objects visits in the train set (5).

$$Pop(o_i) := log\left(\sum_{\forall \text{ users}} ViewCount(o_i)\right) \tag{5}$$

In the prediction phase, the algorithm collects all visited and similar categories for the current user and orders the objects according to the $Pop(o_i) * Sim(c_{[o_i]})$ scoring function. More details can be found in [11].

4 Evaluation and Results

4.1 Evaluation Protocol

In this section, we would like to provide details of the evaluation procedure. In total four datasets of user feedback, five purchase prediction methods and two recommending algorithms were evaluated. Before we describe the protocol itself, let us mention some facts about the datasets used in the experiments.

The dataset of user feedback (including contextual features) was collected by observing real visitors of a mid-sized Czech travel agency. The dataset was collected by the IPIget tool during the period of more than one year, contains over 560K records and is available for research purposes[4]. For the purpose of the evaluation, we restricted the dataset only to the users, who visited at least 3 objects and purchased at least one of them. The resulting datasets contained 516 distinct users, 666 purchases, 1533 objects and over 23000 records, in average 45 records per user. However, please note that the number of records per user approximately follows the power-law distribution.

The evaluation of the proposed methods was carried out in two steps.

In the first step, ***purchase prediction***, the task was to identify, which objects visited by the current user were purchased. Even though it looks like a binary classification, it is not exactly true, as we want a finer grained ordering as an input of the recommender system and we do not insist on proper classification of unpurchased items. We evaluate the problem as a ranking task, where ordering is induced by the purchase probability $\overline{r}_{u,o}$. Objects actually purchased by the user should appear on top of the list.

The evaluation was performed according to the leave-one-out cross-validation protocol applied on the user set. Machine learning algorithms were trained on the feedback data from all users, except for the current one, and afterwards predict for each object o visited by the current user u its purchase probability $\overline{r}_{u,o}$. The ordering induced by $\overline{r}_{u,o}$ was evaluated in terms of normalized discounted cumulative gain (nDCG), recall of purchased objects in top-k items (recall@top-k) and its average ranking position.

This scenario simulates a well-known new user problem. When a new user enters the system, more complicated machine learning models cannot be retrained in real-time, taking into account feedback of the current user, so we need to infer his/her preferences from other users data. Using real-time local models, i.e., train only from the feedback of the current user, is impractical as there is usually not enough (if any) positive feedback.

The second step, ***recommendation experiment***, evaluates quality of the list of recommended objects in terms of position of the actually purchased ones.

[4] See http://bit.ly/2dsjg6j

Table 4. Results of the purchase prediction methods in terms of nDCG for different implicit feedback datasets. The best results are in bold.

Method	***DwellTime***	***Raw feedback***	***Raw + Context***	***All feedback***
LinReg	0.725	0.714	0.834	0.828
Lasso	0.730	0.719	0.831	0.827
Ada LinReg	0.713	0.713	0.863	0.864
J48	0.738	0.740	0.891	0.893
Ada Tree	0.757	0.763	**0.950**	**0.950**

The evaluation of this step was also performed according to the leave-one-out cross-validation, however applied on the set of purchased objects. For each pair of the purchased object o and the user u who bought it, we trained recommender systems based on all other available data and ask it to recommend top-k best objects for the current user $\hat{R}_u : \{o_1, ..., o_k\}$. Again, we consider the task as ranking, so the actually purchased object should appear on top of the list. Results were evaluated in terms of nDCG and recall@top-k metrics.

4.2 Results: Purchase Prediction

Table 4 depicts overall results of the purchase prediction experiment. The results of nDCG are surprisingly high, especially in case of *Ada Tree* prediction method, however please note that the R implementation of nDCG metric[5] compensates for ties in the ranking. The results of other evaluation metrics (recall@top-k, average position) were very similar, so we omit them for the sake of space. Both classification methods clearly outperform all regression methods. Adding contextual features substantially improved prediction capability of all methods, but adding page complexity based features did not improve the results of all methods except for *J48*. *Ada Tree* classifier performed the best across all datasets.

