Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science

MEMICS 2009, November 13–15, 2009, Znojmo, Czech Republic

Edited by

Petr Hlinený
Václav Matyáš
Tomáš Vojnar

OASICS

OASIcs – Vol. 13 – MEMICS 2009 www.dagstuhl.de/oasics

Editors

Petr Hlinený
Faculty of Informatics MU
Masaryk University
Botanicka 68a
602 00 Brno, Czech Republic
hlineny@fi.muni.cz

Václav Matyáš
Faculty of Informatics
Masaryk University
Botanicka 68a
602 00 Brno, Czech Republic
Matyas@fi.muni.cz

Tomáš Vojnar
Faculty of Information Technology
Brno University of Technology
Božetěchova 2
612 66 Brno, Czech Republic
vojnar@fit.vutbr.cz

ACM Classification 1998
G. Mathematics of Computing, J.2 Physical Sciences and Engineering

ISBN 978-3-939897-15-6

Published online and open access by
Schloss Dagstuhl – Leibniz-Center for Informatics GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany.

Publication date
December, 2009.

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/OASIcs.MEMICS.2009.2342

ISBN 978-3-939897-15-6 **ISSN 2190-6807** **http://www.dagstuhl.de/oasics**

ISSN 2190-6807

www.dagstuhl.de/oasics

Preface

The 2009 edition of the International Doctoral Workshop on *Mathematical and Engineering Methods in Computer Science* (MEMICS'09) is the fifth in a row of three-day workshops organized in South Moravia by Faculty of Information Technology of Brno University of Technology and Faculty of Informatics of Masaryk University.

MEMICS'09 was held in Znojmo, Czech Republic, on November 13–15, 2009.

The MEMICS workshops are intended to provide an opportunity for PhD students to present and discuss their work in an international environment. An important feature of MEMICS is its cross-discipline orientation allowing for an exchange of ideas among several different fields of computer science and engineering. The workshop series is run by the two aforementioned schools within the project *Mathematical and Engineering Approaches to Developing Reliable and Secure Concurrent and Distributed Computer Systems*, financially supported by the Czech Science Foundation.

The workshop topics include the following (though not exclusive) areas: software and hardware dependability, computer security, parallel and distributed computing, formal analysis and verification, simulation, testing and diagnostics, GRID computing, computer networks, modern hardware and its design, non-traditional computing architectures, quantum computing, and all related areas of computer science.

While mainly focusing on works primarily authored by PhD students, the MEMICS workshops also include invited lectures given by internationally recognized researchers. In particular, the MEMICS'09 workshop has enjoyed four invited lectures, covering different fields of interest of MEMICS participants and providing both theoretical and more applied results. The invited speakers this year have been Mikolaj Bojanczyk from University of Warsaw in Poland with a talk *Automata for XML*, Rūsiņš Freivalds from University of Latvia in Riga talking about *Randomization and Other Ways to Overcome Determinism in Algorithms*, Peter Habermehl from University Paris 7 in France presenting *Angluin-style Learning of Non-deterministic Finite-state Automata*, and Günther Raidl from Vienna University of Technology in Austria with a talk on *Combining Metaheuristics with Mathematical Programming Techniques for Solving Difficult Network Design Problems*. Thanks go to all these invited lecturers who found time to participate at the workshop and demonstrate how a top quality presentation could look like.

MEMICS workshops invite PhD students to come and present their research results in front of their peers and to receive immediate advice and feedback from participating senior faculty members, including the invited lecturers. Participating students are encouraged to express their opinions, to discuss the presentations, and to exchange ideas, compare methods, traditions, and approaches of groups and institutions whose representatives are participating at the workshop. These discussions and social networking are expected to contribute to the further research collaboration among participants and their institutions. All the time,

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2342

various issues of achieving reliability and security of computer systems (with a special, though not exclusive emphasis on concurrent and distributed systems) lie in the foundations, making sure the participants have a common ground on which to build such a collaboration. This focus corresponds tightly to the need to build as reliable and secure computer systems as possible, which is increasingly stressed worldwide.

The cornerstone of any scientific workshop is its programme. This year, the scientific programme has contained 26 contributed papers selected by the Programme Committee from 45 submissions. Each submitted paper was evaluated by at least three reviewers who provided an extensive feedback to the authors. This hard and time consuming task is highly appreciated as without it, MEMICS could never be a successful event. Apart from regular papers, MEMICS also invites PhD students to present work that has already been peer reviewed and presented on some well known international conference. This workshop programme has included 21 presentations chosen out of 27 submissions. The full programme listing can be found on `http://www.memics.cz/2009`. Altogether, the regular papers and presentations have come from authors from 15 different countries (10 for the regular papers).

After another round of evaluation and negotiation with the authors, five best regular papers of MEMICS'09 have been invited to a special issue of the Computing and Informatics journal, and 14 other high quality contributions are presented in this DROPS volume, making it a really strong issue covering diverse parts of computer science from pure theory to practical applications. We thank all of them for their effort put into the preparation of their papers.

Lastly to say, the organization aspects of the workshop have been taken care by the local Organizing Committee which has worked hard to guarantee a smooth realization of MEMICS'09 and has contributed significantly to the success of the workshop. We would like to explicitly thank to the Organizing Committee members and its chair, Jan Staudek, for all their efforts. More than 100 participants have registered for the workshop, with around one fourth from abroad, demonstrating the interest in this event.

The MEMICS'09 workshop was financially supported by the Doctoral Grant 102/09/H042 from the Czech Science Foundation. Our thanks go also to the companies CEPIA Technologies, Y Soft, and Red Hat Czech for providing funds for the *best paper awards* of MEMICS'09, given at the workshop. Together with other direct and indirect support and help from the organizing faculties these contributions are highly appreciated.

Brno, December 2009 Tomáš Vojnar, Petr Hliněný, and Vashek Matyáš
PC chairs of MEMICS'09

A Privacy-Aware Protocol for Sociometric Questionnaires[⋆]

Marián Novotný

Institute of Computer Science, Pavol Jozef Šafárik University, Jesenná 5, 041 54 Košice, Slovakia
marian.novotny@upjs.sk

Abstract. In the paper we design a protocol for sociometric questionnaires, which serves the privacy of responders. We propose a representation of a sociogram by a weighted digraph and interpret individual and collective phenomena of sociometry in terms of graph theory. We discuss security requirements for a privacy-aware protocol for sociometric questionnaires. In the scheme we use additively homomorphic public key cryptosystem [2], which allows single multiplication. We present the threshold version of the public key system and define individual phases of the scheme. The proposed protocol ensures desired security requirements and can compute sociometric indices without revealing private information about choices of responders.

1 Introduction

Sociometry is a quantitative method for measuring social relationships. It was developed by the psychotherapist Jacob L. Moreno in his studies of the relationship between social structures and psychological well-being [9].

This method is based on *choices* of individuals from a certain social group. Responders are asked to choose one or more persons from the group according to specific criteria known in the whole group. The choices of responders are collected by a *questionnaire*. Relations between individuals can be represented by a *sociogram* – a graphic representation of social links that persons have.

Sociometric techniques can be used for effective management of a school class by a teacher or in team-building in organizations by managers. They can help to discover information about the group or individuals. On the other hand, it is desirable to protect the *privacy* of responders and shield them from misusing delicate information. Our aim is to develop a cryptographic protocol for collection and evaluation of sociometric questionnaires which ensures the desired security requirements, placing emphasis on the privacy of responders.

This paper is organized as follows. The next section introduces a representation of the sociogram in terms of *graph theory*. The section following next describes the proposed scheme for anonymous sociometric questionnaires. In the last section, we present our conclusions and suggestions for the future work.

[⋆] This work was supported in part by Slovak VEGA grant number 1/0035/09.

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2355

2 Representation of a Sociogram by Graph Theory

A sociogram can be represented by a *weighted digraph* [7] $G = (V, E)$, where nodes from V represent individuals from the social group. Each social link is represented by a *weighted arc* from the set $E \subseteq V \times V$. A weight function $w : E \rightarrow \{-s, \ldots, -1, 1, \ldots, s\}$ expresses rates of social links. Common values of the parameter *scale* s include $1, 3$, or 5. We say that an arc is *positive* (*negative*) if and only if the weight of the arc is positive (negative).

2.1 Characteristics of a Node

The number of tail endpoints adjacent to a node v is called *indegree* of the node v, i.e., $\deg^{In}(v) = |\{u \in V; \langle u, v\rangle \in E\}|$. It stands for the number of social links to the corresponding person. We distinguish between the *positive indegree* $\deg^{In^+}(v)$ and the *negative indegree* $\deg^{In^-}(v)$ of a node v, where $\deg^{In}(v) = \deg^{In^+}(v) + \deg^{In^-}(v)$. The positive (negative) indegree expresses the number of positive (negative) arcs incident to the node.

The number of head endpoints adjacent from a node v is called *outdegree* of the node v, i. e., $\deg^{Out}(v) = |\{u \in V; \langle v, u\rangle \in E\}|$. It stands for the number of social links from the person. Analogically, we define *positive* (*negative*) *outdegree* of a node v $\deg^{Out^+}(v)$ $(\deg^{Out^-}(v))$.

We also distinguish between *positive* and *negative weighted indegree* (*outdegree*). The sum of weights of all positive arcs incident to a node v is called positive weighted indegree, i. e., $In^+(v) = \sum\limits_{u \in V, \langle u,v\rangle \in E, w(u,v)>0} w(u, v)$. The sum of weights of all negative arcs incident from a node v is called negative weighted outdegree, i. e., $Out^-(v) = \sum\limits_{u \in V, \langle v,u\rangle \in E, w(v,u)<0} w(v, u)$. Similarly, we define for a node v negative weighted indegree $In^-(v)$ and positive weighted outdegree $Out^+(v)$.

2.2 Sociometric Indices and Objects

There exist two approaches to a sociogram – *individual* and *collective phenomena.* Individual phenomena include individual sociometric indices and objects such as *stars, isolates, ghosts.* In the latter case, collective phenomena include group sociometric indices and structures such as *dyads* and *mutual choices.*

Individual sociometric indices can be computed from the above defined characteristics of a node. For example, *positive social status* of a node p is defined as $\frac{In^+(p)}{|V|-1}$. In similar way, objects such as stars, outsiders, ghosts and isolates can be recognized from individual characteristics of nodes.

A star q is a node with the maximal positive weighted indegree, i. e., $In^+(q) = max\{In^+(v); v \in V\}$. An outsider o is a node with the minimal negative weighted indegree, i. e., $In^-(o) = min\{In^-(v); v \in V\}$. A ghost g is a node with zero indegree and outdegree, i. e., $\deg^{In}(g) + \deg^{Out}(g) = 0$. Finally, an isolate i is a node with zero positive indegree, which is not a ghost, i. e., $\deg^{In^+}(i) = 0 \wedge \deg^{Out}(i) > 0$.

A *dyad* is the smallest and the most elementary social unit, i. e., a group of two members with a mutual choice. We distinguish between *positive, negative* and *combined mutual choices.* In the positive (negative) mutual choice the members positively (negatively) choose each other. In the combined mutual choice, one member chooses positively, but the other one chooses negatively.

A set of positive mutual choices M^+ is defined as $M^+ = \{\{u, v\} \subseteq V, \{\langle u, v\rangle, \langle v, u\rangle\} \subseteq E, w(u, v) > 0, w(v, u) > 0\}$. Similarly, we define a set of negative mutual choices M^- and a set of combined mutual choices $M^\pm$. A set of all mutual choices M is defined as $M = M^+ \cup M^- \cup M^\pm$.

Using above mentioned definitions, we define a group sociometric index – a *coherence* of the group. A *positive coherence* of a group is defined as $coh^+ = \frac{|M^+|}{\binom{|V|}{2}}$. Similarly, we define a *negative (combined) coherence* coh^- ($coh^\pm$).

3 The Proposed Scheme

3.1 The Homomorphic Public-Key System

For encryption of responders' choices we use the homomorphic public-key system from the paper [2], which is *additively homomorphic.* Moreover, it allows us to use a single multiplication. Inter alia, this property is used for computing the cardinality of the set of mutual choices. The encryption system is *semantically secure* assuming the *subgroup decision assumption* [2].

The Key Generation. The construction of the homomorphic scheme from the paper [2] requires to use certain finite groups of composite order that support a *bilinear map.* Let $\mathbb{G}$ and $\mathbb{G}_1$ be two multiplicative cyclic groups of finite order n, where g is a generator of $\mathbb{G}$. Let e denote a bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_1$. It holds, that for all $u, v \in \mathbb{G}$ and $a, b \in \mathbb{Z}$, we have $e(u^a, v^b) = e(u, v)^{ab}$. Moreover, $e(g, g)$ is a generator of $\mathbb{G}_1$.

The key setup works as follows:

- Generate two random primes q_1, q_2 and set $n = q_1 \cdot q_2 \in \mathbb{Z}$.
- Generate a bilinear group $\mathbb{G}$ of order n following the paper [2]. Let g, u be random generators of $\mathbb{G}$ and $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_1$ be the bilinear map. Then $h = u^{q_2}$ is a random generator of the subgroup $\mathbb{G}$ of order q_1.
- The public key is $Pk = (n, \mathbb{G}, \mathbb{G}_1, e, g, h)$. The private key is $Sk = q_1$.

Encryption and Decryption. To encrypt a message $m \in \{0, \ldots, q_2 - 1\}$, a sender chooses a random number $r \in \mathbb{Z}_{n-1}$ and computes the ciphertext $C = g^m h^r \in \mathbb{G}$. To decrypt the ciphertext C using the private key $Sk = q_1$, observe that $C^{q_1} = (g^m h^r)^{q_1} = (g^{q_1})^m$. It is sufficient to compute the discrete logarithm of C^{q_1} base g^{q_1} in order to recover the plaintext m. For our purposes, the message space is bounded by the value $b = \max\{s(|V|-1), \binom{|V|}{2}\}$[1], where s is the scale and $|V|$ is the number of nodes in a sociogram. This way it is sufficient to pre-compute the table of powers $(g^{q_1})^0, \ldots, (g^{q_1})^b$. Using binary-search, one can find an appropriate m in the logarithmic time according to the number of nodes.

Homomorphic Properties. The encryption system is clearly additively homomorphic. Given ciphertexts $C_1, C_2 \in \mathbb{G}$ which are encryptions of plaintexts m_1, m_2, anyone can create an encryption of $m_1 + m_2 \bmod n$ by computing the product $C_1C_2 = g^{m_1}h^{r_1}g^{m_2}h^{r_2} = g^{m_1+m_2}h^{r_1+r_2}$. Note that we can multiply an encrypted message m by an integer $z \in \mathbb{Z}^+$. Given the ciphertext $C = g^m h^r$, anyone can create an encryption of $zm \bmod n$ by computing the exponentiation $C^z = g^{zm}h^{zr}$.

Anyone can once multiply two encrypted messages m_1, m_2 using the bilinear map e. Set $g_1 = e(g,g) \in \mathbb{G}_1$ and $h_1 = e(g,h) \in \mathbb{G}_1$. Then, g_1 is of order n and h_1 is of order q_1. For given ciphertexts $C_1 = g^{m_1}h^{r_1}, C_2 = g^{m_2}h^{r_2} \in \mathbb{G}$ we build an encryption of $m_1 \cdot m_2$ as $C_1 * C_2 = e(C_1, C_2) = e(g^{m_1}h^{r_1}, g^{m_2}h^{r_2}) = e(g^{m_1+\alpha q_2 r_1}, g^{m_2+\alpha q_2 r_2}) = g_1^{m_1 m_2} h_1^{m_1 r_2 + r_2 m_1 + \alpha q_2 r_1 r_2} \in \mathbb{G}_1$, where $h = g^{\alpha q_2}$. Note that the system is still additively homomorphic in $\mathbb{G}_1$.

The Robust Threshold Version. The goal of the *threshold* version (t,l) of the cryptosystem is to share the private key q_1 among l authorities by a threshold secret sharing scheme. A ciphertext can be decrypted when at least $t+1$ shareholders cooperate on decryption in the group $\mathbb{G}_1$. Note that it is sufficient to decrypt only in the group $\mathbb{G}_1$ since we can use the bilinear map to move the ciphertext from $\mathbb{G}$ to the group $\mathbb{G}_1$ without changing the plaintext.

For simplicity, we assume that a trusted dealer first generates the public key including $n = q_1 \cdot q_2$ and the private key q_1. The dealer distributes shares of the private key between l authorities. The shares are created following the technique [13, 5], which is a modification of the *Shamir secret sharing* scheme [12] over $\mathbb{Z}_n$. The dealer sets $a_0 = q_1$ and chooses a_i at random from $\{0, \ldots, n-1\}$ for $i \in \{1, \ldots, t\}$. The numbers $a_0, \ldots, a_t$ define the polynomial $f(X) = \sum_{i=0}^{t} a_i X^i \in \mathbb{Z}[X]$. For each shareholder $i \in \{1, \ldots, l\}$ the dealer computes $s_i = f(i) \bmod n$. Let $\varDelta = l!$ and $\varDelta^\star = \varDelta^{-1} \bmod n$. For any subset P of $t+1$ indices from $\{1, \ldots, l\}$ the modified *Lagrange coefficients* are defined as $\lambda_{i,P} = \varDelta \frac{\prod_{i' \in P/\{i\}} -i'}{\prod_{i' \in P/\{i\}} i-i'} \bmod n$. From the *Lagrange interpolation* we have $\varDelta \cdot f(0) = \sum_{i \in P} \lambda_{i,P} f(i) \bmod n$, i. e., $\varDelta \cdot q_1 = \sum_{i \in P} \lambda_{i,P} s_i \bmod n$.

Moreover, a shareholder i which possesses the secret s_i publishes $y_i = g_1^{s_i}$ in order to make a process of decryption verifiable. To decrypt a ciphertext $C = g_1^m h_1^r \in \mathbb{G}_1$ without reconstructing the secret q_1 each shareholder i publishes $u_i = C^{s_i}$ and following the Chaum-Pedersen protocol [3] proves that $\log_{g_1} y_i = \log_C u_i$. From any subset of $t+1$ participants P who passed the proof

[1] The value $s(|V|-1)$ is the maximal possible absolute value of weighted degrees and $\binom{|V|}{2}$ is the maximum cardinality of the set of mutual choices.

the value $g_1^{\Delta q_1 m}$ is computed as as $\prod_{i \in P} u_i^{\lambda_{i,P}} = \prod_{i \in P} C^{\lambda_{i,P} s_i} = C^{\sum_{i \in P} \lambda_{i,P} s_i} = (g_1^m h_1^r)^{\Delta \cdot q_1} = g_1^{\Delta q_1 m}$. After computing $(g_1^{\Delta q_1 m})^{\Delta^\star} = g_1^{q_1 m}$, the plaintext m can be recovered by comparing with pre-computed tables of powers of $g_1^{q_1}$ as mentioned above.

The zero-knowledge proofs of correct partial decryption [3] from each shareholder can be performed interactively between shareholders and transcripts of such interactions are made public for verification. In order to make these proofs non-interactive, the verifier could be implemented using either a trusted *source of random bits* [10] or using the *Fiat-Shamir heuristic* [4] which requires a hash function. In the latter case security is obtained from the *random oracle model* [8].

3.2 Security Requirements for a Scheme

The scheme is expected to satisfy certain security requirements which are relevant for a privacy-aware protocol for sociometric questionnaires. The social group which consists of responders is defined in the questionnaire. It also contains a selection criterion and other parameters such as the deadline for filling. We enumerate and informally discuss security requirements in the following list.

- *Eligibility.* Only valid responders who are defined as members of the group are eligible to correctly fill in the questionnaire.
- *Privacy.* In the evaluation process, choices of a responder must not identify the responder and any traceability between the responder and his choices must be removed.
- *Verifiability.* Any responder should be able to *individually* verify whether his choices were correctly recorded and accounted. Moreover, anyone can *universally* verify that in the evaluation process only valid choices of eligible responders were recorded and the counting process was accurate.
- *Accuracy.* The scheme must be error-free. The final computations of sociometric indices must correspond with all choices of all responders.