4.3 Results: Recommendation Experiment

Table 5 depicts the overall results of the recommendation experiment. Additionally, to the purchase probability inputs we also evaluated *Binary* baseline method, which simply considers all visited objects as relevant[6]. For the sake of space, we do not display detailed the results of recall@top-k metric, however it mostly corresponds with nDCG. The best performing method in terms of recall@top-5 and recall@top-10 was *J48* with *Raw+Context* dataset and *Popular SimCat* recommender, achieving recall of 0.297 and 0.376 for top-5 and top-10 respectively.

[5] StatRank package, https://cran.r-project.org/web/packages/StatRank

[6] We did not evaluate the input based solely on purchases, because over 90% of users purchased only one item and recommending algorithms could not predict anything for them.

Table 5. Results of the recommendation experiment in terms of average nDCG for different implicit feedback datasets and recommending algorithms. Baseline methods are depicted in grey italics, the best results are in bold.

Method	***Recommender***	***DwellTime***	***Raw feedback***	***Raw + Context***	***All feedback***
Binary	*VSM*			*0.304*	
LinReg	VSM	0.299	0.297	0.215	0.215
Lasso	VSM	0.304	0.298	0.213	0.216
Ada LinReg	VSM	0.302	0.301	0.215	0.215
J48	VSM	0.299	0.295	0.303	**0.311**
Ada Tree	VSM	0.293	0.298	0.294	0.296
Binary	*Popular SimCat*			*0.362*	
LinReg	Popular SimCat	0.342	0.342	0.267	0.270
Lasso	Popular SimCat	0.359	0.343	0.260	0.270
Ada LinReg	Popular SimCat	0.361	0.360	0.264	0.264
J48	Popular SimCat	0.353	0.354	**0.373**	0.372
Ada Tree	Popular SimCat	0.358	0.358	0.363	0.370

As can be seen from the results, all regression based methods performed worse than the baseline and furthermore its performance gradually decreased for enriched datasets in the most cases. This might be a problem of learning regression from only binary input, but as all regression methods were based on a linear model, we do not want to conclude on this subject yet. On the other hand, *Popular SimCat* with both *Ada Tree* and *J48*, as well as *VSM* with *J48* outperformed baselines. Furthermore as can be seen in Table 6, there is a significant performance improvement between *raw feedback* and datasets containing contextual features for those methods. It seems that using page complexity based features $f_{i,j}$ can also improve performance of some methods, however the results are less clear at this point.

Surprisingly, the relatively simple *Popular SimCat* algorithm produced consistently better results than *VSM*. This is in contradiction with our previous experiments with these algorithms [11], however we need to note that the target of the previous experiment was to predict visited instead of purchased objects. We would like to investigate this topic more in our future work.

5 Conclusions and Future Work

In this paper, our aim was to show that user feedback should be considered with respect to the context of the page and device. We defined several features describing such context and incorporate them into the user feedback feature space. In the purchase prediction task, the usage of context clearly improved performance of all learning methods in predicting purchased objects. Furthermore, by using purchase probability as a proxy towards user engagement, we were able to improve quality of the recommendations over both binary feedback baseline and uncontextualized feedback in terms of nDCG and recall@top-k.

Table 6. P-values of the binomial significance test [15] for selected combination of algorithms. The test was performed with respect to the recall of purchased object in top-K.

Recommender	***Baseline***	***Method***	***p-value*** recall@5	***p-value*** recall@10
VSM	*Binary*	J48 (All feedback)	0.028	0.026
	J48 (Dwell time)	J48 (All feedback)	0.024	0.001
Popular SimCat	*Binary*	J48 (All feedback)	0.025	0.198
	Binary	J48 (Raw + Context)	0.036	0.015
	Binary	Ada Tree (All feedback)	0.009	0.154
	J48 (Raw feedback)	J48 (All feedback)	0.001	0.004
	Ada Tree (Raw feedback)	Ada Tree (All feedback)	0.057	0.000

In this paper we did not investigate the influence of each contextual feature separately as well as possibility to combine purchase probabilities coming from different learning methods. Both should be done in our future work. The presented approach can be applied on any domain, as long as there is some natural indicator of user engagement or preference (like purchases in e-commerce). Thus, naturally, one possible direction of our research is to extend this approach beyond its current e-commerce application.

Another task is to combine the contextual approach with our previous work, e.g., on using early user feedback on lists of objects [11] and corroborate the results in on-line experiments.

Acknowledgments. The experiments presented in this paper were done while author was a Ph.D. student at Charles University in Prague. The work was supported by the Czech grant P46. Supplementary materials (datasets, source codes, results) can be found on: http://bit.ly/2dsjg6j.

References

1. Mark Claypool, Phong Le, Makoto Wased & David Brown (2001): *Implicit interest indicators.* In: *Proceedings of the 6th international conference on Intelligent user interfaces*, IUI '01, ACM, New York, NY, USA, pp. 33–40, doi:10.1145/359784.359836.
2. A. Eckhardt, T. Horvath & P. Vojtas (2007): *PHASES: A User Profile Learning Approach for Web Search.* In: *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, WI '07, IEEE Computer Society, Washington, DC, USA, pp. 780–783, doi:10.1109/WI.2007.146.
3. Yi Fang & Luo Si (2012): *A Latent Pairwise Preference Learning Approach for Recommendation from Implicit Feedback.* In: *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, ACM, New York, NY, USA, pp. 2567–2570, doi:10.1145/2396761.2398693.
4. Yifan Hu, Yehuda Koren & Chris Volinsky (2008): *Collaborative Filtering for Implicit Feedback Datasets.* In: *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, ICDM '08, IEEE Computer Society, Washington, DC, USA, pp. 263–272, doi:10.1109/ICDM.2008.22.