Note that these requirements are similar to security requirements for e-voting protocols [11]. However, the submission of choices and computations of the results differ from usual e-voting protocols. On the other hand, a scheme does not need to ensure requirements such as *receipt-freeness* or *incoercibility* [11], because we do not expect "choice-buying" of responder's choices.

3.3 The Proposed Scheme

The realization of the scheme consists of various phases. First, the *questioner* creates a questionnaire in which he defines a social group of responders $R_1, \ldots, R_N$ and sociometric indices which have to be computed. He also sets the deadline for filling and the sociometric parameters such as the scale s for the weights of the arcs. Then, he registers the questionnaire by the *collector*. The collector collects submissions of responders, checks signatures, leads the computations and publishes results. The registration of responders $R_1, \dots, R_N$ is based on digital signatures. Therefore, we assume a pre-established *Public Key Infrastructure* with registered conceivable responders and other participants with relevant *certificates* of public keys for digital *signature* [8].

For encryption of choices, we use the above mentioned robust threshold (t, l) version of the public-key scheme [2], where the private key $Sk = q_1$ is shared between l authorities. For simplicity, we assume that a trusted dealer first generates the public key Pk and the private key $Sk = q_1$. Then, the dealer creates and distributes shares of the private key between l authorities and finally deletes the private key.

The process of decryption is realized by cooperation of at least $t+1$ authorities and is universally verifiable as mentioned above. Note that we do not specify who should be shareholders, since it depends on the usage of the protocol. However, the robust threshold version of the cryptosystem ensures the robustness of the protocol.

Submitting Choices. A responder R_i fills in the questionnaire, i. e., defines all relations from the node R_i in the sociogram. To represent a relation from the node R_i to node R_j we use $s + 2$ bits $b_{ij}^{+}, b_{ij}^{-}, b_{ij}^{w_1}, \dots, b_{ij}^{w_s}$, where s is the scale as defined in Section 2. The bits b_{ij}^{+}, b_{ij}^{-} indicate whether the weight of the arc is positive, negative, or there is missing arc. The bit $b_{ij}^{w_{|w_{ij}|}} = 1$ defines the absolute value of the weight of the arc $|w_{ij}|$. We consider three possible relations from the node R_i to the node R_j:

- The arc $\langle R_i, R_j \rangle$ has a positive weight $w_{ij} > 0$, then $b_{ij}^{+} = 1, b_{ij}^{w_{w_{ij}}} = 1$, and other bits are 0;
- The arc $\langle R_i, R_j \rangle$ has a negative weight $w_{ij} < 0$, then $b_{ij}^{-} = 1, b_{ij}^{w_{|w_{ij}|}} = 1$, and other bits are 0;
- There is missing arc $\langle R_i, R_j \rangle$, then an arbitrary bit $b_{ij}^{w_a} = 1$, where $a \in \{1, \dots, s\}$ and other bits are 0.

Note that, when the parameter scale $s = 1$, it is sufficient to represent a relation from the node R_i to the node R_j with just two bits b_{ij}^{+}, b_{ij}^{-}.

For each responder $R_j, j \neq i$ each bit $b_{ij}^{\diamondsuit}, \diamondsuit \in \{+, -, w_1, \dots, w_s\}$ is encrypted by responder R_i using the public key Pk as $c_{ij}^{\diamondsuit} = E_{Pk}(b_{ij}^{\diamondsuit})$. All these encrypted bits are sent along with the signature of the encrypted bits by the responder R_i to the collector.

Verification of Submissions. The collector checks the validity of signatures of all submissions of responders $R_1, \dots, R_N$. If the responder R_i does not submit his choices in time, or his signature is incorrect, then he is disqualified from the set of responders. The encrypted relations to the node R_i are excluded as well. Finally, the collector publishes submissions with correct signatures in order to verification.

In the e-voting protocols based on homomorphic encryption, are usually used zero-knowledge proofs for verification of validity of ballots [11]. These proofs are used in the non-interactive version using Fiat-Shamir heuristic [4]. As a bonus of the public key system, we do not need to use these proofs according to verification of validity of submissions.

The submissions of responders in the bit representation are valid, if the following conditions hold:

1. $b_{ij}^{\diamondsuit} \in \{0,1\}$, which is equivalent to the formula $b_{ij}^{\diamondsuit} \cdot (b_{ij}^{\diamondsuit} - 1) = 0$, where $i \neq j, \diamondsuit \in \{+, -, w_1, \ldots, w_s\}$;
2. $b_{ij}^{+} \cdot b_{ij}^{-} = 0$, where $i \neq j$;
3. $\sum_{k=1}^{s} b_{ij}^{w_k} = 1$, which is equivalent to the formula $\sum_{k=1}^{s} b_{ij}^{w_k} - 1 = 0$, where $i \neq j$.

We need to verify all these equations of the form – left side le is equal to zero. We can use the homomorphic properties for preparing ciphertexts of le for $(s+2)N(N-1)$ equations of the first type, $N(N-1)$ of the second and $N(N-1)$ of the third type. We have to check $v = (s+4)N(N-1)$ equations total.

To prepare ciphertexts of equations of first and third type the collector publishes a deterministic encryption of $-1 \bmod n$. The equations can be checked by shareholders by v cooperatively-made decryptions. To decrease the computation complexity, the shareholders check simultaneously a batch of equations $\sum_{i=1}^{v} r_i \cdot le_i = 0$, where r_i are chosen cooperatively by shareholders. They can run a binary search to identify the invalid submissions following the technique from [1]. This way, in the optimistic scenario (when all submissions are valid) is used just one decryption of shareholders.

Computations of the Sociometric Indices. We define computations in the bit representation of a sociogram as shown in Table 1. Let J_i denote the set $\{1, \ldots, N\}/\{i\}$. A relation from a node R_i to a node R_j is represented by bits $b_{ij}^{+}, b_{ij}^{-}, b_{ij}^{w_1}, \ldots, b_{ij}^{w_s}$. If there exists an arc $\langle R_i, R_j \rangle$, the value $|w_{ij}| = \sum_{k=1}^{s} k \cdot b_{ij}^{w_k}$ represents the absolute value of the weight of the arc $\langle R_i, R_j \rangle$. If there is no arc $\langle R_i, R_j \rangle$, the value $|w_{ij}| = \sum_{k=1}^{s} k \cdot b_{ij}^{w_k} = a$, since exactly one arbitrary chosen bit $b_{ij}^{w_a} = 1$ as defined above. Note that it is easy to show that the definitions of

Table 1. Computations in the bit representation of a sociogram

$\deg^{In^+}(R_i) = \sum_{j \in J_i} b_{ji}^{+}$	$\deg^{In^-}(R_i) = \sum_{j \in J_i} b_{ji}^{-}$	$\deg^{Out^+}(R_i) = \sum_{j \in J_i} b_{ij}^{+}$
$\deg^{Out^-}(R_i) = \sum_{j \in J_i} b_{ij}^{-}$	$\deg^{In}(R_i) = \sum_{j \in J_i} b_{ji}^{+} + b_{ji}^{-}$	$\deg^{Out}(R_i) = \sum_{j \in J_i} b_{ij}^{+} + b_{ij}^{-}$
$In^+(R_i) = \sum_{j \in J_i} b_{ji}^{+} \cdot \lvert w_{ji} \rvert$	$In^-(R_i) = -\sum_{j \in J_i} b_{ji}^{-} \cdot \lvert w_{ji} \rvert$	
$\lvert M^+ \rvert = \sum_{i=1}^{N} \sum_{j>i} b_{ij}^{+} \cdot b_{ji}^{+}$	$\lvert M^{\pm} \rvert = \sum_{i=1}^{N} \sum_{j>i} (b_{ij}^{-} \cdot b_{ji}^{+}) + (b_{ij}^{+} \cdot b_{ji}^{-})$	
$\lvert M^- \rvert = \sum_{i=1}^{N} \sum_{j>i} b_{ij}^{-} \cdot b_{ji}^{-}$	$\lvert M \rvert = \sum_{i=1}^{N} \sum_{j>i} (b_{ij}^{-} \cdot b_{ji}^{+}) + (b_{ij}^{+} \cdot b_{ji}^{-}) + (b_{ij}^{+} \cdot b_{ji}^{+}) + (b_{ij}^{-} \cdot b_{ji}^{-})$	

computations from Table 1 correspond with the definitions from Section 2.

Computations on Encrypted Sociogram. The collector computes the value c_{ij}^{w} from encrypted values $c_{ij}^{w_1}, \ldots, c_{ij}^{w_s}$, i. e., $c_{ij}^{w} = \prod_{k=1}^{s} (c_{ij}^{w_k})^k = \prod_{k=1}^{s} E_{Pk}(b_{ij}^{w_k})^k = \prod_{k=1}^{s} E_{Pk}(k \cdot b_{ij}^{w_k}) = E_{Pk}(\sum_{k=1}^{s} k \cdot b_{ij}^{w_k}) = E_{Pk}(|w_{ij}|)$. For an encrypted representation of a relation from the node R_i to R_j we use values $c_{ij}^{+}, c_{ij}^{-}, c_{ij}^{w}$ in the encrypted sociogram.

The ciphertext of the positive indegree of a node R_i is computed as $\prod_{j \in J_i} c_{ji}^{+} = \prod_{j \in J_i} E_{Pk}(b_{ji}^{+}) = E_{Pk}(\sum_{j \in J_i} b_{ji}^{+}) = E_{Pk}(\deg^{In^+}(R_i))$. Similarly, we can compute the ciphertext of the negative indegree $E_{Pk}(\deg^{In^-}(R_i))$. Finally the ciphertext of the indegree of the node R_i is $E_{Pk}(\deg^{In}(R_i)) = \prod_{j \in J_i} c_{ji}^{+} c_{ji}^{-}$. Analogously, we can compute ciphertexts of outdegrees, for example the encryption of the positive outdegree $E_{Pk}(\deg^{Out^+}(R_i)) = \prod_{j \in J_i} c_{ij}^{+}$.

To compute encryptions of weighted degrees, we use also the multiplicative property of the homomorphic system. The ciphertext of positive weighted indegree of the node R_i can be computed as $\prod_{j \in J_i} c_{ji}^{+} * c_{ji}^{w} = \prod_{j \in J_i} E_{Pk}(b_{ji}^{+}|w_{ji}|) = E_{Pk}(\sum_{j \in J_i} b_{ji}^{+}|w_{ji}|) = E_{Pk}(In^{+}(R_i))$. Similarly, we can compute other weighted degrees.

Anyone can compute the encrypted value of the cardinality of the set of positive mutual choices as $\prod_{i=1}^{N} \prod_{j>i} c_{ij}^{+} * c_{ji}^{+} = \prod_{i=1}^{N} \prod_{j>i} E_{Pk}(b_{ij}^{+} b_{ji}^{+}) = \prod_{i=1}^{N} E_{Pk}(\sum_{j>i} b_{ij}^{+} b_{ji}^{+}) = E_{Pk}(\sum_{i=1}^{N} \sum_{j>i} b_{ij}^{+} b_{ji}^{+}) = E_{Pk}(|M^{+}|)$. The set of negative and the set of combined mutual choices are defined similarly. The ciphertext of the cardinality of the set of all mutual choices one can count as $\prod_{i=1}^{N} \prod_{j>i} (c_{ij}^{+} * c_{ji}^{+})(c_{ij}^{+} * c_{ji}^{-})(c_{ij}^{-} * c_{ji}^{+})(c_{ij}^{-} * c_{ji}^{-})$.

This way we derived encrypted values of individual and collective phenomena with respect to definitions from Section 2 only by using homomorphic properties of the encryption system. Note that the process of computations is universally verifiable by anyone. After computing and publishing encrypted sociometric indices, the shareholders of the private key $Sk = q_1$ individually verify the correctness of computation and cooperate to decrypt the desired sociometric indices. The process of decryption is universally verifiable by anyone including the responders, the collector and the questioner. Finally, the collector publishes obtained sociometric indices which express quantitative information about individuals or the group.

4 Conclusions

In this paper we designed the protocol for anonymous sociometric questionnaires. In the protocol each responder sends only one message. To prepare the submission costs $(N-1)(s+2)$ encryptions of the cryptosystem [2] and one digital signature, where N is the number of responders and s is the parameter scale. The protocol guarantees the security requirements from Section 3. 2. The eligibility property is ensured by digital signatures of the submissions by responders and checking of the validity of submissions. The signatures are checked by the collector and verified by anyone. The validity of submissions is checked by shareholders and verified by anyone. The privacy of responders is provided by the public key cryptosystem [2], which is semantically secure and homomorphic operations are also commutative. The process of computation and decryption of sociometric indices is universally verifiable according to universal verifiability of the threshold version of the cryptosystem and the defined computations on encrypted sociogram.

In the future work we are planning to formal model and analyze the scheme in the *applied pi-calculus*. For a future design of the protocol, recently announced fully homomorphic public key encryption scheme [6] looks promisingly. The results from the paper were presented as a talk on Primelife/IFIP Summer School 2009 – Privacy and Identity Management for Life

References

1. Bellare, M., Garay, J., Rabin, T.: Fast batch verification for modular exponentiation and digital signatures. EUROCRYPT'98, LNCS, vol. 1403. Springer (1998)
2. Boneh, D., Goh, E.-J., Nissim, K.: Evaluating 2-DNF formulas on ciphertexts. TCC '05. LNCS, vol. 3378. Springer (2005)
3. Chaum, D., Pedersen, T.: Wallet databases with observers. CRYPTO '92. LNCS, vol. 740. Springer (1993)
4. Fiat, A., Shamir, A.: How to prove yourself: practical solutions to identification and signature problems. Advances in cryptology—CRYPTO '86. LNCS, vol. 263. Springer (1987)
5. Gang, Q., et al.: Information-theoretic secure verifiable secret sharing over RSA modulus. Wuhan University Journal of Natural Sciences, vol. 11. Springer (2006)
6. Gentry, C.: Fully homomorphic encryption using ideal lattices. STOC '09. ACM (2009)
7. Hanneman, R. A., Riddle, M.: Introduction to social network methods. Riverside, University of California (2005)
8. Mao, W.: Modern Cryptography: Theory and Practice. Prentice Hall Professional Technical Reference (2003)
9. Moreno, J. L.: Who Shall Survive? Foundations of Sociometry, Group Psychotherapy and Sociodrama. Beacon House, Inc. (1953)
10. Rabin, M. O.: Transaction Protection by Beacons. Journal of Computer and System Sciences, vol. 27(2). Elsevier (1983)
11. Sampigethaya, R., Poovendran, R.: A Framework and Taxonomy for Comparison of Electronic Voting Schemes. Elsevier Computers & Security, vol. 25 (2006)
12. Shamir, A.: How to share a secret, Commun. ACM 22, vol. 11 (1979)
13. Shoup, V.: Practical Threshold Signatures. EUROCRYPT 2000. LNCS, vol. 1807. Springer (2000)

A Quantitative Characterization of Weighted Kripke Structures in Temporal Logic

Uli Fahrenberg, Kim G. Larsen, and Claus Thrane

Dept. of Computer Science, Aalborg University, Denmark
{uli,kgl,crt}@cs.aau.dk

Abstract. We extend the usual notion of Kripke Structures with a weighted transition relation, and generalize the usual Boolean satisfaction relation of CTL to a map which assigns to states and temporal formulae a real-valued distance describing the degree of satisfaction. We describe a general approach to obtaining quantitative interpretations for a generic extension of the CTL syntax, and show that, for one such interpretation, the logic is both *adequate* and *expressive* with respect to quantitative bisimulation.

1 Introduction

We present a general approach to quantitative analysis and approximate characterizations of *weighted Kripke strucures* (WKS) using formulae expressed using a weighted extension of CTL (WCTL). The theory presented here is an extension of a general framwork for quantitative analysis of reactive systems presented in [5].

The goal of [5] was to set the scene for a generic approach to simulation-based analysis, measuring the degree with which one system may simulate another. Developing this paradigm, the current objective is to extend the analysis to verification of *specifications in temporal logic.* Thus we introduce here a quantitative semantics for WCTL which lifts the usual Boolean satisfaction relation of the logic to a function mapping formulae and states into $\mathbb{R}_{\geq 0} \cup \{\infty\}$, and we show that with this semantics, WCTL is both *adequate* and *expressive* with respect to one of the quantitative bisimulation relations introduced in [5].

Using logics for analysis of concurrent and reactive systems is a well-established approach [1], but the standard qualitative techniques are arguably insufficient when reasoning about *quantitative* aspects. Indeed, it can be argued that in a setting where system models and properties include both discrete and continuous, *i.e.* quantitative, information, *e.g.* real-time or probabilistic systems, a quantitative approach is necessary.

The notion of quantitative analysis is closely related to *robustness*, *i.e.* the tolerance for estimation errors and imprecision in order to provide more realistic analysis for real-world applications. Existing work on quantitative logics comparable to ours includes [3] which presents an interpretation with relaxed timing constraints for *timed CTL* and a discounted notion of quantitative CTL where discounting is applied according to the depth of a subformula.

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2345

Another related work is [2], which presents an alternative approach to quantifying versions of LTL and μ-calculus, giving a mapping from states and formulae to the interval $[0, 1]$, where formulae are interpreted over a notion of quantitative transition systems.

In both [2] and [3], quantitative information is only evaluated for atomic propositions, where as path operators are only used to propagate the values obtained at subformulae. Moreover, the semantic interpretations measure only a point-wise property similar to one also discussed in [5], whereas the semantics in the present work accumulates quantitative information based on the paths used to evaluate formulae.

2 Weighted Kripke Structures and Bisimulation

We present a notion of *weighted Kripke structures* (WKS) and bisimulation based measurements for these. The following definition represents a straight forward extension of Kripke structures with weight functions labelling each transition, which may be interpreted as the cost of taking transitions in the structure. This extension is similar to the one presented in [5] for labelled transition systems, thus the results presented in this paper are transferable to our setting.

Definition 1. *For a finite set $\mathcal{AP}$ of atomic propositions, a* weighted Kripke structure *is a quadruple $M = (S, T, \mathcal{L}, w)$ where*

- *S is a finite set of states,*
- *$T \subseteq S \times S$ is a transition relation*
- *$\mathcal{L} \; : S \to 2^{\mathcal{AP}}$ is the proposition labelling, and*
- *$w \; : T \to \mathbb{R}_{\geq 0}$ assigns weights to transitions.*

We write $s \to s'$ instead of $(s, s') \in R$ and $s \xrightarrow{w} s'$ to indicate $w(s, s') = w$.

A *(weighted) path* in a $M = (S, T, \mathcal{L}, w)$ is a (possibly infinite) sequence $\sigma = ((s_0, w_0), (s_1, w_1), (s_2, w_2) \cdots)$ with $(s_i, w_i) \in S \times \mathbb{R}_{\geq 0}$ and such that $s_i \to s_{i+1}$ and $w_i = w(s_i, s_{i+1})$ for all i. We denote by $\mathsf{P}(s)$ the set of paths in M starting at state s. Given path σ, we write $\sigma(i) = (\sigma(i)_s, \sigma(i)_w)$ for its i-th state-weight pair, and σ^i for the suffix starting at $\sigma(i)$.

Notice that we have restricted ourselves to *finite* weighted Kripke structures here, *i.e.* structures with a finite set of states and finitely many atomic propositions. Our characterization results in Section 4 only hold for such finite structures.

2.1 Quantitative Bisimulation

We extend the standard notion of *strong bisimulation* [4] to distances (formally pseudometrics, see below) over WKS, thereby filling the gap between *unweighted* and *weighted* strong bisimulation defined for WKS as follows:

Definition 2. *Let $(S, T, \mathcal{L}, w)$ be a WKS on a set $\mathcal{AP}$ of atomic propositions. A relation $B \subseteq S \times S$ is*

– *an* unweighted bisimulation *provided that for all $(s,t) \in B$, $\mathcal{L}(s) = \mathcal{L}(t)$ and*
  if $s \to s'$, then also $t \to t'$ where $(s',t') \in B$ for some $t' \in S'$,
  if $t \to t'$, also also $s \to s'$ where $(s',t') \in B$ for some $s' \in S$;
– *a* (weighted) bisimulation *provided that for all $(s,t) \in B$, $\mathcal{L}(s) = \mathcal{L}(t)$ and*
  if $s \xrightarrow{c} s'$, then also $t \xrightarrow{c} t'$ and $(s',t') \in B$ for some $t' \in S'$,
  if $t \xrightarrow{c} t'$, then also $s \xrightarrow{c} s'$ and $(s',t') \in B$ for some $s' \in S$.