5. Y. Lai, X. Xu, Z. Yang & Z. Liu (2012): *User Interest Prediction Based on Behaviors Analysis.* *International Journal of Digital Content Technology and its Applications* 6(13), pp. 192–204, doi:10.4156/jdcta.vol6.issue13.22.
6. Pasquale Lops, Marco de Gemmis & Giovanni Semeraro (2011): *Content-based Recommender Systems: State of the Art and Trends.* In Francesco Ricci, Lior Rokach, Bracha Shapira & Paul B. Kantor, editors: *Recommender Systems Handbook*, Springer US, pp. 73–105, doi:10.1007/978-0-387-85820-3_3.
7. Ladislav Peska (2014): *IPIget: The Component for Collecting Implicit User Preference Indicators.* In: *ITAT 2014: Information Technologies - Applications and Theory*, ITAT '14, Ustav informatiky AV CR, pp. 22–26. Available at http://itat.ics.upjs.sk/workshops.pdf.
8. Ladislav Peska, Alan Eckhardt & Peter Vojtas (2011): *UPComp - A PHP Component for Recommendation Based on User Behaviour.* In: *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Volume 03*, WI-IAT '11, IEEE Computer Society, Washington, DC, USA, pp. 306–309, doi:10.1109/WI-IAT.2011.180.
9. Ladislav Peska, Alan Eckhardt & Peter Vojtas (2015): *Preferential Interpretation of Fuzzy Sets in E-shop Recommendation with Real Data Experiments.* *Archives for the Philosophy and History of Soft Computing* 2. Available at http://aphsc.org/index.php/aphsc/article/view/32/23.
10. Ladislav Peska & Peter Vojtas (2015): *How to Interpret Implicit User Feedback?* In: *Poster Proceedings of the 9th ACM Conference on Recommender Systems (RecSys 2015)*, 1441, CEUR-WS. Available at http://ceur-ws.org/Vol-1441/recsys2015_poster8.pdf.
11. Ladislav Peska & Peter Vojtas (2016): *Using Implicit Preference Relations to Improve Recommender Systems.* *Journal on Data Semantics*, pp. 1–16, doi:10.1007/s13740-016-0061-8.
12. Filip Radlinski & Thorsten Joachims (2005): *Query chains: learning to rank from implicit feedback.* In: *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, KDD '05, ACM, New York, NY, USA, pp. 239–248, doi:10.1145/1081870.1081899.
13. Karthik Raman, Pannaga Shivaswamy & Thorsten Joachims (2012): *Online Learning to Diversify from Implicit Feedback.* In: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, ACM, New York, NY, USA, pp. 705–713, doi:10.1145/2339530.2339642.
14. Steffen Rendle, Christoph Freudenthaler, Zeno Gantner & Lars Schmidt-Thieme (2009): *BPR: Bayesian Personalized Ranking from Implicit Feedback.* In: *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI '09, AUAI Press, Arlington, Virginia, United States, pp. 452–461. Available at http://dl.acm.org/citation.cfm?id=1795114.1795167.
15. Steven L. Salzberg (1997): *On Comparing Classifiers: Pitfalls to Avoid and a Recommended Approach.* *Data Mining and Knowledge Discovery* 1(3), pp. 317–328, doi:10.1023/A:1009752403260.
16. Byoungju Yang, Sangkeun Lee, Sungchan Park & Sang-goo Lee (2012): *Exploiting Various Implicit Feedback for Collaborative Filtering.* In: *Proceedings of the 21st International Conference Companion on World Wide Web*, WWW '12 Companion, ACM, New York, NY, USA, pp. 639–640, doi:10.1145/2187980.2188166.
17. Xing Yi, Liangjie Hong, Erheng Zhong, Nanthan Nan Liu & Suju Rajan (2014): *Beyond Clicks: Dwell Time for Personalization.* In: *Proceedings of the 8th ACM Conference on Recommender Systems*, RecSys '14, ACM, New York, NY, USA, pp. 113–120, doi:10.1145/2645710.2645724.

Part II

Presentation Abstracts

SMT Query Decomposition and Caching in Data-Symbolic Model Checking

Jan Mrázek and Jiří Barnat

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{xmrazek7,barnat}@fi.muni.cz

Satisfiability Modulo Theory (SMT) solvers have been recently used to build or improve a number of tools for formal verification of C and C++ programs. For these tools, most of verification time is typically spent on internal decisions on satisfiability of relevant first-order formulae over bit-vectors, which inevitably leads the tool developers to apply appropriate countermeasures, such as caching of SMT queries. While caching is widely used in symbolic execution, where the underlying SMT formulae are quantifier-free by construction, in control-explicit data-symbolic model checking approach the SMT formulae to be checked for satisfiability contain a number of universally bounded variables. In such a case, the caching principle is rendered rather inefficient. In this paper, we introduce a new scheme for decomposition of SMT formulae with universally bounded variables into a data-independent sub-formula, and show that application of caching to these sub-formulae restores some efficiency of the caching mechanism. We illustrate efficiency of the new caching scheme on a set of examples from the Software Verification Competition (SV-COMP) benchmark using our own prototype implementation in the SymDIVINE model checker.

Index appearance record for transforming Rabin automata into parity automata

Jan Křetínský, Tobias Meggendorfer, Clara Waldmann, and Maximilian Weininger

Technical University of Munich

Constructing correct-by-design systems from specifications given in linear temporal logic (LTL) is a classical problem called *LTL synthesis*. The automata-theoretic solution to this problem is to translate the LTL formula to a deterministic automaton and solve the corresponding game on this automaton. Often, parity automata (DPA) are used due to the efficiency of parity game solvers.

The classical way to transform LTL formulae into DPA is to first create a non-deterministic Büchi automaton and then determinize it. Since determinization procedures based on Safra's construction are practically inefficient, many alternative approaches to LTL synthesis arose, trying to avoid determinization. However, new results on translating LTL directly and efficiently into deterministic Rabin automata (DRA) [2] open new possibilities for the automata-theoretic approach. Consequently, we aim to efficiently transform DRA into DPA. Transformations of deterministic automata into DPA are mostly based on *appearance records* and have been presented for Muller automata [4] and Streett automata [3], implicitly yielding a procedure for DRA. Our contribution in this paper is as follows:

- We provide an IAR construction transforming DRA to DPA, different from [3] and specifically tailored to DRA.
- We provide optimizations applicable to all appearance records.
- We evaluate all the unoptimized and optimized versions of our IAR and the IAR of [3] experimentally, in comparison to determinization implemented in GOAL. Surprisingly, our method produces smaller automata than both GOAL and the dual IAR of [3].
- We compare our approach $\text{LTL}\xrightarrow{\text{Rabinizer}}\text{DRA}\xrightarrow{\text{optimized IAR}}\text{DPA}$ to the state-of-the-art translation of LTL to DPA by Spot 2.1 [1].

References

1. Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *ATVA*, 2016. To appear.
2. Zuzana Komárková and Jan Křetínský. Rabinizer 3: Safraless translation of LTL to small deterministic automata. In *ATVA*, pages 235–241, 2014.
3. Christoph Löding. Methods for the transformation of automata: Complexity and connection to second order logic. Master's thesis, Institute of Computer Science and Applied Mathematics, Christian-Albrechts-University of Kiel, Germany, 1999.
4. Stefan Schwoon. Determinization and complementation of streett automata. In *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, pages 79–91, 2001.

Automorphisms of the Cube n^d

Pavel Dvořák[1] and Tomáš Valla[2] *

[1] Faculty of Mathematics and Physics,
Charles University, Prague, Czech Republic
`koblich@iuuk.mff.cuni.cz`
[2] Faculty of Information Technology,
Czech Technical University in Prague, Czech Republic
`tomas.valla@fit.cvut.cz`

Consider a hypergraph $H_n^d = (V, E)$ where $V = [n]^d$ and E contains all subsets of V which lie in a geometric line in the d-dimensional space where the set $[n]^d$ is embedded. We characterize the structure of the group T_n^d of automorphisms of H_n^d. This is a generalization of Silver [?] who characterized T_4^3. Moreover, we consider the Colored Cube Isomorphism problem of deciding whether for two colorings of the vertices of H_n^d there exists an automorphism of H_n^d preserving the colors.

The hypergraph H_n^d is a board for d-dimensional version of the game Tic-tac-toe. Any position in Tic-tac-toe can be viewed as a coloring of the hypercube by three colors: crosses, rings and empty points. Therefore, when designing a strategy by searching the game state space it is very useful to recognize isomorphic positions. To do so we provide the characterization of all automorphisms of the cube. On the other hand, we show how computationally hard is to recognize if there is an automorphism which preserves the cube coloring.

We use two basic groups of automorphisms for characterization of the group $\mathbb{T}_n^d$. The first group $\mathbb{G}_d$ is the group of the d-dimensional hypercube. The second group $\mathbb{F}_n$ contains mappings $F_\pi([x_1, \ldots, x_d]) = [\pi(x_1), \ldots, \pi(x_d)]$ for all $\pi \in \mathbb{S}_n$ such that $\pi(n - p + 1) = n - \pi(p) + 1$. Note that the group structure of $\mathbb{T}_n^d$ is richer than only the obvious rotations and symmetries.

Theorem 1. *Let $n > 2$. The group $\mathbb{T}_n^d$ is generated by the elements of $\mathbb{R}_d \cup \mathbb{F}_n$. The order of the group $\mathbb{T}_n^d$ is $2^{d-1+k} d!k!$ where $k = \lfloor \frac{n}{2} \rfloor$.*

The class of decisions problems GI contains all problems with a polynomial reduction to the problem Graph Isomorphism.