We write $s \overset{u}{\sim} t$ if $(s,t) \in B$ for some unweighted bisimulation B, and $s \sim t$ if $(s,t) \in B$ for some weighted bisimulation B.

The idea is that, in order to relate structures, we do not always need perfect matching of transition weights, rather it is relevant to know how close weights are matched. Similar to the simulation distances of [5], we call a *bisimulation distance* any pseudometric on the states of a WKS which mediates between unweighted and weighted bisimilarity:

Definition 3. *A* bisimulation distance *on a WKS $(S, T, \mathcal{L}, w)$ is a function $d : S \times S \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ which satisfies the following for all $s_1, s_2, s_3 \in S$:*

– $d(s_1, s_1) = 0$,
– $d(s_1, s_2) + d(s_2, s_3) \geq d(s_1, s_3)$,
– $d(s_1, s_2) = d(s_2, s_1)$,
– $s_1 \sim s_2$ *implies* $d(s_1, s_2) = 0$ *and*
– $d(s_1, s_2) \neq \infty$ *implies* $s_1 \overset{u}{\sim} s_2$

The distance which we shall consider here corresponds to the *accumulated simulation distance* from [5], but we expect that results similar to the ones of this paper also are available for the other distances considered in [5]. Our distance is based on a distance of (infinite) sequences of real numbers, which is appropriate as for (s,t) in $\overset{u}{\sim}$ (or $\sim$), any path $(s, a, s_1, a_1 s_2, \dots) \in \mathsf{P}(s)$ must be matched by an equal-length path $(t, b, t_1, b_1, t_2, \dots) \in \mathsf{P}(t)$ with (s_i, t_i) in $\overset{u}{\sim}$ (respectively $\sim$).

If $a = (a_i)$ and $b = (b_i)$ are sequences representing the weights of such paths, then the following distance measures the *discounted* accumulated sum (in terms of absolute values) of the entries' differences:

$$d_+(a, b) = \sum_i \lambda^i |a_i - b_i| \tag{1}$$

Discounting, with a factor $\lambda \in]0, 1[$, ensures finiteness of such (possibly infinite) sums, by reducing the contribution from each step (difference) exponentially over time. For the remainder of this paper we fix a discounting factor $\lambda \in]0, 1[$.

By extending bisimulation with the d_+ distance, we collect a family of relations $\{\mathcal{R}_\varepsilon \subseteq S \times S\}$ (*i.e.* a map $\mathbb{R}_{\geq 0} \to 2^{S \times S}$) since, due to discounting, for each step the distance between each successor pair may grow:

Definition 4. *A family of relations* $\mathbf{R} = \{\mathcal{R}_\varepsilon \subseteq S \times S \mid \varepsilon > 0\}$ *on a WKS* $(S, T, \mathcal{L}, w)$ *is an* accumulating bisimulation family *provided that for all* $(s, t) \in \mathcal{R}_\varepsilon \in \mathbf{R}$, $\mathcal{L}(s) = \mathcal{L}(t)$ *and*

- *for all* $s \xrightarrow{c} s'$, *also* $t \xrightarrow{d} t'$ *with* $|c - d| \leq \varepsilon$ *for some* $d \in \mathbb{R}_{\geq 0}$ *and* $(s', t') \in \mathcal{R}_{\varepsilon'} \in \mathbf{R}$ *with* $\varepsilon' \leq \frac{\varepsilon - |c-d|}{\lambda}$,
- *for all* $t \xrightarrow{c} t'$, *also* $s \xrightarrow{d} s'$ *with* $|c - d| \leq \varepsilon$ *for some* $d \in \mathbb{R}_{\geq 0}$ *and* $(s', t') \in \mathcal{R}_{\varepsilon'} \in \mathbf{R}$ *with* $\varepsilon' \leq \frac{\varepsilon - |c-d|}{\lambda}$.

We write $s \overset{+}{\sim}_\varepsilon t$ *if* $(s, t) \in \mathcal{R}_\varepsilon \in \mathbf{R}$ *for an accumulating bisimulation family* $\mathbf{R}$.

An accumulating bisimulation family $\mathbf{R}$ gives raise to a bisimulation distance in the sense of Definition 3 by $d(s, t) = \inf\{\varepsilon \mid s \overset{+}{\sim}_\varepsilon t\}$. Observe the following easy facts:

Lemma 1.

1. *For* $\varepsilon \leq \varepsilon'$ *and members* $\mathcal{R}_\varepsilon, \mathcal{R}_{\varepsilon'} \in \mathbf{R}$ *of an accumulating bisimulation family,* $\mathcal{R}_\varepsilon \subseteq \mathcal{R}_{\varepsilon'}$.
2. *Given* $s \overset{+}{\sim}_\varepsilon t$, *then every path* $\sigma = (s_0, w_0, s_1, w_1 s_2, \dots) \in \mathsf{P}(s)$ *has a corresponding path* $\sigma' = (t_0, w'_0, t_1, w'_1 t_2, \dots) \in \mathsf{P}(t)$ *such that* $\varepsilon = \varepsilon_0$ *and* $s_i \overset{+}{\sim}_{\varepsilon_i} t_i$ *for all* i, *where* $\varepsilon_{i+1} = \frac{\varepsilon_i - |w_i - w'_i|}{\lambda}$.

Note that as we only consider finite WKS, all $\mathcal{R}_\varepsilon$ relations are finite. Also, we shall use the term *correspondence* between paths to denote the second property of the above lemma.

3 Weighted CTL

We now consider a generalization of the well-known CTL formalism to quantities. Our notion of *weighted CTL* (WCTL) is as usual defined in terms of state and path formulae. Notice that our syntactic extensions are restricted to path formulae, which are annotated with real numbers (weights). Satisfaction of a formula by a system is no longer interpreted as a true or false statement, but rather in terms of a real-valued distance. A smaller distance is to mean a closer (better) match of the specified weights in the formula, and 0 denotes the exact match, whereas ∞ indicates an incompatibility between the system and the specified atomic propositions of a formula. Hence in some sense, 0 corresponds to truth and ∞ to falsehood. We will use $[\![\varphi]\!](s) = \varepsilon$ to denote the value $\varepsilon \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ obtained by evaluating φ at state s.

For the remainder of this paper we fix a set $\mathcal{AP}$ of atomic propositions and a WKS $(S, T, \mathcal{L}, w)$. All definitions and results below will be given for the states of one single WKS, but we note that to relate states of different WKS, one can simply form the disjoint union.

Definition 5. *For* $\mathsf{p} \in \mathcal{AP}$*,* $\varPhi$ *generates the set of state formulae, and* $\varPsi$*, the set of path formulae, annotated by weights* $c \in \mathbb{R}_{\geq 0}$*, according to the following abstract syntax:*

$$\varPhi ::= \mathsf{p} \mid \neg\mathsf{p} \mid \varPhi_1 \wedge \varPhi_2 \mid \varPhi_1 \vee \varPhi_2 \mid \mathsf{E}\varPsi \mid \mathsf{A}\varPsi$$
$$\varPsi ::= \mathsf{X}_c\varPhi \mid \mathsf{G}_c\varPhi \mid \mathsf{F}_c\varPhi \mid [\varPhi_1\mathsf{U}_c\varPhi_2]$$

The logic WCTL is the set of state formulae, which we denote $\mathcal{L}_w(\mathcal{AP})$ *or simply* $\mathcal{L}_w$*.*

The annotated modalities in the above syntax specify requirements on weights in a WKS. Before discussing these exact requirements, let us consider the usual meaning of the CTL modalities, as well as how these may be generalized to adhere to the type of quantitative analysis considered in the previous section:

Given CTL propositions on the form $M, s \models \mathsf{E}\psi$ and $M, s \models \mathsf{A}\psi$, we may interpret these as infinite *existential*, respectively *universal*, quantifications over paths in M from s satisfying ψ. Similarly, $M, \sigma \models \mathsf{F}\varphi$ and $M, \sigma \models \mathsf{G}\varphi$ may be interpreted as an infinite *disjunction*, respectively *conjunction*, over propositions on the form: $M, s_i \models \varphi$ for $i \geq 0$, where s_i is a state on σ.

Using this observation, we expect that a generic approach to defining quantitative semantics, *i.e.* a function $\mathcal{L}_w \times S \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ for WCTL is obtainable. To this end, the standard sup and inf operators are reasonable generalization of E,A,F and G (interpreted as disjunction and conjunction over the standard Boolean domain) to the (complete) lattice $\mathbb{R}_{\geq 0} \cup \{\infty\}$.

Furthermore, this approach requires only modification to the evaluation (*i.e.* semantics) of path formulae. Observe that our semantics below specializes to the usual one in two ways: by mapping a distance $\varepsilon < \infty$ to true and ∞ to false, or by mapping 0 to true and $\varepsilon > 0$ to false.

In the following we present a *discounted accumulating semantics*, designed to match the d_+ distance (1), where weights of transition are accumulated (and discounted). Formally, the semantics of $\varphi \in \mathcal{L}_w$ defines a map from the set of states S to the set $\mathbb{R}_{\geq 0} \cup \{\infty\}$. Given a state formula φ and a state s, an evaluation $[\![\varphi]\!](s) = \varepsilon$ means that s satisfies φ with distance ε. Also, given a path formulae ψ and a path σ, an evaluation $[\![\psi]\!](\sigma) = \varepsilon$ means that ψ holds along σ with distance ε. Conversely, ε describes how close s (or σ) satisfies the specified weights in the formula.

Definition 6. *Let* $\varphi, \varphi_1, \varphi_2$ *be state formulae and* ψ *a path formula. The valuation* $[\![\cdot]\!] : S \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ *is defined inductively. For state formulae:*

$$[\![\mathsf{p}]\!](s) = \begin{cases} 0 & \text{if } \mathsf{p} \in \mathcal{L}(s) \\ \infty & \text{otherwise} \end{cases} \qquad [\![\neg\mathsf{p}]\!](s) = \begin{cases} 0 & \text{if } \mathsf{p} \in \mathcal{AP} \setminus \mathcal{L}(s) \\ \infty & \text{otherwise} \end{cases}$$

$$[\![\varphi_1 \vee \varphi_2]\!](s) = \inf\left\{[\![\varphi_1]\!](s), [\![\varphi_2]\!](s)\right\} \qquad [\![\varphi_1 \wedge \varphi_2]\!](s) = \sup\left\{[\![\varphi_1]\!](s), [\![\varphi_2]\!](s)\right\}$$

$$[\![\mathsf{E}\psi]\!](s) = \inf\left\{[\![\psi]\!](\sigma) \mid \sigma \in \mathsf{P}(s)\right\} \qquad [\![\mathsf{A}\psi]\!](s) = \sup\left\{[\![\psi]\!](\sigma) \mid \sigma \in \mathsf{P}(s)\right\}$$

For path formulae:

$$\llbracket \varphi \rrbracket(\sigma) = \llbracket \varphi \rrbracket(\sigma(0)_s)$$

$$\llbracket \mathsf{X}_c \varphi \rrbracket(\sigma) = |c - \sigma(0)_w| + \lambda \llbracket \varphi \rrbracket(\sigma^1)$$

$$\llbracket \mathsf{F}_c \varphi \rrbracket(\sigma) = \inf_k \left(\Big| \sum_{j=0}^{k-1} \lambda^j \sigma(j)_w - c \Big| + \lambda^k \llbracket \varphi \rrbracket(\sigma^k) \right)$$

$$\llbracket \mathsf{G}_c \varphi \rrbracket(\sigma) = \sup_k \left(\Big| \sum_{j=0}^{k-1} \lambda^j \sigma(j)_w - c \Big| + \lambda^k \llbracket \varphi \rrbracket(\sigma^k) \right)$$

$$\llbracket \varphi_1 \mathsf{U}_c \varphi_2 \rrbracket(\sigma) = \inf_k \left(\Big| \sum_{j=0}^{k-1} \lambda^j \llbracket \varphi_1 \rrbracket(\sigma^j) - c \Big| + \lambda^k \llbracket \varphi_2 \rrbracket(\sigma^k) \right)$$

Note again that this interpretation matches the d_+ equation (1). To measure other types of quantitative properties of systems, one may define an alternative semantic valuation for paths.

4 Characterization

In this section we show that WCTL with accumulating semantics is adequate and expressive with respect to accumulating bisimilarity.

4.1 Adequacy

The link between accumulating bisimilarity and our accumulating semantics for WCTL is as follows:

Theorem 1. *For states $s, t \in S$, $s \overset{+}{\sim}_\varepsilon t$ if and only if $\big| \llbracket \varphi \rrbracket(s) - \llbracket \varphi \rrbracket(t) \big| \le \varepsilon$ for all $\varphi \in \mathcal{L}_w$.*

The proof follows from Lemmas 2 and 3 below.

Corollary 1. *For states $s, t \in S$, $s \overset{+}{\sim}_0 t$ if and only if $\llbracket \varphi \rrbracket(s) = \llbracket \varphi \rrbracket(t)$ for all $\varphi \in \mathcal{L}_w$.*

Lemma 2. *Let $s, t \in S$ with $s \overset{+}{\sim}_\varepsilon t$, and let $\sigma_s = (s, u, s_1, u_1, \ldots) \in \mathsf{P}(s)$, $\sigma_t = (t, v, t_1, v_1, \ldots) \in \mathsf{P}(t)$ be corresponding paths. Then $\big| \llbracket \varphi \rrbracket(s) - \llbracket \varphi \rrbracket(t) \big| \le \varepsilon$ for all state formulae φ, and $\big| \llbracket \varphi \rrbracket(\sigma_s) - \llbracket \varphi \rrbracket(\sigma_t) \big| \le \varepsilon$ for all path formulae φ.*

Proof. We prove the lemma by structural induction in φ. The induction base is clear, as $s \overset{+}{\sim}_\varepsilon t$ implies that $\mathsf{p} \in \mathcal{L}(s)$ if and only if $\mathsf{p} \in \mathcal{L}(t)$, hence $\llbracket \varphi \rrbracket(s) = \llbracket \varphi \rrbracket(t)$ for $\varphi = \mathsf{p}$ or $\varphi = \neg \mathsf{p}$. For the inductive step, we examine each syntactic construction in turn:

1. $\varphi = \varphi_1 \vee \varphi_2$
   There are four cases to consider, corresponding to whether $[\![\varphi_1]\!](s) \leq [\![\varphi_2]\!](s)$ or $[\![\varphi_1]\!](s) > [\![\varphi_2]\!](s)$, and whether $[\![\varphi_1]\!](t) \leq [\![\varphi_2]\!](t)$ or $[\![\varphi_1]\!](t) > [\![\varphi_2]\!](t)$. We show the proof for one of the "mixed" cases; the other three are similar or easier:
   Assume $[\![\varphi_1]\!](s) \leq [\![\varphi_2]\!](s)$ and $[\![\varphi_1]\!](t) > [\![\varphi_2]\!](t)$. Then $[\![\varphi_1 \vee \varphi_2]\!](s) - [\![\varphi_1 \vee \varphi_2]\!](t) = [\![\varphi_1]\!](s) - [\![\varphi_2]\!](t)$, and $[\![\varphi_1]\!](s) - [\![\varphi_1]\!](t) \leq [\![\varphi_1]\!](s) - [\![\varphi_2]\!](t) \leq [\![\varphi_2]\!](s) - [\![\varphi_2]\!](t)$, and by induction hypothesis, $-\varepsilon \leq [\![\varphi_1]\!](s) - [\![\varphi_1]\!](t)$ and $[\![\varphi_2]\!](s) - [\![\varphi_2]\!](t) \leq \varepsilon$.
2. $\varphi = \varphi_1 \wedge \varphi_2$. This is similar to the previous case.
3. $\varphi = \mathsf{E}\varphi_1$
   By definition of $[\![\mathsf{E}\varphi_1]\!]$ there is a path $\sigma \in \mathsf{P}(s)$ for which $[\![\varphi_1]\!](\sigma) = [\![\varphi]\!](s)$. By Lemma 1 there is a corresponding path $\sigma' \in \mathsf{P}(t)$, and from the induction hypothesis we know that $|[\![\varphi_1]\!](\sigma) - [\![\varphi_1]\!](\sigma')| \leq \varepsilon$. Thus $|[\![\varphi]\!](s) - [\![\varphi]\!](t)| \leq \varepsilon$.
4. $\varphi = \mathsf{A}\varphi_1$. This is similar to the previous case.
5. $\varphi = \mathsf{X}_c\varphi_1$
   By definition, $[\![\varphi]\!](\sigma_s) = \lambda[\![\varphi_1]\!](\sigma_s^1) + |c-u|$ and $[\![\varphi]\!](\sigma_t) = \lambda[\![\varphi_1]\!](\sigma_t^1) + |c-v|$ where $\sigma_s = s \xrightarrow{u} \sigma_s^1$ and $\sigma_t = t \xrightarrow{v} \sigma_t^1$. Since $s \overset{+}{\sim}_\varepsilon t$ and σ_s and σ_t correspond, we have $\sigma_s(1) \overset{+}{\sim}_{\varepsilon'} \sigma_t(1)$ with $\varepsilon' \leq \frac{\varepsilon - |u-v|}{\lambda}$, and by induction hypothesis $|[\![\varphi_1]\!](\sigma_t^1) - [\![\varphi_1]\!](\sigma_s^1)| \leq \varepsilon'$. Hence $\big|[\![\varphi]\!](\sigma_s) - [\![\varphi]\!](\sigma_t)\big| \leq \big||c-u| - |c-v|\big| + \lambda\big|[\![\varphi_1]\!](\sigma_s^1) - [\![\varphi_2]\!](\sigma_t^1)\big| \leq |u-v| + \varepsilon - |u-v| = \varepsilon$.
6. $\varphi = \mathsf{F}_c\varphi_1$
   By definition, $[\![\varphi]\!](\sigma_s) = \inf_k \big(\big|\sum_{j=0}^{k-1} \lambda^j \sigma(j)_w - c\big| + \lambda^k [\![\varphi]\!](\sigma^k)\big)$, hence there is a k for which the infimum is obtained. Now as σ_s and σ_t correspond, the infimum for $[\![\varphi]\!](\sigma_t)$ is obtained for the same k. Repeated use of the definition of $\overset{+}{\sim}_\varepsilon$ yields $\sigma_s(k) \overset{+}{\sim}_{\varepsilon'} \sigma_t(k)$ with $\varepsilon' \leq \lambda^{-k}\big(\varepsilon - \sum_{j=0}^{k-1} \lambda^j \big|\sigma_s(j)_w - \sigma_t(j)_w\big|\big)$, and $\big|[\![\varphi]\!](\sigma_s) - [\![\varphi]\!](\sigma_t)\big| \leq \varepsilon$ follows by the triangle inequality as in the previous case.
7. $\varphi = \mathsf{G}_c\varphi_1$. This is similar to the previous case.
8. $\varphi = \varphi_1 \mathsf{U}_c \varphi_2$
   Assume $[\![\varphi]\!](\sigma_s) = \delta$, then by definition there is a k such that $\lambda[\![\varphi_2]\!](\sigma_s^k) = \delta'$ and $\delta = \delta' + \big|\sum_{j=0}^{k-1} \lambda[\![\varphi_1]\!](\sigma_s^j) - c\big|$. Since σ_s and σ_t correspond, so do σ_s^j and σ_t^j for any j. Therefore by induction hypothesis, $\big|[\![\varphi_2]\!](\sigma_s^k) - [\![\varphi_2]\!](\sigma_t^k)\big| \leq \varepsilon$ and $\big|[\![\varphi_1]\!](\sigma_s^j) - [\![\varphi_1]\!](\sigma_t^j)\big| \leq \varepsilon$ for all $0 \leq j \leq k$. Again we can apply the triangle inequality to arrive at $\big|[\![\varphi]\!](\sigma_s) - [\![\varphi]\!](\sigma_t)\big| \leq \varepsilon$.