Theorem 2. *The problem* Colored Cube Isomorphism *is* GI*-complete.*

The conference paper was accepted to Cocoon 2016 [?].

* Supported by the Centre of Excellence – Inst. for Theor. Comp. Sci. 79 (project P202/12/G061 of GA ČR).

Finding Boundary Elements in Ordered Sets with Application to Requirements Analysis[⋆]
(Originally Presented at SEFM 2016)

Jaroslav Bendík, Nikola Beneš, Jiří Barnat, and Ivana Černá

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{xbendik,xbenes3,barnat,cerna}@fi.muni.cz

The motivation of this work comes from various source areas, such as parametric formal verification, requirements engineering, safety analysis, or software product lines. In these areas, the following situation often arises: We are given, as an input, a set of configurations and a property of interest. The goal is to compute the set of all the configurations that satisfy the given property. We call such configurations valid. As a short example, one may imagine a system with tunable parameters that is to be verified for correctness. The set of configurations, in that case, is a set of all possible parameter values and the goal is to find all such values that ensure the correctness of the given system. If we are given a method to ascertain the validity of a single configuration, we could try running the method repeatedly for each configuration to obtain the desired result. In the case of an infinite set of configurations, this approach does not terminate, and we get at most a partial answer. However, even if the configuration space is finite, checking configurations one by one may be too costly. We are thus interested in reducing the number of validity checks in the finite case.

Although such reduction might be impossible in general, we focus on problems whose configuration space is equipped with a certain structure that is preserved by the property of interest. This may then be exploited in order to check a smaller number of configurations and still obtain the full answer. The desired structure is a set of dependencies of the form: "If configuration A violates the property then configuration B does too." Mathematically, we can either view such structure as a directed acyclic graph of those dependencies, or as a partial ordering on the set of all configurations induced by this graph. Viewed as an ordered set, the set of all the valid configurations can be effectively represented by the set of all the maximal valid (alternatively, minimal invalid) configurations.

We present a novel algorithm to compute such boundary elements and we explain how this general setting applies to concistency checking of requirements. We also give an experimental comparison with a state-of-the-art tool.

⋆ This project has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 692474, project name AMASS. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme and Spain, Czech Republic, Germany, Sweden, Austria, Italy, United Kingdom, France

A Model Checking Approach to Dynamical Systems Analysis (PRESENTATION)*

Nikola Beneš, Luboš Brim, Matej Hajnal, Martin Demko, Samuel Pastva and David Šafránek

Systems Biology Laboratory, Faculty of Informatics, Masaryk University
Botanická 68a, 602 00 Brno, Czech Republic,
{xbenes3,brim,xhajnal,xdemko,xpastva,xsafran1}@fi.muni.cz

Techniques developed in computer science for automated formal verification are promising formal methods that have the potential to be exploited in computational systems biology where biological processes are studied as dynamical systems. Of special interest is model checking that provides a feasible methodology to verify/refute interesting biological hypotheses. We have contributed to this field by adapting model checking to be applicable to parametrised dynamical systems appearing in biology [1, 2].

In our recent research, we have proposed a novel high-performance algorithm for synthesis of interdependent parameters from CTL specifications for non-linear dynamical systems based on coloured model checking. The method employs a symbolic representation of sets of parameter valuations in terms of first-order theory of the reals [4].

Moreover, we employed this approach for bifurcation analysis of parameterised high-dimensional dynamical systems. The classical numerical and analytical methods are typically limited to a small number of system parameters. Therefore, in [3] we have proposed an important improvement in the use of an extended CTL logic, a kind of a hybrid CTL augmented with direction formulae, in order to describe various behaviour patterns, or phase portraits. In combination with interdependent parameters synthesis algorithm this approach is able to overpass previously mentioned limitation and to discover critical values of such parameters, or bifurcation points, where phase portraits appear or disappear.