Lemma 3. *Let $s, t \in S$ and assume that $\big|[\![\varphi]\!](s) - [\![\varphi]\!](t)\big| \leq \varepsilon$ for all state formulae $\varphi \in \mathcal{L}_w$. Then $s \overset{+}{\sim}_\varepsilon t$.*

Proof. This follows directly from Theorem 2, but one can also observe that the family $\mathbf{R} = \{\mathcal{R}_\varepsilon\}$ defined by

$$\mathcal{R}_\varepsilon = \big\{(s,t) \mid s,t \in S, \forall \varphi \in \mathcal{L}_w : \big|[\![\varphi]\!](s) - [\![\varphi]\!](t)\big| \leq \varepsilon\big\}$$

is indeed an accumulating bisimulation in terms of Definition 4.

4.2 Expressivity

We show that WCTL with accumulating semantics is expressive with respect to accumulating bisimulation in the following sense:

Theorem 2. *For each $s \in S$ and every $\gamma \in \mathbb{R}_+$, there exists a state formula $\varphi_\gamma^s \in \mathcal{L}_w$ which characterizes s up to accumulating bisimulation and up to γ, i.e. such that for all $s' \in S$, $s \overset{+}{\sim}_\varepsilon s'$ if and only if $[\![\varphi_\gamma^s]\!](s') \in [\varepsilon - \gamma, \varepsilon + \gamma]$ for all γ.*

Proof. We define characteristic formulae of unfoldings, as follows: For each $s \in S$ and $n \in \mathbb{N}$, denote $\mathcal{L}(s) = \{\mathsf{p}_1, \dots, \mathsf{p}_k\}$ and $\mathcal{AP} \setminus \mathcal{L}(s) = \{\mathsf{q}_1, \dots, \mathsf{q}_\ell\}$ and let $\varphi(s, n)$ be the WCTL formula defined inductively as follows:

$$\varphi(s,0) = (\mathsf{p}_1 \wedge \dots \wedge \mathsf{p}_k) \wedge (\neg \mathsf{q}_1 \wedge \dots \wedge \neg \mathsf{q}_\ell)$$
$$\varphi(s,n+1) = \bigwedge_{s \xrightarrow{w} s'} \mathsf{EX}_w \varphi(s',n) \;\wedge \bigwedge_{w : s \xrightarrow{w} s'} \mathsf{AX}_w \Big(\bigvee_{s \xrightarrow{w} s'} \varphi(s',n) \Big) \wedge \varphi(s,0)$$

It is easy to see that $[\![\varphi(s,n)]\!](s) = 0$ for all n.

To complete the proof, one observes that for each $\gamma > 0$, there is $n(\gamma) \in \mathbb{N}$ such that $\varphi(s, n(\gamma))$ can play the role of φ_γ^s in the theorem. Intuitively this is due to discounting: The further the unfolding in $\varphi(s,n)$, the higher are the weights discounted, hence from some $n(\gamma)$ on, maximum weight difference is below γ.

References

1. Luca Aceto, Anna Ingólfsdóttir, Kim G. Larsen, and Jiři Srba. *Reactive Systems: Modelling, Specification and Verification.* Cambridge University Press, 2007.
2. Luca de Alfaro, Marco Faella, and Mariëlle Stoelinga. Linear and branching metrics for quantitative transition systems. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *ICALP*, volume 3142 of *Lecture Notes in Computer Science*, pages 97–109. Springer, 2004.
3. Thomas A. Henzinger, Rupak Majumdar, and Vinayak Prabhu. Quantifying similarities between timed systems. In *Proc. FORMATS'05*, volume 3829 of *Lecture Notes in Computer Science*, pages 226–241. Springer-Verlag, 2005.
4. Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.
5. Claus Thrane, Uli Fahrenberg, and Kim G. Larsen. Quantitative analysis of weighted transition systems. *Journal of Logic and Algebraic Programming*, 2009. To Appear.

Comparison of Algorithms for Checking Emptiness on Büchi Automata

Andreas Gaiser[1]⋆ and Stefan Schwoon[2]

[1] Institut für Informatik, Technische Universität München, Germany
[2] LSV, CNRS, ENS de Cachan, INRIA Saclay, France
gaiser@model.in.tum.de, schwoon@lsv.ens-cachan.fr

Abstract. We re-investigate the problem of LTL model-checking for finite-state systems. Typical solutions, like in Spin, work on the fly, reducing the problem to Büchi emptiness. This can be done in linear time, and a variety of algorithms with this property exist. Nonetheless, subtle design decisions can make a great difference to their actual performance in practice, especially when used on-the-fly. We compare a number of algorithms experimentally on a large benchmark suite, measure their actual run-time performance, and propose improvements. Compared with the algorithm implemented in Spin, our best algorithm is faster by about 33 % on average. We therefore recommend that, for on-the-fly explicit-state model checking, nested DFS should be replaced by better solutions.

1 Introduction

Model checking is the problem of determining whether a given hardware or software system meets its specification. In the automata-theoretic approach, the system may have finitely many states, and the specification is an LTL formula, which is translated into a Büchi automaton, intersected with the system, and checked for emptiness. Thus, model checking becomes a graph-theoretic problem.

Because of its importance, the problem has been intensively investigated. For instance, *symbolic* algorithms use efficient data structures such as BDDs to work on sets of states; a survey of them can be found in [5]. Moreover, *parallel* model-checking algorithms have been developed [1]. The best known symbolic or parallel solutions have suboptimal asymptotic complexity ($\mathcal{O}(n \log n)$, where n is the number of states), but are often faster than that in practice.

Büchi emptiness can also be solved in $\mathcal{O}(n)$ time. All known linear algorithms are *explicit*, i.e. they construct and explore states one by one, by depth-first search (DFS). Typically, they compute some data about each state: its unique *state descriptor* and some *auxiliary data* needed for the emptiness check. Since the state descriptor is usually much larger than the auxiliary data, approximative techniques such as bitstate hashing have been developed that avoid them, storing just the auxiliary information in a hash table [13]. This entails the risk of undetectable hash collisions; however the probability of a wrong result can be

⋆ The author was supported by the DFG Graduiertenkolleg 1480 (PUMA).

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2349

reduced below a chosen threshold by repeating the emptiness test with different hash functions. Thus they represent a trade-off between time and memory requirements. Henceforth, we shall refer to non-approximative methods that do use state descriptors as *exact* methods.

We further identify two subgroups of explicit algorithms: *Nested-DFS* methods directly look for acceptings cycle in a Büchi automaton; they need very little auxiliary memory and work well with bitstate hashing. *SCC-based* algorithms identify strongly connected components containing accepting cycles; they require more auxiliary memory but can find counterexamples more quickly.

All explicit algorithms can work "on-the-fly", i.e. the (intersected) Büchi automaton is not known at the outset. Rather, one begins with a Büchi automaton for the formula (typically small) and a compact system description and extracts the initial state from these. Successor states are computed during exploration as needed. If non-emptiness is detected, the algorithms terminate before constructing the entire intersection. Moreover, in this approach the transition relation need not be stored in memory. As we shall see, the on-the-fly nature of explicit algorithms is very significant when evaluating their performance properly.

In this paper, we investigate performance aspects of explicit, exact, on-the-fly algorithms for Büchi emptiness. The best-known example for such a tool is Spin [12], which uses the nested-DFS algorithm proposed by Holzmann et al [13], henceforth called HPY. The reasons for this choice are partly historic; the faster detection capabilities of SCC-based algorithm were not known when Spin was designed, having first been pointed out by Couvreur in 1999 [3]. Thus, the status of HPY as the best choice is questionable, all the more so since the memory advantages of nested DFS are comparatively scant in our setting. Moreover, improved nested DFS algorithms have been proposed in the meantime.

We therefore evaluate several algorithms based on their actual running time and memory usage on a large suite of benchmarks. Previous papers, especially those on SCC-based algorithms [10, 15, 4, 11], provided similar experimental results, however, experiments were few or random and unsatisfying in one important aspect: they worked from pre-computed Büchi automata, rather than truly on-the-fly. This aspect will play a significant role in our evaluation.

To summarize, this paper contains the following contributions and findings:

- We provide improvements in both subgroups, nested DFS and SCC-based. These concern the algorithms of Couvreur [3] and Schwoon/Esparza [15]. For new, self-contained proofs, see [7].
- One of the algorithms we study can be extended to generalized Büchi automata, and we investigate this aspect.
- We implemented existing and new algorithms and compare them on a large benchmark suite. We analyze the structural properties of Büchi automata that cause performance differences.

We make the following observations: The overall memory consumption of all algorithms is dominated by the state descriptors, the differences in auxiliary memory play virtually no role. The running times depend practically exclusively on the number of successor computations. When experimenting with pre-computed automata – as done in some other papers – this operation becomes cheap, which causes misleading results. Our results allow to derive detailed recommendations which algorithms to use in which circumstances. These recommendations revise those from [15]; Couvreur's algorithm which was recommended there, is shown to have weak performance; however, the modification mentioned above amends it. Moreover, our modification of Schwoon/Esparza improves the previous best nested-DFS algorithm.

We proceed as follows: Section 2 establishes preliminaries, Sections 3 and 4 present nested-DFS and SCC-based algorithms, including our modifications. Section 5 details our experimental results and concludes.

2 Preliminaries

A *Büchi automaton* (BA) is a tuple $\mathcal{B} = (S, s_I, \mathrm{post}, A)$, where S is a finite set of *states*, $s_I \in S$ is the *initial state*, $\mathrm{post}\colon S \to 2^S$ is the *successor function*, and $A \subseteq S$ are the *accepting states*. A *path* of $\mathcal{B}$ is a sequence of states $s_1 \cdots s_m$ for some $m \geq 1$ such that $s_{i+1} \in \mathrm{post}(s_i)$ for all $1 \leq i < m$. If a path from s to t exists, we write $s \to^* t$. When $m > 1$, we write $s \to^+ t$, and if additionally $s = t$, we call the path a *loop*. A *run* of $\mathcal{B}$ is an infinite sequence $(s_i)_{i \geq 0}$ such that $s_0 = s_I$ and $s_{i+1} \in \mathrm{post}(s_i)$ for all $i \geq 0$. A run is called *accepting* if $s_i \in A$ for infinitely many different i. The *emptiness problem* is to determine whether no accepting run exists. If an accepting run exists, it is also called a *counterexample*. From now on, we assume a fixed Büchi automaton $\mathcal{B}$.

Note that we omit the usual input alphabet because we are just interested in emptiness checks. Moreover, the transition relation is given as a mapping from each state to its successors, which is suitable for on-the-fly algorithms.

A *strongly connected component* (SCC) of $\mathcal{B}$ is a subset $C \subset S$ such that for each pair $s, t \in C$, we have $s \to^* t$, and moreover, no other state can be added to C without violating this property. An SCC C is called *trivial* if $|C| = 1$ and for the singleton $s \in C$, $s \notin \mathrm{post}(s)$. The following two facts are well-known:

(1) A counterexample exists iff there exists some $s \in A$ such that $s_I \to^* s$ and $s \to^+ s$. This fact is exploited by nested-DFS algorithms.
(2) A counterexample exists iff there exists a non-trivial SCC C reachable from s_I such that $C \cap A \neq \emptyset$. This fact is exploited by SCC-based algorithms.

A Büchi automaton is called *weak* if each of its SCCs is either contained in A or in $S \setminus A$. This implies the following fact:

(3) Each loop in a weak BA is entirely contained in A or in $S \setminus A$.

A *generalized Büchi automaton* (GBA) is a tuple $\mathcal{G} = (S, s_I, \mathrm{post}, \mathcal{A})$, where S, s_I, and post are as before, and $\mathcal{A} = (A_1, \ldots, A_k)$ is a *set* of acceptance conditions, i.e. $A_j \subseteq S$ for all $j = 1, \ldots, k$. Paths and runs are defined as for normal Büchi automata; a run $(s_i)_{i \geq 0}$ of $\mathcal{G}$ is called *accepting* iff for each $j = 1, \ldots, k$ there exist infinitely many different i such that $s_i \in A_j$.

GBA are generally more concise than BA: a GBA with k acceptance conditions and n states can be transformed into a BA with nk states. There is no known nested-DFS algorithm that avoids this k-fold blowup for checking emptiness of a GBA, although Tauriainen's algorithm mitigates it [17]. Some SCC-based algorithms, however, can exploit the following fact:

(4) A counterexample exists in $\mathcal{G}$ iff there exists a non-trivial SCC C reachable from s_I such that $C \cap A_j \neq \emptyset$ for all $j = 1, \ldots, k$.

3 Nested depth-first search

Nested DFS was first proposed by Courcoubetis et al [2], and all other algorithms in this subgroup still follow the same pattern. There are two DFS iterations: the "blue" DFS is the main loop and marks every newly discovered state as blue. Upon backtracing from an accepting state s, it initiates a "red" DFS that tries to find a loop back to s, marking every encountered state as red. If a loop is found, a counterexample is reported, otherwise the blue DFS continues, but the established red markings remain. Thus, both blue and red DFS visit each state at most once each. Only two bits of auxiliary data are required per state.

This pattern of searching for accepting loops in post-order ensures that multiple red searches do not interfere; states in "deep" SCCs are coloured red first, and when a red DFS terminates, red states are guaranteed not to be part of any counterexample. While being memory-efficient and simple, this has two disadvantages. First, nested DFS prefers long counterexamples over shorter ones; secondly, the blue DFS never notices that a complete counterexample has already been explored and continues exploring potentially many more states than necessary before eventually noticing the counterexample during backtracking. Also, nested DFS computes the successors of many states twice.

Several improvements have been suggested in the past, e.g. the HPY algorithm [13], implemented in Spin, and the SE algorithm [15]. We present an improvement of SE, shown in Figure 1. A self-contained presentation and proof is provided in [7]; here, we just describe the differences w.r.t. SE.

The additions to SE are in lines 4 and from 12 to 15. These exploit the fact that red states cannot be part of any counterexample; therefore a state that has only red successors cannot be either. This avoids certain initiations of the red search. The improvement is similar in spirit to [8], but avoids some unnecessary invocations of post. Like in [2], only two bits per state are used. Our experiments shall show that it performs best among the known nested DFS algorithms.

Finally, we remark that for weak automata a much simpler algorithm suffices, as observed by Černá and Pelánek [18]. Exploiting Fact (3), one can simply omit the red search because all counterexamples are bound to be reported by line 9 in Figure 1. In that case, post is only invoked once per state.

```
procedure $new\_dfs$ ()
  call $dfs\_blue(s_I)$

procedure $dfs\_blue$ ($s$)
  $allred$ := true;
  $s.colour := cyan$;
  for all $t \in \mathrm{post}(s)$ do
    if $t.colour = cyan$
        $\wedge\ (s \in A \vee t \in A)$ then
      report cycle
    else if $t.colour = white$ then
      call $dfs\_blue$(t);
    if $t.colour \neq red$ then
      $allred$ := false;
  if $allred$ then
    $s.colour := red$
  else if $s \in A$ then
    call $dfs\_red$(s);
    $s.colour := red$
  else
    $s.colour := blue$

procedure $dfs\_red$ ($s$)
  for all $t \in \mathrm{post}(s)$ do
    if $t.colour = cyan$ then
      report cycle
    else if $t.colour = blue$ then
      $t.colour := red$;
      call $dfs\_red$(t)
```

Fig. 1. New Nested-DFS algorithm.

4 SCC-based algorithms

An efficient algorithm for determining SCCs that works on-the-fly was first proposed by Tarjan [16]. However, for model-checking purposes Tarjan's algorithm was deemed unsuitable because it used more memory than nested DFS while offering no advantages. More recent innovations by Geldenhuys/Valmari [10] and Couvreur [3] change the picture, however: their modifications allow SCC-based analysis to report a counterexample as soon as all its states and transitions were discovered, no matter in which order. In other words, if the order in which successors are explored by the DFS is fixed, both can find a counterexample in optimal time (w.r.t. to the exploration order).

Space constraints prevent us from presenting the algorithms in detail. However, we mention a few salient points. Tarjan places all newly discovered states onto a stack (henceforth called *Tarjan stack*) and numbers them in pre-order. Certain properties of the DFS ensure that at any time during the algorithm, states belonging to the same SCC are stored consecutively on the stack and therefore also numbered consecutively. The *root* of an SCC is the state explored first during DFS, having the lowest number and being deepest on the Tarjan stack. For each state s, Tarjan computes a so-called "lowlink" number, which is identical to the number of s iff s is a root, and less than that otherwise. An SCC is completely explored when backtracking from its root, and at that point it can be identified as a complete SCC and removed from the Tarjan stack.

Geldenhuys/Valmari (GV) exploit properties of lowlinks; they remember the number of the deepest accepting state on the current search path, say k, and when a state with lowlink $\leq\ k$ is found, a counterexample is reported. They also propose some memory savings that are of minor importance in our context.

Couvreur (C99) omits both Tarjan stack and lowlinks but introduces a *roots stack* that stores the roots of all partially explored SCCs on the current search path. When one finds a transition to a state with number k, properties of the

```
 1  **procedure** *couv* ()
 2    $count := 0$;
 3    $Roots := \emptyset$; $Active := \emptyset$;
 4    **call** *couv_dfs*($s_I$)
 5  **procedure** *couv_dfs*($s$):
 6    $count := count + 1$;
 7    $s.dfsnum := count$;
 8    $s.current :=$ **true**;
 9    **push**($Roots, (s, \mathcal{A}(s))$);
10    **push**($Active$,s);
11    **for all** $t \in \mathrm{post}(s)$ **do**
12      **if** $t.dfsnum = 0$ **then**
13        **call** *couv_dfs*($t$)
14      **else if** $t.current$ **then**
15        $B := \emptyset$;
16        **repeat**
17          $(u, C) :=$ **pop**($Roots$);
18          $B := B \cup C$;
19          **if** $B = K$ **then report cycle**
20        **until** $u.dfsnum \leq t.dfsnum$;
21        **push**($Roots, (u, B)$);
22    **if top**($Roots$) $= (s, ?)$ **then**
23      **pop**($Roots$);
24      **repeat**
25        u:=**pop**($Active$);
26        u.$current :=$ **false**
27      **until** $u = s$
```

Fig. 2. Amendment of Couvreur's algorithm.

numbering imply that no state with number larger than k can be a root, prompting their removal from the roots stack. This effectively merges some SCCs, and one checks whether the merger creates an SCC with the conditions from Fact (2).

Both algorithms report a counterexample after seeing the same states and transitions, provided they work with the same exploration order. However, it turns out that the removal of the Tarjan stack in C99, while more memory efficient, was a crucial oversight: when backtracking from a root, another DFS is necessary to mark these states as "removed". These extra post computations severely impede its performance. This makes GV superior to C99 in practice.

We propose to amend C99 by re-inserting the Tarjan stack.[3] This amendment makes it competitive with GV while using slightly less memory; more crucially, C99 can deal directly with GBAs, which GV cannot. Since GBAs tend to be smaller than BAs for the same LTL formula, the amended algorithm can hope to explore fewer states and be faster.

The amended algorithm, working with GBAs, is shown in Figure 2. A more detailed presentation and a proof are given in [7]. Note that in C99 accceptance conditions are annotated on the transitions, whereas here we place them on the states, which is only a minor difference. Figure 2 assumes k acceptance sets, denoting $\mathcal{A}(s) := \{\, j \mid s \in A_j \,\}$ and $K := \{1, \ldots, k\}$. Note that if k is "small", the union operation in line 18 can be implemented with bit parallelism.

5 Experiments

We implemented a framework for testing and comparing the actual performance of all the known Büchi emptiness algorithms. For practical relevance, the best framework for such an implementation would have been Spin. However, Spin

[3] The problem with C99 was first hinted at in [15]. After creating this improvement independently, we learned that similar changes were already proposed in [4] and [11].

turned out too difficult to modify for this purpose. Instead, we based our testbed on NIPS [19], a reverse-engineered Promela engine. Essentially, NIPS allows to process a Promela model, provides the initial state descriptor and a function for computing its successors. It is thus ideally suited for testing on-the-fly algorithms, and we believe that the conditions are as close to Spin as possible.

We used 266 test cases from the BEEM database [14], including many different algorithms, e.g., the Sliding Window protocol, Lamport's Bakery algorithm, Leader Election, and many others, together with various LTL properties.