We applied these methods to a class of piecewise multi-affine dynamical systems representing dynamics of biological systems with complex non-linear behaviour involving bistability [5] or limit cycles [6].

In this presentation, we would like to give an overview of our recent results.

References

1. Barnat, J., Brim, L., Krejci, A., Streck, A., Safranek, D., Vejnar, M., Vejpustek, T.: On Parameter Synthesis by Parallel Model Checking. IEEE/ACM Transactions on Computational Biology and Bioinformatics 9(3), 693–705 (2012)
2. Barnat, J., Brim, L., Safránek, D.: High-performance analysis of biological systems dynamics with the divine model checker. Brief. Bioinform 11(3), 301–312 (2010)
3. Beneš, N., Brim, L., Demko, M., Pastva, S., Šafránek, D.: A model checking approach to discrete bifurcation analysis. In: International Symposium on Formal Methods. Springer (2016), to appear.
4. Beneš, N., Brim, L., Demko, M., Pastva, S., Šafránek, D.: Parallel smt-based parameter synthesis with application to piecewise multi-affine systems. In: International Symposium on Automated Technology for Verification and Analysis. Springer (2016), to appear.
5. Demko, M., Beneš, N., Brim, L., Pastva, S., Šafránek, D.: High-performance symbolic parameter synthesis of biological models: A case study. In: International Conference on Computational Methods in Systems Biology. Springer (2016), to appear.
6. Hajnal, M., Šafránek, D., Demko, M., Pastva, S., Krejčí, P., Brim, L.: Toward modelling and analysis of transient and sustained behaviour of signalling pathways. In: International Workshop on Hybrid Systems Biology. Springer (2016), to appear.

* This work has been supported by the Czech Science Foundation grant GA15-11089S.

Parameterized complexity of length-bounded cuts and multicuts [⋆] [⋆⋆]

Pavel Dvořák[1] and Dušan Knop[2]

[1] Computer Science Institute, Charles University
Prague, Czech Republic
koblich@iuuk.mff.cuni.cz

[2] Department of Applied Mathematics, Charles University
Prague, Czech Republic
knop@kam.mff.cuni.cz

We show that the Minimum Length-Bounded Cut problem can be computed in linear time with respect to L and the tree-width of the input graph as parameters.

We show a W[1]-hardness result when the parameterization is done by the path-width only (instead of the tree-width) and that this problem does not admit polynomial kernel when parameterized by path-width and L. We also derive an FPT algorithm for the Minimum Length-Bounded Cut problem when parameterized by the tree-depth. Thus showing an interesting paradigm for this problem and parameters tree-depth and path-width.

Length bounded cuts. Let $s, t \in V$ be two distinct vertices of a graph $G = (V, E)$ – we call them the source and the sink, respectively. We call a subset of edges $F \subseteq E$ of G an *L-bounded cut* (or *L-cut* for short), if the length of the shortest path between s and t in the graph $(V, E \setminus F)$ is at least $L + 1$. We measure the length of the path by the number of its edges. In particular, we do not require s and t to be in distinct connected components as in the standard cut, instead we do not allow s and t to be close to each other. We call the set F *a minimum L-cut* if it has the minimum size among all L-bounded cuts of the graph G. The associated decision problem is the Minimum Length Bounded Cut problem. Throughout the paper we denote by n the number of vertices of input graph G.

Theorem 1. Minimal Length Bounded Cut *parameterized by path-width is* W[1]*-hard.*

Theorem 2. *Let G be a graph, denote by k treedepth of G, and s and t be two distinct vertices of G. Now for any $L \in \mathbb{N}$ a minimum L-cut between s and t can be found in time $O\big(\max\{2^{6k^3} \cdot n, nm\}\big)$.*

The only other result of this type we are aware of, in the time of writing this, is by Gutin et al. [2]. The Minimum Length-Bounded Cut problem is also a problem of this kind – as Theorem 1 and Theorem 2 demonstrate.