Among the algorithms tested and implemented were HPY [13], GV [10], C99 [3], SE [15], and the amended algorithms presented in Sections 3 and 4, henceforth called AND and ASCC. For weak automata, we report on simple DFS (SD, see Section 3). We also implemented and tested other algorithms, notably those from [2] and [8]. However, these were always dominated by others, and we omit them in the following. Naturally, our concrete running times and memory consumptions are subject to certain implementation-specific issues. Nonetheless, we believe that the tendencies exhibited by our experiments are transferrable.

In the following, we give a summary of our results. A more detailed description of our framework, the benchmarks, and the experimental results is given in [6]; here, we just summarize the most important findings.

We first found that, in the context of exact model checking, the differences in auxiliary memory usage was basically irrelevant. Certainly, the auxiliary memory used by the various algorithms ranged from 2 bits to 12 bytes, a comparatively large difference. However, this was dwarved by the memory consumption of state descriptors, which ranged from 20 to 380 bytes, averaging at 130.

The only practical difference therefore was in the running time. Here, we found that, no matter which auxiliary data structures were employed, the running time was practically proportional to the number of post invocations (more precisely: the number of individual successor states generated by post), by far the most costly operation. In retrospect, these two observations may seem obvious; however, we find that they were consistently under-represented in previous papers, therefore it is worth re-emphasizing their relevance. The two main factors contributing to the running time were fast counterexample detection and whether an algorithm had to compute each transition at most once or twice.

Discussing individual test cases would not be very meaningful: for instance, the early-detection properties of some algorithms can cause arbitrarily large differences. Instead, we exhibit certain structural properties that occurred in many test cases and caused those differences. We first discuss algorithms working on "normal" Büchi automata, followed by a discussion of ASCC with GBAs.

First, we observe that most test cases constitute weak Büchi automata. Note that the intersection BA is weak if the BA arising from the formula is weak. Černá and Pelánek [18] estimate the proportion of weak formulae in practice to 90–95 %; indeed, we found that only 8 % of our test cases were non-weak. For weak test cases, five out of six tested algorithms (GV, C99, SE, AND, SD) detect counterexamples with minimal exploration. The three main structural effects causing performance differences (which may overlap) were as follows:

- In 86 test cases, we observed many trivial SCCs consisting of one accepting state. A typical example is the LTL property GFp, which (when negated) yields a weak automaton with a looping accepting state. Then, any non-looping part of the system necessarily yields such trivial SCCs. In these cases, GV and SD dominate, sometimes with a factor of two, whereas C99, SE, and HPY fall behind because they explore the accepting trivial SCCs twice. In our test cases, the AND algorithm had the same performance as GV and SD, although this is not guaranteed in general.
- In 98 cases, we observed non-accepting SCCs not preceded by accepting SCCs. In this case, C99 falls behind all the others.
- HPY reports counterexamples only during the red DFS, whereas SE and AND discovers some during the blue DFS. This accounts for 101 test cases in which HPY fared worst, whereas all others showed the same performance.

Non-weak automata also had these effects, affecting 18, 17, and 7 out of 21 test cases. In 7 cases, GV and C99 found counterexamples more quickly than the others, being faster by a factor of up to six. Since we used the same exploration order in all algorithms, these results are directly comparable.

algorithm	run-time
ASCC	67.0 %
GV	69.2 %
AND	69.7 %
SE	96.3 %
HPY	100.0 %
C99	128.3 %

Fig. 3. Performances

We then tested the ASCC algorithm with GBA, generated by the LTL2BA tool [9]. Most formulae yielded GBA with only one acceptance condition, meaning that the GBA had the same size as the corresponding BA. Notice that the running times of GBA with multiple conditions are not directly comparable with those of the corresponding BA. This is because using a different automaton changes the order of exploration, therefore in some "lucky" cases the BA-based algorithms may still find a counterexample more quickly.

The running times summed up over all 266 test cases are given in Figure 3, expressed as percentages of each other. Additionally, SD had the same performance as GV for the weak cases. Note that every set of benchmarks would lead to the same order among the algorithms because it reflects their different qualitative properties (e.g., quick counterexample detection or number of post calls). The concrete numbers in Figure 3 tell their quantitative effect in what we believe to be a representative set of benchmarks. We draw the following conclusions:

- Because of the dominance of weak test cases and GBAs with only one acceptance condition, the sum of running times yields small differences; only SE, HPY, and C99 clearly fall behind. The performance differences in the comparatively few other cases is very pronounced however.
- Overall, ASCC is the best algorithm if GBAs can be used. Due to the technical reasons explained above, it did not perform best in all examples.
- Among the BA-based algorithms, GV is the best for general formulae; it is never outperformed on any test case by any other BA-based algorithm. ASCC performs equally well when used with simple BAs.
- For weak formulae, SD is the best algorithm for bitstate hashing.

– For general formulae, AND is the best algorithm for bitstate hashing, improving the previous best algorithm for this setting (SE) by 28 %.
– There remains no reason to use either SE, HPY, or C99.

Acknowledgements: The authors would like to thank Michael Weber for creating and helping us use the NIPS framework.

References

1. Jiří Barnat, Luboš Brim, and Petr Ročkai. DiVinE multi-core - a parallel LTL model-checker. In *Proc. ATVA*, LNCS 5311, pages 234–239, 2008.
2. Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.
3. Jean-Michel Couvreur. On-the-fly verification of linear temporal logic. In *Proc. Formal Methods*, LNCS 1708, pages 253–271, 1999.
4. Jean-Michel Couvreur, Alexandre Duret-Lutz, and Denis Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In *Proc. SPIN*, LNCS 3639, pages 169–184, 2005.
5. Kathi Fisler, Ranan Fraer, Gila Kamhi, Moshe Y. Vardi, and Zijiang Yang. Is there a best symbolic cycle-detection algorithm? In *Proc. TACAS*, LNCS 2031, pages 420–434, 2001.
6. Andreas Gaiser. Vergleich von Algorithmen für den Leerheitstest von Büchiautomaten. Studienarbeit, Universität Stuttgart, 2007. In German.
7. Andreas Gaiser and Stefan Schwoon. Comparison of algorithms for checking emptiness on Büchi automata. Technical report, arxiv.org (`arXiv:0910.3766`), 2009.
8. Paul Gastin, Pierre Moro, and Marc Zeitoun. Minimization of counterexamples in SPIN. In *Proc. 11th SPIN Workshop*, LNCS 2989, pages 92–108, 2004.
9. Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *Proc. CAV*, LNCS 2102, pages 53–65, 2001.
10. Jaco Geldenhuys and Antti Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In *Proc. TACAS*, LNCS 2988, pages 205–219, 2004.
11. Jaco Geldenhuys and Antti Valmari. More efficient on-the-fly LTL verification with Tarjan's algorithm. *Theoretical Computer Science*, 345(1):60–82, 2005.
12. Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.
13. Gerard J. Holzmann, Doron A. Peled, and Mihalis Yannakakis. On nested depth first search. In *Proc. 2nd SPIN Workshop*, pages 23–32, 1996.
14. Radek Pelánek. Beem: Benchmarks for explicit model checkers. In *Proc. SPIN*, LNCS 4595, pages 263–267, 2007.
15. Stefan Schwoon and Javier Esparza. A note on on-the-fly verification algorithms. In *Proc. TACAS*, LNCS 3440, pages 174–190, 2005.
16. Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
17. Heikki Tauriainen. Nested emptiness search for generalized Büchi automata. *Fundamenta Informaticae*, 70(1–2):127–154, 2006.
18. Ivana Černá and Radek Pelánek. Relating hierarchy of linear temporal properties to model checking. In *Proc. MFCS*, LNCS 2747, pages 318–327, 2003.
19. Michael Weber. An embeddable virtual machine for state space generation. In *Proc. SPIN*, LNCS 4595, pages 168–186, 2007.

Derivation in Scattered Context Grammar via Lazy Function Evaluation

Ota Jirák[1] and Dušan Kolář[2]

[1] FIT BUT, Brno, Czech Republic,
ijirak@fit.vutbr.cz,
WWW home page: http://www.fit.vutbr.cz/~ijirak/
[2] FIT BUT, Brno, Czech Republic,
kolar@fit.vutbr.cz,
WWW home page: http://www.fit.vutbr.cz/~kolar/

Abstract. This paper discusses scattered context grammars (SCG) and considers the application of scattered context grammar production rules. We use function that represents single derivation step over the given sentential form. Moreover, we define this function in such a way, so that it represents the delayed execution of scattered context grammar production rules using the same principles as a lazy evaluation in functional programming. Finally, we prove equivalence of the usual and the delayed execution of SCG production rules.

1 Introduction

Family of languages that is described by scattered context grammars is very important due to their generative power. This paper discusses usage of functions over sentential forms to simulate derivation steps. Function representing delayed execution of scattered context grammar rules is introduced. Next, we discuss lazy evaluation of this recursively defined function.

The main goal of this article is to prove that this function is equivalent to commonly known derivation step.

The proof is divided into several parts:

– we use example to demonstrate that sentential form completely processed and the same sentential form partially processed are equivalent when processed by the delayed execution function,
– we demonstrate that introduced function can handle any SCG rules on any sentential form,
– we demonstrate that application of nested calling of delayed function is equivalent with nested calling of regular derivation function,
– we demonstrate that application of nested delayed derivation, lazy evaluated, is equivalent to nested regular derivation function.

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2351

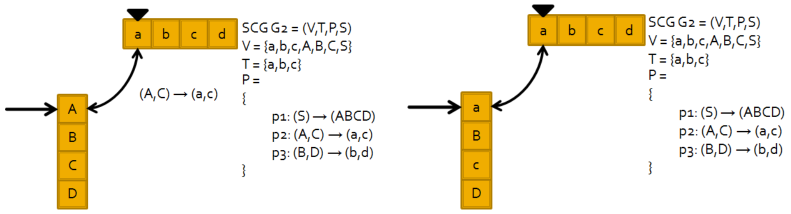
2 Motivation

We have several principles for implementation of compilers for SCG: deep pushdown [8] (with a certain limitation), and regulated pushdown automata [4–6] (RPDA).

The first approach uses nonterminal expansion not only on the pushdown top, but even deeper. Implementation of this pushdown is inefficient (linked list).

The RPDA usually uses auxiliary pushdown to restore the main pushdown. This data shuffling from one pushdown to another is also inefficient.

We are interested in deterministic compilers. Thus, we have to use leftmost derivation principles and LL SCGs to make the parser work deterministically [5, 6]. We need to work only with the pushdown top.



(a) Before Applying the Rule (b) After Applying the Rule

Fig. 1. Normal Derivation Using RPDA

Productions of SCG are defined as an n-tupple of CFG productions (see Section 3 - Preliminaries and Definitions). This is origin of basic idea. We would like to use parsing principles from CFG parsers to parse context-free parts of SCG productions in the right time.

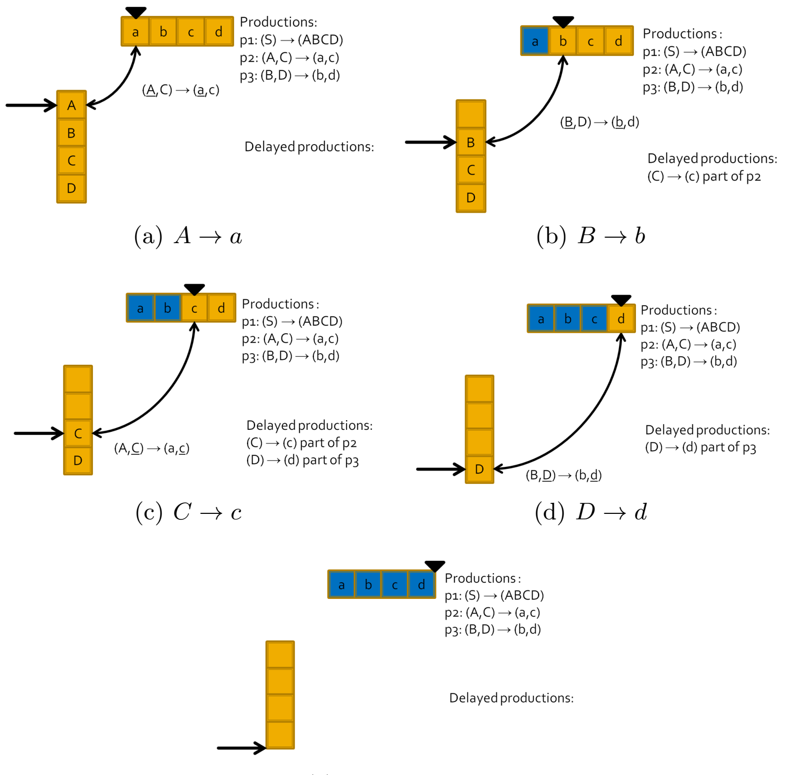
We rely on principles of deterministic context-free parsers. We use leftmost derivation. We have to use some kind of LL/LR grammars to choose productions in a deterministic way.

We use one CFG production of particular SCG production. The others are delayed and used in the right time. The unprocessed part of sentential form is marked to be processed later with this delayed part of SCG production.

We can see difference between regular derivation and delayed derivation on Figures 1 and 2. Applying one production in regular way means rewrite several nonterminals in one step. One regular derivation step is shown on Figure 1(a) and 1(b). Nonterminals A and C are rewritten in one derivation step.

Example of delayed execution is shown on Figure 2. Production p2 should be used in Figure 2(a). We apply $A \to a$ and we delay $C \to c$. Then, pop is applied on symbol a on the top of pushdown and a under the reading head.

Production p3 should be used in Figure 2(b). We apply $B \to b$ and we delay $D \to d$. Then, pop is applied on symbol b on the top of pushdown and b under the reading head.



(a) $A \to a$ (b) $B \to b$

(c) $C \to c$ (d) $D \to d$

(e) Acceptation

Fig. 2. Delayed Derivation

In Figure 2(c), there is no other option than to use delayed production $C \to c$. If there are more delayed productions that could be used then we use the oldest one. So we applied delayed production $C \to c$ and then removed it from delayed production list. Then, pop is applied on symbol c on the top of pushdown and c under the reading head.

In Figure 2(d), there we used delayed production $D \to d$ and then removed it from the delayed production list. Then, pop is applied on symbol d on the top of pushdown and d under the reading head.

In Figure 2(e), there we can see accepted sentential form by given scattered context grammar. Sentence is accepted with empty pushdown and zero delayed productions.

3 Preliminaries and Definitions

It is expected that a reader is familiar with formal language theory [7].

Let V^* be a free monoid over alphabet V, $w \in V^*$, w is called string of symbols from V, $|w|$ denotes the length of w. Let ε be an empty string, $|\varepsilon| = 0$.

A context-free grammar (CFG, see [7]) is a quadruple $G = (V, T, P, S)$, where V is a finite set of symbols, $T \subset V$ is a terminal alphabet, $S \in V \backslash T$ is the starting nonterminal, and P is a finite set of rules of the form $A \to w$, where $A \in V \backslash T$ and $w \in V^*$.

Now, we introduce definition of sentential form indexing. Besides common concepts from the formal languages, we define for a string, X, X[n] and X[n:] to denote one symbol from string and a substring of the string.

Definition 1. *Let $X = a_1 a_2 \ldots a_n, a_i \in V, i \in \{1, \ldots, n\}, n \in \mathbb{N}$.*

$$\begin{aligned} X[k] &= a_k, k \in \mathbb{N}, 1 \leq k \leq n, \\ X[k:] &= a_k \ldots a_n, k \in \mathbb{N}, 1 \leq k \leq n, \\ X[k] = X[k:] &= \varepsilon, k \in \mathbb{N}, k > n. \end{aligned}$$

A scattered context grammar (see [1]) is a quadruple $G = (V, T, P, S)$, where V is a finite set of symbols, $T \subset V$ is a terminal alphabet, $S \in V \backslash T$ is the starting nonterminal, and P is a finite set of production rules of the form $(A_1, A_2, \ldots, A_n) \to (w_1, w_2, \ldots, w_n)$, for some $n \geq 1$, where $A_i \in V \backslash T$ and $w_i \in V^*$. Let $u = x_1 A_1 x_2 A_2 \ldots x_n A_n x_{n+1}$, $v = x_1 w_1 x_2 w_2 \ldots x_n w_n x_{n+1}$, $x_i \in V^*, A_i \in V \backslash T, 1 \leq i \leq n$, for some $n \geq 1$. $u \Rightarrow v, x_1 A_1 x_2 A_2 \ldots x_n A_n x_{n+1} \Rightarrow x_1 w_1 x_2 w_2 \ldots x_n w_n x_{n+1}$ is a derivation. If $x_1 \in T^*, x_i \in (V \backslash \{A_i\})^*$, it is a leftmost derivation. Let $u_0, \ldots, u_n \in V^*, p_1, \ldots, p_n \in P$, $u_0 \Rightarrow_1 u_1 \Rightarrow_2 \cdots \Rightarrow_i u_i \Rightarrow \cdots \Rightarrow_n u_n$ is a sequence of leftmost derivations. Number of the derivation step, i, is a position in the sequence of derivations. $\Rightarrow_i$ represents usage of p_i — production of the i-th derivation step.

Definition 2. *Let $X = x_1 A_1 x_2 A_2 \ldots x_n A_n x_{n+1}$ be a sentential form, $A_i \in V \backslash T, x_i \in (V \backslash \{A_i\})^*, x_{n+1} \in V^*$, for some $n \geq 1$, j is a number of derivation step and $p_j : (A_1, A_2, \ldots, A_n) \to (w_1, w_2, \ldots, w_n) \in P$ is an SCG rule used in the j-th derivation step. Function $h_j(X)$ stands for leftmost application of the SCG rule used in the j-th derivation step that is:*

$$\begin{aligned} h_j(X) &= h_j(x_1 A_1 x_2 A_2 \ldots x_n A_n x_{n+1}) \\ &= x_1 w_1 x_2 w_2 \ldots x_n w_n x_{n+1} \end{aligned} \tag{1}$$

Note 1. We say, this is a regular derivation step.

Definition 3. *Let $X = x_1 A_1 x_2 A_2 \ldots x_n A_n x_{n+1}$ be a sentential form, $A_i \in V \backslash T, x_i \in (V \backslash \{A_i\})^*, x_{n+1} \in V^*$, for some $n \geq 1$, j is a number of derivation step and $p_j : (A_1, A_2, \ldots, A_n) \to (w_1, w_2, \ldots, w_n) \in P$ is an SCG rule used in the j-th derivation step, $m \in \{1, \ldots, n+1\}$.*

$$g_j(m, X) = \begin{cases} X[1] g_j(m, X[2:]) & \text{for } m \leq n, X[1] \neq A_m, \\ w_m g_j(m+1, X[2:]) & \text{for } m \leq n, X[1] = A_m, \\ \varepsilon & \text{for } m > n, |X| = 0, \\ X[1] g_j(m, X[2:]) & \text{for } m > n, |X| > 1, \\ X[1] & \text{for } m > n, |X| = 1. \end{cases} \tag{2}$$

Note 2. We say, this is a delayed derivation step.

Let x be some sentential form. $x^{'}$ is sentential form processed by function g or h — added apostroph to x.

Let n_b^x is an index into SCG rule, which is used in the b-th derivation step. Symbol x is a counter of symbols processed by function, which is used in b-th derivation step.

Example 1. Now, we demonstrate that sentential form completely processed and the same sentential form partially processed are equivalent when processed by the delayed execution function.
$g(n_1^0, x_1x_2 \dots x_i x_{i+1} \dots x_k) = x_1^{'} \dots x_i^{'} g(n_1^i, x_{i+1} \dots x_k)$, $i \in \{1, \dots, k\}$, $x_i \in V$ for some $k \in \mathbb{N}$

$$\begin{aligned}
g(n_1^0, x_1x_2 \dots x_k) &= \Big|_{\text{based on definition 3}} x_1^{'} g(n_1^1, x_2 \dots x_k) && (3)\\
&= \Big|_{\text{based on definition 3}} x_1^{'} x_2^{'} g(n_1^2, x_3 \dots x_k) && (4)\\
&\dots \\
&= \Big|_{\text{based on definition 3}} x_1^{'} x_2{'} \dots x_{k-1}^{'} g(n_1^{k-1}, x_k) && (5)\\
&= \Big|_{\text{based on definition 3}} x_1^{'} x_2^{'} \dots x_k^{'} && (6)
\end{aligned}$$

$\Rightarrow g(n_1^0, x_1x_2 \dots x_i x_{i+1} \dots x_k) = x_1^{'} \dots x_i^{'} g(n_1^i, x_{i+1} \dots x_k)$ for some $i \in \{1, \dots, k\}$

Lazy evaluation [9] based on call-by-need strategy is a scheduling policy that does not evaluate an expression (or invoke a procedure) until the results of the evaluation are needed. Lazy evaluation may avoid some unnecessary work. It may allows a computation to terminate in some situations that otherwise would not.

Lazy evaluation is often used in functional and logic programming, e.g. Haskell[2].

Lazy evaluation of delayed derivation is application of delayed derivation in a lazy way. It means that the leftmost symbols of sentential form are processed by several derivation steps while the rest of the sentential form is still unchanged. In other words, we make recursive step (see definition 2) only on the leftmost outermost symbol being unprocessed by particular function for delayed derivation.

4 Basic Idea

Each scattered context grammar derivation step can be described as an application of some function over given sentential form. In our case, this function represents leftmost application of an SCG rule.

We go through the sentential form and test each symbol. If the tested symbol is a particular nonterminal from the appropriate (context-free) part of the SCG rule we replace it with the right-hand side of the part of the SCG rule. And so on until the whole string is processed.

5 Results

At first, we show that delayed function can process sentential forms of any length and any number of context-free parts of an SCG rule.

Lemma 1. *$g(1, uA_1\alpha_1 \dots \alpha_k A_{k+1}\alpha_{k+1}) = h(uA_1\alpha_1 \dots \alpha_k A_{k+1}\alpha_{k+1}), i \in \{1, \dots, k\}$ for some $k \in \mathbb{N}, u \in T^*, x_i \in (V\backslash\{A_{i+1}\})^*, (A_1, \dots, A_{k+1}) \rightarrow (\beta_1, \dots, \beta_{k+1}) \in P$*

Proof. k — number of parts of SCG rule.
Basis. $g(1, uA_1\alpha_1) = h(uA_1\alpha_1), k = 1$

$$\begin{aligned}
g(1, uA_1\alpha_1) &=\Big|_{\text{based on definition 3}} ug(1, A_1\alpha_1) && (7)\\
&=\Big|_{\text{based on definition 3}} u\beta_1 g(2, \alpha_1) && (8)\\
&=\Big|_{\text{based on definition 3}} u\beta_1\alpha_1 && (9)\\
&=\Big|_{\text{based on definition 2}} h(uA_1\alpha_1) && (10)
\end{aligned}$$

$\Rightarrow g(1, uA_1\alpha_1) = h(uA_1\alpha_1)$ □

$g(1, uA_1\alpha_1A_2\alpha_2) = h(uA_1\alpha_1A_2\alpha_2)$, k=2

$$\begin{aligned}
g(1, uA_1\alpha_1A_2\alpha_2) &=\Big|_{\text{based on definition 3}} ug(1, A_1\alpha_1A_2\alpha_2) && (11)\\
&=\Big|_{\text{based on definition 3}} u\beta_1 g(2, \alpha_1A_2\alpha_2) && (12)\\
&=\Big|_{\text{based on definition 3}} u\beta_1\alpha_1 g(2, A_2\alpha_2) && (13)\\
&=\Big|_{\text{based on definition 3}} u\beta_1\alpha_1\beta_2 g(3, \alpha_2) && (14)\\
&=\Big|_{\text{based on definition 3}} u\beta_1\alpha_1\beta_2\alpha_2 && (15)\\
&=\Big|_{\text{based on definition 2}} h(uA_1\alpha_1A_2\alpha_2) && (16)
\end{aligned}$$

$\Rightarrow g(1, uA_1\alpha_1A_2\alpha_2) = h(uA_1\alpha_1A_2\alpha_2)$ □

Induction Hypothesis. We suppose that the statement holds for all $k, 1 \leq k \leq n$, for some $n \geq 1$.

Induction Step. $g(1, uA_1\alpha_1 \dots A_{k+1}\alpha_{k+1}) = h(uA_1\alpha_1 \dots A_{k+1}\alpha_{k+1}), n = k+1$

$$\begin{aligned}
& g(1, uA_1\alpha_1 \dots A_{k+1}\alpha_{k+1}) && (17)\\
&=\Big|_{\gamma=\alpha_k A_{k+1}\alpha_{k+1}} g(1, uA_1\alpha_1 \dots A_{k-1}\alpha_{k-1}A_k\gamma) && (18)\\
&=\Big|_{\text{based on induction hypothesis and equation 14}} u\beta_1\alpha_1 \dots \beta_{k-1}\alpha_{k-1}\beta_k g(k+1, \gamma) && (19)\\
&=\Big|_{\gamma=\alpha_k A_{k+1}\alpha_{k+1}} u\beta_1\alpha_1 \dots \beta_{k-1}\alpha_{k-1}\beta_k g(k+1, \alpha_k A_{k+1}\alpha_{k+1}) && (20)\\
&=\Big|_{\text{based on definition 3}} u\beta_1\alpha_1 \dots \beta_{k-1}\alpha_{k-1}\beta_k\alpha_k g(k+1, A_{k+1}\alpha_{k+1}) && (21)\\
&=\Big|_{\text{based on definition 3}} u\beta_1\alpha_1 \dots \beta_{k-1}\alpha_{k-1}\beta_k\alpha_k\beta_{k+1} g(k+2, \alpha_{k+1}) && (22)\\
&=\Big|_{\text{based on definition 3}} u\beta_1\alpha_1 \dots \beta_{k-1}\alpha_{k-1}\beta_k\alpha_k\beta_{k+1}\alpha_{k+1} && (23)\\
&=\Big|_{\text{based on induction hypothesis and definition 2}} h(uA_1\alpha_1 \dots \alpha_{k-1}A_k\alpha_k A_{k+1}\alpha_{k+1}) && (24)
\end{aligned}$$

Therefore, $g(1, uA_1\alpha_1 \dots A_{k+1}\alpha_{k+1}) = h(uA_1\alpha_1 \dots A_{k+1}\alpha_{k+1})$, so the lemma holds.

Next, we show that we can use any number of rules.

Lemma 2. *$g_k(1, \dots g_2(1, g_1(1, \alpha))) = h_k(\dots h_2(h_1(\alpha))), \alpha \in V^*$ for some $k \in \mathbb{N}$*

Proof. k — number of nested functions.
Basis. $g_1(1, \alpha) = h_1(\alpha), k = 1$. Proof in Lemma 1.
$g_2(1, g_1(1, \alpha)) = h_2(h_1(\alpha)), k = 2$

$$\begin{aligned}
g_2(1, g_1(1, \alpha)) &=\Big|_{\text{apply lemma1 on } g_1} g_2(1, h_1(\alpha)) && (25)\\
&=\Big|_{\text{apply lemma1 on } g_2} h_2(h_1(\alpha)) && (26)
\end{aligned}$$

$\Rightarrow g_2(1, g_1(1, \alpha)) = h_2(h_1(\alpha))$ □
Induction Hypothesis. We suppose that the statement holds for all $k, 1 \leq k \leq n$ for some $n \geq 1$.
Induction Step. $g_{k+1}(1, g_k(1, \dots g_1(1, \alpha))) = h_{k+1}(h_k(\dots h_1(\alpha)))$

$$\begin{aligned}
g_{k+1}(1, g_k(1, \dots g_1(1, \alpha))) &=\Big|_{\text{based on induction and } \beta=h_k(h_{k-1}\dots h_1(\alpha))} g_{k+1}(1, \beta) && (27)\\
&=\Big|_{\text{apply lemma1 on } h_{k+1}} h_{k+1}(\beta) && (28)\\
&=\Big|_{\text{based on induction and } \beta=h_k(h_{k-1}\dots h_1(\alpha))} h_{k+1}(h_k(\dots g_1(\alpha)) && (29)
\end{aligned}$$

Therefore, $g_{k+1}(1, g_k(1, \dots g_1(1, \alpha))) = h_{k+1}(h_k(\dots h_1(\alpha_1)))$, so the lemma holds.

The following Lemma 3 shows that delayed derivation steps (lazy evaluated) return the same values as regular derivation steps.

Lemma 3. *Lazy evaluation of* $g_{k+1}(1, \dots g_1(1, \omega_1 \dots \omega_m)) = h_{k+1}(\dots h_1(\omega_1 \dots \omega_m))$, $\omega_i \in V, i \in \{1, \dots, m\}$ *for some* $m \geq 1$ *and* $k \geq 1$.

Proof. Basis. For $k = 1$ holds from definition 3. For $k = 2$, lazy evaluation of $g_2(n_2^0, g_1(n_1^0, \omega_1 \dots \omega_m)) = h_2(h_1(\omega_1 \dots \omega_m)), n_1^0 = n_2^0 = 1$

$$g_2(n_2^0, g_1(n_1^0, \omega_1 \dots \omega_m)) \tag{30}$$

$$= \Big|_{\text{apply one step from definition3 on } g_1} \quad g_2(n_2^0, \omega_1^{'} g_1(n_1^1, \omega_2 \dots \omega_m)) \tag{31}$$

$$= \Big|_{\text{apply one step from definition3 on } g_2} \quad \omega_1^{''} g_2(n_2^1, g_1(n_1^1, \omega_2 \dots \omega_m)) \tag{32}$$

$\dots$

$$= \Big| \quad \omega_1^{''} \dots \omega_{m-1}^{''} g_2(n_2^{m-1}, g_1(n_1^{m-1}, \omega_m)) \tag{33}$$

$$= \Big|_{\text{apply one step from definition3 on } g_1} \quad \omega_1^{''} \dots \omega_{m-1}^{''} g_2(n_2^{m-1}, \omega_m^{'}) \tag{34}$$

$$= \Big|_{\text{apply one step from definition3 on } g_2} \quad \omega_1^{''} \dots \omega_m^{''} \tag{35}$$

$$= \Big|_{\text{based on definition 2}} \quad h_2(h_1(\omega_1 \dots \omega_m)) \tag{36}$$

$\Rightarrow$ lazy evaluation of $g_2(n_2^0, g_1(n_1^0, \omega_1 \dots \omega_k)) = h_2(h_1(\omega_1 \dots \omega_k))$ □

Induction Hypothesis. Suppose that the statement holds for all $k, 1 \leq k \leq n$, for some $n \in \mathbb{N}$.

Induction Step. $g_{k+1}(n_{k+1}^0, \dots g_1(n_1^0, \omega_1 \dots \omega_m)) = h_{k+1}(\dots h_1(\omega_1 \dots \omega_m)), \omega_i \in V, i \in \{1, \dots, m\}$ for some $m \in \mathbb{N}$ and $n_j^0 = 1, j \in \{1, \dots, k+1\}$. We can say without loss of generality that it returns one processed symbol in each step. To simplify the proof, we write $g_k(n_k^0, \omega_1 \dots \omega_m)$ instead of $g_k(n_k^0, \dots g_1(n_1^0, \omega_1 \dots \omega_m))$.

$$g_{k+1}(n_{k+1}^0, g_k(n_k^0, \omega_1\omega_2 \dots \omega_m)) \tag{37}$$

$$= \Big|_{\text{apply one step from definition3 on } g_k} \quad g_{k+1}(n_{k+1}^0, \omega_1^{'} g_k(n_k^1, \omega_2 \dots \omega_m)) \tag{38}$$

$$= \Big|_{\text{apply one step from definition3 on } g_{k+1}} \quad \omega_1^{''} g_{k+1}(n_{k+1}^1, g_k(n_k^1, \omega_2 \dots \omega_m)) \tag{39}$$

$\dots$

$$= \omega_1^{''} \dots \omega_{m-1}^{''} g_{k+1}(n_{k+1}^{m-1}, g_j(n_j^{m-1}, \omega_m)) \tag{40}$$

$$= \Big|_{\text{apply one step from definition3 on } g_k} \quad \omega_1^{''} \dots \omega_{m-1}^{''} g_{k+1}(n_{k+1}^{m-1}, \omega_m) \tag{41}$$

$$= \Big|_{\text{apply one step from definition3 on } g_{k+1}} \quad \omega_1^{''} \dots \omega_m^{''} \tag{42}$$

$$= \Big|_{\text{based on definition 2}} \quad h_{k+1}(h_k(\omega_1 \dots \omega_m)) \tag{43}$$

Therefore, lazy evaluation of $g_{k+1}(n_{k+1}^0, \dots g_1(n_1^0, \omega_1\omega_2 \dots \omega_m)) = h_{k+1}(\dots h_1(\omega_1 \dots \omega_m))$, so the lemma holds.

The following theorem and its proof, which represents the main result of this paper, demonstrates that delayed execution of SCG rules is equivalent with SCG derivation.

Theorem 1. *Lazy evaluation of* $g_m(1, g_{m-1}(1, \dots g_1(1, \omega_1 \dots \omega_j))) = h_m(h_{m-1}(\dots h_1(\omega_1 \dots \omega_j))) = w \equiv \omega_1 \dots \omega_j \Rightarrow^m w, \omega_i \in V, i \in \{1, \dots, j\}$ *for some* $j, m \in \mathbb{N}$.

Proof. $\alpha \in V^*$

1. Using function g or h is equivalent for sentential forms of any length and for any SCG rules. It has been proved in Lemma 1.
2. Using any number of functions, $g_n(\dots g_1(\alpha))$, is equivalent to $h_n(\dots h_1(\alpha))$, for any n. It has been proved in Lemma 2.
3. Lazy evaluated g returns the same result as h. It has been proved in Lemma 3.
4. $h_m(\dots h_1(\omega_1 \dots \omega_j)) = w \equiv \omega_1 \dots \omega_j \Rightarrow^m w$ holds by definition 2.

From 1, 2, 3, and 4 follows:
$g_m(1, g_{m-1}(1, \dots g_1(1, \omega_1 \dots \omega_j))) = h_m(h_{m-1}(\dots h_1(\omega_1 \dots \omega_j))) = w \equiv \omega_1 \dots \omega_j \Rightarrow^m w$. □

6 Conclusion

In this paper, we have shown usage of functions instead of derivation steps. Lazy evaluation of delayed execution of scattered context grammar rules has been presented.

The main result of this article is equivalence of lazy evaluated delayed executed function and the function representing regular leftmost derivation over a string. This approach allows us to work only with the pushdown top during compilation time.

7 Open Questions and Future Work

Scattered context grammar was introduced by Greibach and Hopcroft in 1969 (see [1]). Since these days, several implementation methods of compilers for scattered context grammars has been discovered [4–6, 8].

Next research will lead to study compilers that use delayed execution of SCG rules and to compare with compilers using regulated pushdown automata.

Intuitively, it should be faster, because we expand only topmost symbol on the stack. Basic principle of using delayed executed SCG rules in compilers is in [3].

Nevertheless, exploitation of lazy evaluation in implementation of an SCG parser traditional way [6] may be an option. That is why; we want to compare both approaches.

The work has been supported by Research Plan No. MSM 0021630528 - Security-Oriented Research in Information Technology and by the Czech Ministry of Education, Youth and Sports grant MŠMT 2C06008 "Virtual Laboratory of Microprocessor Technology Application".

References

1. Greibach, S., Hopcroft, J.: Scattered context grammars. J. Comput. Syst. Sci. 3, 233-247(1969)
2. Haskell, http://www.haskell.org/haskellwiki/Haskell/Lazy_evaluation, cited Sep. 2009
3. Jirák, O.: Delayed Execution of Scattered Context Grammar Rules, In: Proceedings of the 15th Conference and Competition STUDENT EEICT 2009 Volume 4, Brno, CZ, FIT VUT, 2009, p. 405-409, ISBN 978-80-214-3870-5
4. Kolář, D., Meduna, A.: Regulated Pushdown Automata, In: Acta Cybernetica, Vol. 2000, No. 4, US, p. 653–664, ISSN 0324-721X
5. Kolář, D.:Pushdown Automata: Another Extensions and Transformations, Brno, CZ, FIT BUT, 2005, p. 76
6. Kolář, D.: Scattered Context Grammar Parsers, In: Proceedings of the 14th International Congress of Cybernetics and Systems of WOSC, Wroclaw, PL,PWR WROC, 2008, p. 491–500, ISBN 978-83-7493-400-8
7. Meduna, A.: Automata and Languages: Theory and Applications. Springer-Verlag, London, 2000
8. Meduna, A.: Deep Pushdown Automata, In: Acta Informatica, Vol. 2006, No. 98, DE, p. 114–124, ISSN 0001-5903
9. University of Florida, http://www.cise.ufl.edu/research/ParallelPatterns/glossary.htm, cit. Sep 07 2009

Embedded Process Functional Language

Marek Běhálek[1] and Petr Šaloun[2]

[1] Department of Computer Science, Faculty of Electrical Engineering and Computer Science, VŠB Technical University of Ostrava, 17. listopadu 15, Ostrava, Czech Republic
marek.behalek@vsb.cz

[2] Department of Informatics and Computers, Faculty of Science, University of Ostrava, 30. dubna 22, Ostrava, Czech Republic
petr.saloun@osu.cz

Abstract. Embedded systems represent an important area of computer engineering. Demands on embedded applications are increasing. To address these issues, different agile methodologies are used in traditional desktop applications today. These agile methodologies often try to eliminate development risks in early design phases. Possible solution is to create a working model or a prototype of critical system parts. Then we can use this prototype in negotiation with customer and also to prove technological aspects of our solution. From this perspective functional languages are very attractive. They have excellent abstraction mechanism and they can be used as a tool producing a kind of executable design. In this paper we present our work on a domain specific functional language targeted to embedded systems — *Embedded process functional language* (e-$\mathcal{PFL}$). Created language works on a high level of abstraction and it uses other technologies (even other functional languages) created for embedded systems development on lower levels. It can be used like a modeling or a prototyping language in early development phases.

1 Introduction

Embedded systems represent an important area of computer engineering. Most of these systems are programmed in low level languages due to strict performance or memory constrains. On the other hand, demands on embedded applications are increasing. For example we want to decrease time to market, improve maintenance or make development process cheaper.

Different approaches are used to solve such problems. For example different agile methodologies [1] are more and more popular in the area of traditional desktop applications today. These agile methodologies try to eliminate development risks. Development risks are mainly related to: *business risks* (confusion in communication with customer, created product cannot be used in practice) and *technological risks* (inability to use developed application in practice due to technological issues). Possible solution that eliminates these risks is to develop working model or prototype of critical system parts in early development phases.

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2353

We can use this prototype for communication with customer and also to prove technological aspects of the solution.

To address these issues we need a better tool (or tools). From this point of view, functional languages are very attractive. They have several interesting properties [2]. Among others they have excellent abstraction mechanism (represented by functions composition and high-order functions) and that is why they can be used as a tool producing a kind of executable design. To conclude functional languages definitely have a place in specification, prototyping and simulation in early design phases.

In our work we are developing domain specific functional language targeted to embedded systems development. Developed language can be used like a modeling or a prototyping language in early development phases. Created language is called *Embedded process functional language* (e-$\mathcal{PFL}$).

2 Related Works

There are other tools able to model embedded systems. For example we can use *Unified Modeling Language* – UML. Or we can use simulation tools like *Simulink*[3]. We can address business risks using such tools. We are able to create concrete model and use it during negotiations with customer. On the other hand we may not be able to solve technological issues.

Embedded systems differ in many ways from common desktop applications [3]. For example a development platform is separated from a target platform, debugging is possible only with emulator, or there is no operating system present. Really try some application on a real device can be the only possibility to eliminate technological risks. Create language is still programming language. It can be straightforwardly transformed into a target code and used on concrete embedded systems.

Usage of functional paradigm of programming for development of embedded applications was suggested in [4]. Also there exist several different functional languages for implementation of embedded systems. Those languages cover wide range of abstraction levels (from hardware description to high-level application logic).