[⋆] Research was supported by the project SVV–2016–260332, by project Kontakt LH12095, project GAUK 1784214, and project CE-ITI P202/12/G061.

[⋆⋆] Results were presented during TAMC 2015 [1].

References

1. Pavel Dvořák & Dušan Knop (2015): *Parametrized Complexity of Length-Bounded Cuts and Multi-cuts.* In: *Theory and Applications of Models of Computation - 12th Annual Conference, TAMC 2015, Singapore, May 18-20, 2015, Proceedings*, pp. 441–452, doi:10.1007/978-3-319-17142-5_37. Available at `http://dx.doi.org/10.1007/978-3-319-17142-5_37`.
2. Gregory Gutin, Mark Jones & Magnus Wahlström (2015): *Structural Parameterizations of the Mixed Chinese Postman Problem.* In Nikhil Bansal & Irene Finocchi, editors: *Algorithms – ESA 2015*, *Lecture Notes in Computer Science* 9294, Springer Berlin Heidelberg, pp. 668–679, doi:10.1007/978-3-662-48350-3_56. Available at `http://dx.doi.org/10.1007/978-3-662-48350-3_56`.

Solving Quantified Bit-Vector Formulas Using Binary Decision Diagrams

Martin Jonáš

Faculty of Informatics, Masaryk University, Brno, Czech Republic
martin.jonas@mail.muni.cz

We described a new approach to deciding satisfiability of quantified bit-vector formulas using binary decision diagrams and approximations. The approach is motivated by the observation that the binary decision diagram for a quantified formula is typically significantly smaller than the diagram for the subformula within the quantifier scope. The approach relies on the combination of syntactic formula simplifications, tailored initial ordering of BDD variables, and using underapproximations and overapproximations, all of which help to reduce the size of the BDDs during the computation of the BDD corresponding to the input formula.

The suggested approach has been implemented in the experimental open-source SMT solver called Q3B. The experimental results show that it decides more benchmarks from the SMT-LIB repository than state-of-the-art SMT solvers for this theory, namely Z3 and CVC4, which solve quantified bit-vector formulas using the quantifier instantiation. Furthermore, we compared the implemented solver with the mentioned state-of-the art solvers on the quantified formulas produced by the symbolic model checker SymDIVINE. Again, the BDD based solver was able to decide more queries than Z3 and CVC4. Moreover, according to the results of the SMT competition 2016, Q3B is the best solver in the category of quantified bit-vectors, outperforming Z3, CVC4 and Boolector.

Underapproximations used in the approach are performed by reducing the bit-width of all existentially quantified variables. Similarly, overapproximations are performed by reducing the bit-width of all universally quantified variables. We describe several ways of reducing bit-width of a variable and we evaluate the effect of each of these on the performance of the solver.

The publication with the detailed description of the approach, experimental results, and the evaluation of all three components used in the approach was accepted to the *19th International Conference on Theory and Applications of Satisfiability Testing (SAT 2016).*

Author Index

Organisers

Institute of Computer Science
Masaryk University
Botanická 68a
Brno, Czech Republic
`http://www.ics.muni.cz`

Faculty of Informatics
Masaryk University
Botanická 68a
Brno, Czech Republic
`http://www.fi.muni.cz`

Faculty of Information Technology
Brno University of Technology
Božetěchova 2
Brno, Czech Republic
`http://www.fit.vutbr.cz`

Best Paper Sponsors

Lexical 文 Computing

Jan Bouda, Lukáš Holík, Jan Kofroň, Jan Strejček,
and Adam Rambousek (Eds.)

MEMICS 2016
11th Doctoral Workshop on Mathematical
and Engineering Methods in Computer Science

Published by Masaryk University, Žerotínovo nám. 617/9, 601 77 Brno
First electronic edition, 2016

Typesetting: Adam Rambousek from camera ready by paper authors and PC members

Cover design: Tomáš Staudek

ISBN 978-80-210-8362-2

muni
PRESS