Erlang [5] is probably the most widely known language in this area. It has its origin in Ericsson, so unlike most declarative languages it did not come from academic development. It is a combination of logic and functional programming. Typical Erlang program consists of many light-weighted processes communicating trough asynchronous messages. It is primarily used in heavily concurrent and distributed applications. Other programming languages that can be used in this area are in fact languages designed mainly for reactive systems like: *Concurrent Haskell* [6] or *Eden* [7]. Other examples of languages for embedded systems development are: *Embedded Gofer* [4] (strongly typed purely functional language), *Lava* (Haskell library with ability to generate VHDL code) or *Lustre*

[3] Description available at: `http://www.mathworks.com/products/simulink/`

(synchronous data-flow language used for reactive systems and hardware description). *Hume* [8] represents different approach. Unlike presented languages (they often come from some general purpose language) it was specially developed for (especially real-time) embedded systems implementation. It tries to address performance issues, time and space constrains and controllability. The compiler can calculate for instance how much heap and stack each part of program will ever require at most.

Also different case studies comparing usage of functional languages for embedded systems development with traditional approaches were performed [9, 10].

3 Coordination Layer

Embedded systems are often described as a set of communicating functional units, no matter if multiple processing units are present on target machine or not [4]. Also our model is a set of communicating devices on the highest level.

Most of functional languages used for implementation of embedded applications take two-level approach to language design. Purely functional expression layer is often embedded into a coordination layer. Coordination layer describes communicating processes (or communicating functional units in context of embedded systems). Presented languages are often extended with side-effecting constructs to address issues on coordination layer. These constructs often enable creation of functional units and maintain their synchronization. There are two extremes. First can be represented by language *Embedded Gofer* [4]. It uses monads to encapsulate processes and language is extended with message passing primitives to maintain their communication. Such side-effecting constructs make reasoning about program properties hard or impossible. Another extreme is *Hume* [8]. Coordination layer in Hume must be strictly defined on a static level inside a source code. Communication is then implicit and we are able to compute necessary system properties at a compile time.

When creating new language, we must decide about language aspects on coordination layer. We use dynamic coordination mechanism in e-$\mathcal{PFL}$. There are language constructions that allow functional units creation and define functional units' connections. On the other hand, static model on coordination layer have certain advantages.

Primary purpose of e-$\mathcal{PFL}$ is to create working model of an embedded system in early design phases. We want to be able to use created language on real devices to eliminate technological risks. In our approach, we want to use other technologies even other functional languages on lower levels. Static model of the coordination simplifies usage of other technologies on lower levels.

As a compromise we use following model.

– Coordination layer is dynamic on language level in EPFL. This feature simplifies embedded systems modeling.
– Then we use partial evaluation and concrete system *configuration* is produced as a result of computation. This configuration represents static model

of future system on coordination layer. Created configuration is stored in XML file.

This approach has several advantages. For example to eliminate technological risks we can produce several concrete models of future system using the same program logic on higher level. Then we can produce different target codes for different architectures using these configurations.

4 Embedded Process Functional Language

In our work we are developing domain specific functional programming language (Embedded process functional language – e-$\mathcal{PFL}$) targeted to embedded systems development. Developed language can be used like a modeling or a prototyping language in early development phases. Presented language come from *Process Functional Language* – $\mathcal{PFL}$ [11] on language level. On implementation level it extends *Parallel Process Functional Language* [12] (that we have created before). Created language was introduced in [13].

Process Functional Language improves state representation by introducing variables while trying not to compromise its declarative nature. Usage of variables is bound to processes and is maintained by a compiler. That is why we are able to determine program parts manipulating with state at a compile time. Process application is the only place where we can access or update variables. The scope of variables is defined by the scope of processes that use them. Access and update of variable environment is uniform. We must use processes from the same scope. For instance programmer cannot directly access or update variables. Similar language constructions were used for e-$\mathcal{PFL}$.

Created language is shortly described in this section. It uses eager evaluation. Syntax and semantics come from $\mathcal{PFL}$ (it is close to pure functional subset of Haskell). This set of constructions was extended to support embedded systems development.

Embedded systems are often described as a set of functional units. Also in e-$\mathcal{PFL}$ embedded system is a set of communicating devices. These devices are modeled using data type `Device` for default module `Prelude`.

```
data Device = Process  EmbProcess
            | Fair     [Device]
            | Unfair   [Device]
```

Devices are strictly built from embedded processes. Embedded processes cannot be used directly. Embedded processes are described by data type `EmbProcess` (this data type was used in `Device` definition). Embedded processes encapsulate issues related to communication on coordination layer. Syntax of embedded processes is close to common functions. Embedded process definition is extended by variables (like in processes in $\mathcal{PFL}$). Variables are bounded to parameters and also to return value (it can be a tupple and then variables are bounded to every tupple element). Following example shows embedded process definition.

```
work :: a Integer -> b Integer ->(c Integer, d Integer)
work x y = (x, x+y)
```

Used variables represent communication channels. Embedded processes define operations with known input (variables bounded to parameters: `a` and `b` in our example) and output (variables bounded to return value: `c` and `d` in our example). Input or output of created devices is defined by used embedded processes. Each of variables can be used like an input maximally by one device and like an output also maximally by one device.

Devices can be started using native function `startDevice`.

```
startDevice :: Device -> EmbSystem -> [Annotation] -> ()
```

Each `Device` is an autonomous system working independently on other devices. Devices are working asynchronously[4]. When a device is started, it tries to execute embedded processes that it encapsulates. Processes compete for execution time within a single device. Only one process can be running at given time. When there is no process running new candidate for execution is selected according to available input and *fairness* (according to data constructors `Fair` and `Unfair`).

These devices can be divided into distant parts — embedded system components. Embedded system components are defined using data type `EmbSystem` (first argument is component name, the second is mediator). Data type `Mediator` defines concrete mediator. Both are from basic module `Prelude`.

```
data Mediator = Hume | MicroNET

data EmbSystem = EmbComponent [Character] Mediator | Emulator
```

In our approach we do not produce one target code. Programmer can divide embedded system into parts. Each of these parts is associated with exactly one mediator. Each component can use different mediator and thus can have different features and properties. For every component is generated target code with respect to used mediator. We have integrated two mediators into e-$\mathcal{PFL}$ now. Programmer can use *Hume* like an intermediate language or created run-time environment for *.NET Micro Framework*.

For example when Hume is used as a mediator then source codes in Hume are produced by e-$\mathcal{PFL}$ compiler. These codes can be then ported using tools created by Hume developers or we can use developed tool to compute runtime constrains.

Using this technique we are able to address technological risks of embedded systems development. Also we are able to implement even distributed embedded systems as a single application in e-$\mathcal{PFL}$ and divide it into distant parts during application porting. Using this technique we are able to integrate several different languages or platforms into one solution. This solution then composes different approaches to embedded systems development and benefits from their properties.

[4] A timer can be implemented using library functions if needed. Form this point of view the timer is a device periodically generating output necessary for other devices to continue.

Finally we are able to change certain device features when the device is started. We are able to add *annotations* to started devices. Annotations are related to created configuration. They are stored into generated configuration XML file and they affect produced target code. Using annotation we are able to change coordination, initial values or target code optimization level. Annotations are defined using data type `Annotation`. Programmer must not use annotations directly. He can use standard functions from module `Prelude`. For example function `rename`. Using this function programmer can *rename* inputs or outputs for a device to change devices connections.

```
data Attribute  = CAttribute  [Character] [Character]
data Annotation = CAnnotation [Character] [Attribute]

rename :: [Character] -> [Character] -> Annotation
rename x y = CAnnotation "rename"
             [(CAttribute "old" y),(CAttribute "new" x)]
```

Communication is implicit in e-$\mathcal{PFL}$. Related issues are solved during a configuration run. Each of variables used in started devices represents a communication channel. Type of this communication channel is known at a compile time and we are able to compute all necessary information related to usage of these channels (for example initial values or underlying device architecture). Potential communication issues simplify that there is maximally one input device and one output device to each of these channels and each of them can hold up to one value only. Main issues are thus related to data synchronization. A sort of default communication is computed during the configuration run and computed information is stored in a resulted configuration. Programmer can modify this configuration in the future and thus he can control communication.

5 Example

This section shows simple application written in e-$\mathcal{PFL}$.

```
produce :: a Integer -> (a Integer, b Integer, c Integer)
produce x = (x+1, x, x)
work :: Device
work = Process produce

showB ::b Integer -> ()
showB x = writeLine (show x)
printer :: Device
printer = Process showB

annotation :: Annotation
annotation = rename "b" "c"

component1 :: EmbSystem
component1 = EmbComponent "worker_component" Hume
```

```
component2 :: EmbSystem
component2 = EmbComponent "printer_component" MicroNET

main= (startDevice work    component1 []) `bl`
      (startDevice printer component2 []) `bl`
      (startDevice printer component2 [annotation])
```

Listing 1.1: Simple example in e-$\mathcal{PFL}$

Previous example use process `bl`. This process comes from $\mathcal{PFL}$. It forms a sequence of process. Its functionality is similar to construction `do` from Haskell.

Example composes from two components (first is using Hume, second is using .NET Micro Framework). Fist component contains one device based on `Device` named `work`. Second component contains two devices based on `Device` named `printer`. Example also shows how default connections of devices can be changed by a programmer (using annotations).

6 Conclusion and Future Work

We are developing a domain specific language called Embedded Process Functional Language (e-$\mathcal{PFL}$) targeted to embedded systems development. Created language works on a high level of abstraction. It uses other technologies (even other functional languages) created for embedded systems development on lower levels. It can be used like a modeling or a prototyping language in early development phases.

The contribution is that we are able to eliminate development risks using e-$\mathcal{PFL}$. In e-$\mathcal{PFL}$ we are able to create working prototype of future system (or its critical parts). Then we can use this prototype in negotiation with customer to eliminate business risks. Applications created in e-$\mathcal{PFL}$ can be simulated using implemented simulator. Using partial evaluation we are able to extract static model (or models) of future system. This model can be for example visualized and we can use it in communication with customer. Still created e-$\mathcal{PFL}$ is a programming language and we can straightforwardly produce target codes. We are using other technologies on lower level (now we are using Hume and .NET Micro Framework) and we can benefit from their features. Produced codes can be directly used on real devices. Using this technique we can eliminate technological risks during the development process.

For practical experiments we have implemented e-$\mathcal{PFL}$ simulator using .NET platform and a distributing cross-platform compiler. This compiler use Hume and .NET Micro Framework on a lower levels. Also we have created GUI containing tool that simplifies configuration of the applications.

Proposed e-$\mathcal{PFL}$ is under active development now. We are considering other language constructs that may improve its capabilities. For example we are considering different language constructions changing devices connections. We are also improving implemented tools. For example compiler implements only basic language constructions now. Constructions like list generators (common in

Haskell) are not supported yet. Also we want to extend basic libraries. Another area is practical applications of presented ideas. We want to use e-$\mathcal{PFL}$ for development of real embedded systems.

On the other hand presented approaches and principles are actively developed and used mainly in academic circles. There is still a long way ahead to see if presented usage of functional paradigm can truly compete with current methodologies.

Work is partially supported by Czech-Slovak fund KONTAKT MEB 080878: Cooperation in area of design and implementation of language systems.

References

1. Highsmith, J., Fowler, M.: The agile manifesto. Software Development Magazine **9**(8) (2001) 29–30
2. Hughes, R.: Why functional programming matter. The Computer Journal **32**(2) (1989) 98–107
3. Vahid, F., Givargis, T.: Embedded System Design: A Unified Hardware/Software Introduction. John Wiley & Sons, Inc., New York, NY, USA (2001)
4. Wallace, M., Runciman, C.: Extending a functional programming system for embedded applications. Softw. Pract. Exper. **25**(1) (1995) 73–96
5. Armstrong, J.: A history of erlang. In: HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages, New York, NY, USA, ACM (2007) 6–1–6–26
6. Peyton Jones, S., Gordon, A., Finne, S.: Concurrent haskell. In: POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (1996) 295–308
7. Loogen, R., Ortega-mallén, Y., Peńamarí, R.: Parallel functional programming in eden. J. Funct. Program. **15**(3) (2005) 431–475
8. Hammond, K., Michaelson, G.: Hume: A domain-specific language for real-time embedded systems. In Pfenning, F., Smaragdakis, Y., eds.: Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings. Volume 2830 of Lecture Notes in Computer Science., Springer (2003) 37–56
9. Nyström, J.H., Trinder, P.W., King, D.J.: Evaluating distributed functional languages for telecommunications software. In: ERLANG '03: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang, New York, NY, USA, ACM (2003) 1–7
10. Specht, E., Redin, R.M., Carro, L., Lamb, L.d.C., Cota, E.F., Wagner, F.R.: Analysis of the use of declarative languages for enhanced embedded system software development. In: SBCCI '07: Proceedings of the 20th annual conference on Integrated circuits and systems design, New York, NY, USA, ACM (2007) 324–329
11. Kollár, J., Porubän, J., Václavík, P.: From eager pfl to lazy haskell. Computers and Artificial Intelligence **25**(1) (2006)
12. Běhálek, M., Šaloun, P.: Parallel process functional language. In: SOFSEM 2007: Theory and Practice of Computer Science, 33rd Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 20-26, Proceedings Volume II. (2007) 1–12
13. Běhálek, M., Šaloun, P.: Simulation of embedded applications implemented in embedded process functional language. In: First International Conference on Computational Intelligence, Modelling, and Simulation, Brno, Czech Republic, IEEE Computer Society (7-9 September 2009) 253–258

Exact Quantum Query Algorithm for Error Detection Code Verification

Alina Vasilieva

Faculty of Computing, University of Latvia
Raina bulv. 29, LV-1459, Riga, Latvia
alina.vasilieva@gmail.com

Abstract. Quantum algorithms can be analyzed in a query model to compute Boolean functions. Function input is provided in a black box, and the aim is to compute the function value using as few queries to the black box as possible. A repetition code is an error detection scheme that repeats each bit of the original message r times. After a message with redundant bits is transmitted via a communication channel, it must be verified. If the received message consists of r-size blocks of equal bits, the conclusion is that there were no errors. The verification procedure can be interpreted as an application of a query algorithm, where input is a message to be checked. Classically, for N-bit message, values of all N variables must be queried. We demonstrate an exact quantum algorithm that uses only $N/2$ queries. [1]

1 Introduction

Quantum computing is an exciting alternative way of computation, which is based on the laws of quantum mechanics. This branch of computer science is developing rapidly; various computational models exist, and this is a study of one of them.

Let $f(x_1, x_2, ..., x_N) : \{0,1\}^N \rightarrow \{0,1\}$ be a Boolean function. We consider the black box model (also known as the query model), where a black box contains the input $X = (x_1, x_2, ..., x_N)$ and can be accessed by querying x_i values. The goal is to compute the value of the function. The complexity of a query algorithm is measured by the number of questions it asks. The classical version of this model is known as decision trees [1]. This computational model is widely applicable in software engineering. For instance, a database can be considered a black box, and, to speed up application performance, the goal is to reduce the number of database queries.

Quantum query algorithms can solve certain problems faster than classical algorithms. The quantum query model differs from the quantum circuit model [2–4], and algorithm construction techniques for this model are less developed. The problem of quantum query algorithm construction is very non-trivial. Although

[1] This research is supported by the European Social Fund project Nr. 2009/0138/1DP/1.1.2.1.2/09/IPIA/VIAA/004, Nr. ESS2009/77

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2343

there are many lower bound and upper bound estimations of quantum query algorithm complexity [2, 5–7], there are very few examples of original quantum query algorithms.

In this paper, we demonstrate an exact quantum query algorithm for resolving a specific problem. The task is to verify a codeword message that has been encoded using repetition code for detecting errors [8] and has been transmitted across a communication channel. Considered repetition code duplicates each bit of the message. The verification procedure can be considered as an application of a query algorithm, where the codeword to be checked is contained in a black box. To verify the message in the classical way, we would need to access all bits. That is, for a codeword of length N, all N queries to the black box would be required. We will demonstrate an exact quantum query algorithm that requires $N/2$ queries only.

An exact algorithm always produces a correct answer with 100% probability. Another variation is to use a bounded-error model, where an error margin of $1/3$ is allowed. It is well known that in the bounded-error model, a large difference between classical and quantum computation is possible. The complexity gap can be exponential as, for instance, in the case of Shor's algorithm [9]. Another famous example is Grover's search algorithm that achieves a quadratic speed up [10]. However, in certain types of computer software, we cannot allow even a small probability of error, for example, in spacecraft, aircraft, or medical software. For this reason, the development of exact algorithms is extremely important.

Regarding exact quantum algorithms, the maximum speedup achieved as of now is half the number of queries compared with a classical deterministic case [11]. The major open question is: is it possible to reduce the number of queries by more than 50%? In this paper, we present an algorithm that achieves a borderline gap of $N/2$ versus N.

2 Preliminaries

This section contains definitions and provides theoretical background on the subject.

2.1 2.1 Error Detection and Repetition Codes

In this article, we investigate a problem related to information transmission across a communication channel. The bit message is transmitted from a sender to a receiver. During that transfer, information may be corrupted. Because of the noise in a channel or adversary intervention some bits may disappear, or may be reverted, or even added. Various schemes exist to detect errors during transmission. In any case, a verification step is required after transmission. The received codeword is checked using defined rules and, as a result, a conclusion is made as to whether errors are present.

We consider a repetition error detection scheme known as repetition codes. A repetition code is a $(r,\, n)$ coding scheme that repeats each n-bit block r times

[8]. Verification procedure for repetition code is the following - we need to check if in each group of r consecutive blocks of size n all blocks are equal.

In this article, we examine verification of the (2,1) repetition code. The verification process can be expressed naturally as a computing Boolean function in a query model. We assume that the codeword to be checked is located in a black box. We define the Boolean function to be computed by the query algorithm as follows.

Definition 1. *The Boolean function $VERIFY_N(X)$, where $N = 2k$, $X = (x_1, x_2, ..., x_{2k})$ is defined to have a value of "1" Iff variables are equal by pairs:*

$$VERIFY_{2k}(X) = \begin{cases} 1, & if\ x_1 = x_2\ \&\ x_3 = x_4\ \&\ x_5 = x_6\ \&\ ...\ \&\ x_{2k-1} = x_{2k} \\ 0, & otherwise \end{cases}$$

2.2 Classical Decision Trees

The classical version of the query model is known as decision trees [1]. A black box contains the input $X = (x_1, x_2, ..., x_N)$ and can be accessed by querying x_i values. The algorithm must be able to determine the value of a function correctly for arbitrary input. The complexity of the algorithm is measured by the number of queries on the worst-case input. For more details, see the survey by Buhrman and de Wolf [1].

Definition 2. *[1] The deterministic complexity of a function f, denoted by D(f), is the maximum number of questions that must be asked on any input by a deterministic algorithm for f.*

Definition 3. *[1] The sensitivity $s_x(f)$ of f on input $X = (x_1, x_2, ..., x_N)$ is the number of variables x_i with the following property: $f(x_1, .., x_i, .., x_N) \neq f(x_1, .., 1 - x_i, .., x_N)$. The sensitivity of f is $s(f) = max_x s_x(f)$.*

It has been proved that $D(f) \geq s(f)$ [1].

Theorem 1. *$D(VERIFY_N) = N$.*

Proof. Check function sensitivity on any accepting input, for instance, on $X =$ 1111..11. Inversion of any bit will invert the function value, because a pair of bits with different values will appear. $s(VERIFY_N) = N \Rightarrow D(VERIFY_N) = N$.

2.3 Quantum Computing

This section briefly outlines the basic notions of quantum computing that are necessary to define the computational model used in this paper. For more details, see the textbooks by Nielsen and Chuang [3] and Kaye et al. [4].

An n-dimensional quantum pure state is a unit vector in a Hilbert space. Let $|0\rangle, |1\rangle, ..., |n-1\rangle$ be an orthonormal basis for $\mathbb{C}^n$. Then, any state can be expressed as $|\psi\rangle = \sum_{i=0}^{n-1} a_i |i\rangle$ for some $a_i \in \mathbb{C}$. Since the norm of $|\psi\rangle$ is 1, we have $\sum_{i=0}^{n-1} |a_i|^2 = 1$. States $|0\rangle, |1\rangle, ..., |n-1\rangle$ are called *basis states*. Any state of the form $\sum_{i=0}^{n-1} a_i |i\rangle$ is called a *superposition* of basis states. The coefficient a_i is called an *amplitude* of $|i\rangle$. The state of a system can be changed by applying *unitary transformation*. Unitary transformation U is a linear transformation on $\mathbb{C}^n$ that maps vector of unit norm to vector of unit norm. The *transpose* of a $m \times n$ matrix A is the $n \times m$ matrix $A^T_{i,j} = A_{j,i}$ for $1 \leq i \leq n$, $1 \leq j \leq n$. We denote the *tensor product* of two matrices by $A \otimes B$.

The simplest case of quantum *measurement* is used in our model. It is the full measurement in the computation basis. Performing this measurement on a state $|\psi\rangle = a_0 |0\rangle + ... + a_{n-1} |n-1\rangle$ gives the outcome i with probability $|a_i|^2$. The measurement changes the state of the system to $|i\rangle$ and destroys the original state.

2.4 Quantum Query Model

The quantum query model is the quantum counterpart of decision trees and is intended for computing Boolean functions. For a detailed description, see the survey by Ambainis [6] and textbooks by Kaye, Laflamme, Mosca [4] and de Wolf [2]. A quantum computation with T queries is a sequence of unitary transformations:

$$U_0 \to Q_0 \to U_1 \to Q_1 \to ... \to U_{T-1} \to Q_{T-1} \to U_T.$$

U_i's can be arbitrary unitary transformations that do not depend on the input bits. Q_i's are query transformations. A computation starts in the initial state $\left|\vec{0}\right\rangle$. Then we apply $U_0, Q_0, ..., Q_{T-1}, U_T$ and measure the final state.

We use the following definition of query transformation: if an input is a state $|\psi\rangle = \sum_i a_i |i\rangle$, then an output is $|\phi\rangle = \sum_i (-1)^{x_{k_i}} a_i |i\rangle$, where we can arbitrarily choose a variable assignment x_{k_i} for each basis state $|i\rangle$.

Each quantum basis state corresponds to the algorithm's output. We assign a value of a function to each output. The probability of obtaining result $j \in \{0, 1\}$ after executing an algorithm on an input X equals the sum of squared modulus of all amplitudes, which correspond to outputs with value j.

Definition 4. *[1] A quantum query algorithm computes f exactly if the output equals f(x) with a probability p=1, for all $x \in \{0, 1\}$. The complexity is denoted by $Q_E(f)$.*

3 Computing $VERIFY_N$ in a Quantum Query Model

In this section, we present the results of designing an exact quantum query algorithm for Boolean function $VERIFY_N(X)$. We start from the case of four variables and then show how to extend the algorithm to verify N-bit codewords. We have used a combinatorial approach to determine the structure of the algorithm, and have used *Mathematica* [14] software to verify its correctness. In our approach, we have tried to employ the full power of quantum parallelism, also known as computing in a superposition.

3.1 Exact Quantum Query Algorithm for $\boldsymbol{VERIFY_4}$

To familiarize the reader with the quantum query model and to build a base for extension, we demonstrate an algorithm for verification of 4-bit codewords. The algorithm flow is presented in Fig. 1.

Theorem 2. *There exists an exact quantum query algorithm Q1 that computes the Boolean function $VERIFY_4(X)$ using two queries: $Q_E(Q1) = 2$.*

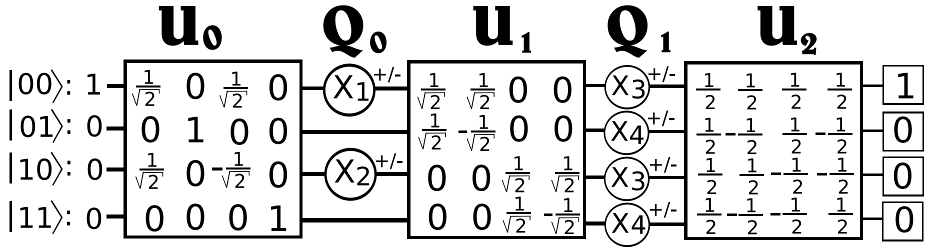


Fig. 1. Exact quantum query algorithm Q1 for computing $VERIFY_4$

The algorithm uses a 2-qubit quantum system. Each horizontal line corresponds to the amplitude of the basis state. Computation starts with amplitude distribution $|START\rangle = (1,0,0,0)^T$. Three large rectangles correspond to the 4×4 unitary matrices U_0, U_1 and U_2. Two vertical layers of circles specify the queried variable order for queries Q_0 and Q_1. Finally, four small squares at the end of each horizontal line define the assigned function value for each basis state.

We demonstrate an example of computational flow for accepting input X=1100:

$$|final\rangle = U_2Q_1U_1Q_0U_0\,(1,0,0,0)^T = U_2Q_1U_1Q_0\left(\tfrac{1}{\sqrt{2}},0,\tfrac{1}{\sqrt{2}},0\right)^T =$$
$$= U_2Q_1U_1\left(-\tfrac{1}{\sqrt{2}},0,-\tfrac{1}{\sqrt{2}},0\right)^T = U_2Q_1\left(-\tfrac{1}{2},-\tfrac{1}{2},-\tfrac{1}{2},-\tfrac{1}{2}\right)^T =$$
$$= U_2\left(-\tfrac{1}{2},-\tfrac{1}{2},-\tfrac{1}{2},-\tfrac{1}{2}\right)^T = (\mathbf{-1,\ 0,\ 0,\ 0}) \overset{Measure}{\Rightarrow} [ACCEPT : f(1100) = 1]$$

3.2 Exact Quantum Query Algorithm for $\boldsymbol{VERIFY_N}$

This section describes a generalized algorithm for computing the Boolean function $VERIFY_N$. In the previous section, we demonstrated in detail the first algorithm in the sequence. Now, we will show how to extend this approach to verify codewords of length N.

Theorem 3. *The Boolean function $VERIFY_N(X)$ can be computed by an exact quantum query algorithm using N/2 queries: $Q_E(VERIFY_N) = N/2$.*

We introduce an algorithm that will construct all required transformation matrices for a specified N. Then obtained transformations must be applied to the initial state in a specified order.

The algorithm is described in Table 1. The algorithm was implemented using *Mathematica* software, and its correctness was verified by a computer program.

Table 1. Exact quantum query algorithm for computing $VERIFY_N$

1. Setup

Boolean function to be computed: $VERIFY_N = (x_1, x_2, ..., x_N)$.
Number of queries: $T = N/2$. Number of qubits: T.
Number of amplitudes (dimension of Hilbert space): $K = 2^T = 2^{N/2}$.

2. Algorithm structure construction

```
FOR (i=1 to T) {
  STEP 1: Calculate a set of indices:
    $|IND| = 2^i; IND = \{ind_1, ind_2, ..., ind_{2^i}\}$;
    $IND = \left\{j \cdot \frac{K}{2^i} + 1 | j \in \left\{0, 1, ..., (2^i - 1)\right\}\right\}$
  STEP 2: Construct matrices $U_i$ and $Q_i$ :
    Initialize $U_i$ with the identity matrix $U_i = I_K$
    Initialize $Q_i$ with the identity matrix $Q_i = I_K$
    index=1;
    WHILE(index < $2^i$) {
      t1=IND[index] // $index^{th}$ element from the set IND
      t2=IND[index+1]
      Replace elements of $U_i$ and $Q_i$:
        $U_{t1,t1} = U_{t1,t2} = U_{t2,t1} = \frac{1}{\sqrt{2}}$
        $U_{t2,t2} = -\frac{1}{\sqrt{2}}$
        $Q_{t1,t1} = (-1)^{X_{2i-1}}$
        $Q_{t2,t2} = (-1)^{X_{2i}}$
      index = index + 2;
    }
}
STEP 3: Final transformation - $U_{FINAL} = H^{\otimes T}$, where $H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$.
STEP 4: Initial state - $|START\rangle = (1, 0, 0, ..., 0)^T$
STEP 5: Measurement - the only accepting state is $\left|\vec{0}\right\rangle = |000...0\rangle$.
```

3. Algorithm application

Execute the algorithm on input X by applying a constructed unitary and query transformations in the following order:
$|START\rangle \rightarrow U_1 \rightarrow Q_1 \rightarrow ... \rightarrow Q_T \rightarrow U_T \rightarrow U_{FINAL} \rightarrow [Measure]$.

3.3 Algorithm Analysis

To improve intuition and understanding, general algorithm for verification of N-bit codeword can be visualized as an abstract tree (see Fig. 2). We start at the top with state vector that has exactly one amplitude initialized to a=1 .

Queries and unitary transformations are formed and combined in such a way, that if values of function variables are equal by pairs, then in the final state

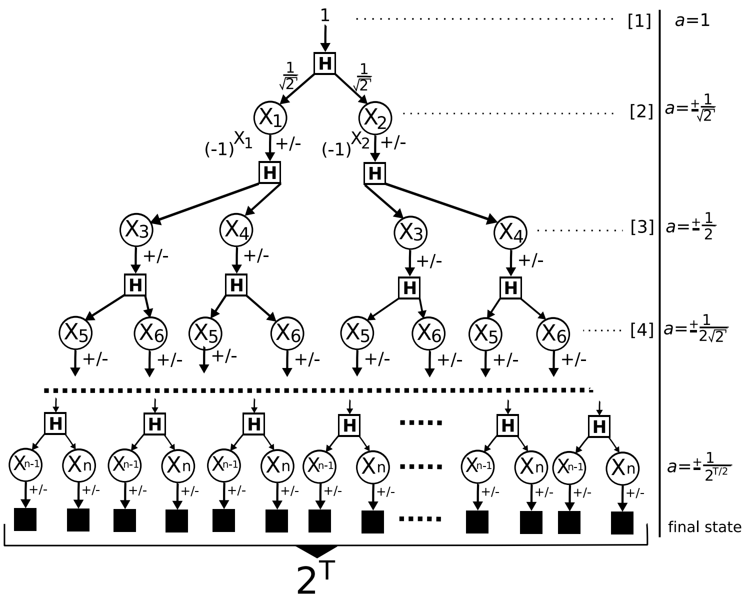


Fig. 2. Visualization of the quantum query algorithm as an abstract tree

vector signs of all amplitudes will be identical. At the same time, the first row of matrix U_{FINAL} consists of equal elements $+\frac{1}{2^{T/2}}$. It means that application of U_{FINAL} will join together all amplitudes and results in the state vector with $a = 1$ in the first position. So, the measurement will output the state $\left|\vec{0}\right\rangle$ with 100% probability. This is the accepting state $\Rightarrow VERIFY_N(X) = 1$.

If algorithm is executed on rejecting input, i.e., there is at least one pair of variables with different values, then after all T queries number of $+\frac{1}{2^{T/2}}$ and $-\frac{1}{2^{T/2}}$ amplitudes in state vector will be equal. This is provided by the algorithm structure. After multiplication with U_{FINAL} the value of the first amplitude will be zero, so there is no probability to obtain $\left|\vec{0}\right\rangle$ state after the measurement.

4 Application for a String Equality Problem

Described quantum algorithm can be adapted for solving such computational problem as testing if two binary strings are equal. This is a well-known task, which can be used as a subroutine in various algorithms.

Quantum algorithm for the Boolean function $VERIFY_N$ checks whether variables are equal by pairs, i.e., $x_1 = x_2$ & $x_3 = x_4$ & $x_5 = x_6$ & ... & $x_{2k-1} = x_{2k}$. On the other hand, we can consider that our algorithm is checking whether two binary strings, $Y = x_1x_3x_5...x_{2k-1}$ and $Z = x_2x_4x_6...x_{2k}$, are equal. Therefore, presented quantum algorithm can be easily used not only to verify repetition codes, but also for checking the equality of binary strings.

5 Conclusion

In this paper, we investigated the verification of error detection codes. We have represented the verification procedure as an application of a query algorithm to an input codeword contained in a black box. We have presented an exact quantum query algorithm, which allows verifying a codeword of length N using only $N/2$ queries to the black box. This algorithm saves exactly half the number of queries comparing to the classical case. This result repeats the largest difference between classical deterministic and quantum exact algorithm complexity for a total Boolean function known today in this model.

We see many possibilities for future research in the area of quantum query algorithm design. The most significant open question still remains: is it possible to increase exact algorithm performance more than two times using quantum tools? We believe that it may be possible. Next, there are many computational tasks waiting for efficient solution in a quantum setting. Regarding the verification of repetition codes, we would like to be able to verify not only (2,1) code, but also an arbitrary (r, n) code. Another fundamental goal is to develop a framework for building efficient ad-hoc quantum query algorithms for arbitrary Boolean functions.

References

1. H. Buhrman and R. de Wolf: Complexity Measures and Decision Tree Complexity: A Survey. Theoretical Computer Science, v. 288(1): 21-43 (2002).
2. R. de Wolf: Quantum Computing and Communication Complexity. University of Amsterdam (2001).
3. M. Nielsen, I. Chuang: Quantum Computation and Quantum Information. Cambridge University Press (2000).
4. P.Kaye, R.Laflamme, M.Mosca: An Introduction to Quantum Computing. Oxford (2007).
5. A.Ambainis: Quantum query algorithms and lower bounds (survey article). In Proceedings of FOTFS III, Trends on Logic, vol. 23 (2004), pp. 15-32.
6. A.Ambainis and R. de Wolf: Average-case quantum query complexity. Journal of Physics A 34, pp. 6741-6754 (2001).
7. A.Ambainis: Polynomial degree vs. quantum query complexity. Journal of Computer and System Sciences 72, pp. 220-238 (2006).
8. T. M. Cover and J. A. Thomas: Elements of Information Theory. pp. 209-212, Wiley-Interscience, (1991).
9. P. W. Shor: Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing, 26(5):1484-1509 (1997).
10. L. Grover: A fast quantum mechanical algorithm for database search. In Proceedings of 28th STOC'96, pp. 212. -219 (1996).
11. A.Ambainis. Personal communication, April 2009.
12. D. Deutsch and R. Jozsa: Rapid solutions of problems by quantum computation. In Proceedings of the Royal Society of London, volume A 439, pp. 553-558 (1992).
13. R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca: Quantum algorithms revisited. In Proceedings of the Royal Society of London, volume A 454, pp. 339-354 (1998).
14. Wolfram Research, Mathematica, http://www.wolfram.com/

Faster Algorithm for Mean-Payoff Games[⋆]

Jakub Chaloupka and Luboš Brim

Faculty of Informatics, Masaryk University,
Botanická 68a, 60200 Brno, Czech Republic
{xchalou1,brim}@fi.muni.cz

Abstract. We study some existing techniques for solving mean-payoff games (MPGs), improve them, and design a randomized algorithm for solving MPGs with currently the best expected complexity.

Key words: mean-payoff games, randomized algorithms, complexity

Introduction

A *Mean-Payoff Game (MPG)* [7, 8, 14] is a two-player infinite game on a finite weighted directed graph. The game is given by a graph $\mathcal{G} = (V, E, w)$ with integer edge-weights and a partition of the set of vertices V into the sets V_{Max} and V_{Min}. The two players, named Max and Min, move a token along the edges of $\mathcal{G}$ ad infinitum. If the token is on a vertex $v \in V_{\text{Max}}$, Max chooses an edge $(v, u) \in E$ and the token goes to u. If the token is on a vertex $v \in V_{\text{Min}}$, it is Min's turn to choose an outgoing edge. This way an infinite path is formed. Max wants to maximize the average edge weight of the path and Min wants to minimize it. It was proved [7] that each vertex $v \in V$ has a *value*, denoted by $\nu(v)$, which each player can secure by a positional strategy, i.e. strategy that always chooses the same outgoing edge in the same vertex. To solve a MPG is to find the values of all vertices and, optionally, also optimal strategies for both players, i.e. strategies that secure the values.

MPGs have many applications, especially in the synthesis, analysis and verification of reactive (non-terminating) systems. Many natural models of such systems include quantitative information, and the corresponding question requires the solution of quantitative games, like MPGs. Quantities may represent, for example, the power usage of an embedded component, or the buffer size of a networking element [4].

Examples of applications include various kinds of scheduling, finite-window online string matching, or more generally, analysis of online problems and algorithms, and selection with limited storage [14]. Moreover, μ-calculus model-checking is polynomial-time reducible to MPGs via parity games [9]. MPGs can even be used for solving the max-plus algebra $Ax = Bx$ problem, which in turn has further applications [6].

[⋆] This work has been partially supported by the Grant Agency of the Czech Republic grant No. 201/09/1389.

MPGs are also important from a theoretical point of view. The problem whether the value of a certain vertex is greater or less than a certain threshold is in the complexity class $\mathsf{NP} \cap \mathsf{co\text{-}NP}$ and it is not known whether the problem is in P.

Because of their importance, MPGs have attracted many researchers, especially in the last decade, and several algorithms for solving MPGs have been proposed. They can be roughly divided into two categories. In the first category are algorithms based on linear programming [2, 13]. This category also includes algorithms based on reduction to discounted payoff games [11] and simple stochastic games [5]. In the second category are pure combinatorial graph algorithms [14, 3, 10, 6, 8, 12].

The best complexity attained by a deterministic algorithm for solving MPGs is pseudo-polynomial, namely $O(|V|^3 \cdot |E| \cdot W)$, where $|V|$ and $|E|$ are numbers of vertices and edges, respectively, and W is the maximal absolute edge-weight. It is the complexity of the algorithm of Zwick and Paterson [14] (ZP).

In this paper, we design a deterministic combinatorial algorithm with the complexity $O(|V|^2 \cdot |E| \cdot W \cdot \log(|V| \cdot W))$, which is better for W up to $2^{O(|V|)}$. To get an algorithm which is better for all values of W, we combine our algorithm with the randomized algorithm of Andersson and Vorobyov [1], and get an algorithm with currently the best expected complexity. We note that in typical applications, where the edge-weights represent, for example, the energy consumption of a physical device, W is usually small in comparison with $|V|$, in which case our deterministic algorithm is significantly better than ZP without the need to combine it with any other algorithm.

1 Preliminaries

A *Mean-Payoff Game* (MPG) [7, 8, 14] is given by a triple $(\mathcal{G}, V_{\text{Max}}, V_{\text{Min}})$, where $\mathcal{G} = (V, E, w)$ is a finite weighted directed graph such that V is a disjoint union of the sets V_{Max} and V_{Min}, $w : E \to \mathbb{Z}$ is the weight function, and each $v \in V$ has out-degree at least one. The game is played by two opposing players, named Max and Min. A play starts by placing a token on some given vertex and the players then move the token along the edges of $\mathcal{G}$ ad infinitum. If the token is on vertex $v \in V_{\text{Max}}$, Max moves it. If the token is on vertex $v \in V_{\text{Min}}$, Min moves it. This way an infinite path $p = (v_0, v_1, v_2, \ldots)$ is formed. Max's aim is to maximize his gain: $\liminf_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} w(v_i, v_{i+1})$, and Min's aim is to minimize her loss: $\limsup_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} w(v_i, v_{i+1})$. For each vertex $v \in V$, we define its *value*, denoted by $\nu(v)$, as the maximum gain that Max can ensure if the play starts at vertex v. It was proved that it is equal to the minimum loss that Min can ensure. Moreover, both players can ensure $\nu(v)$ by using positional strategies defined below [7]. A strategy that ensures $\nu(v)$, for all $v \in V$ is called an *optimal* strategy. To solve an MPG is to find the values of all vertices and, optionally, also optimal positional strategies for both players.

A *positional strategy* for Max is a function $\sigma : V_{\text{Max}} \to V$ such that $(v, \sigma(v)) \in E$, for each $v \in V_{\text{Max}}$ (Recall that each vertex has out-degree at least one).

A positional strategy for Min is defined analogously, except that it is usually denoted by π. We define $\mathcal{G}_\sigma$, the restriction of $\mathcal{G}$ to σ, as the graph (V, E_σ, w_σ), where $E_\sigma = \{(u,v) \in E \mid u \in V_{\mathrm{Min}} \vee \sigma(u) = v\}$, and $w_\sigma = w|_{E_\sigma}$. That is, we get $\mathcal{G}_\sigma$ from $\mathcal{G}$ by deleting all the edges emanating from Max's vertices that do not follow σ. Let now σ be a strategy of Max and let π be a strategy of Min. $\mathcal{G}_\sigma$ has just been defined, $\mathcal{G}_\pi$ is defined analogously, and $\mathcal{G}_{\sigma\cup\pi}$ is the intersection of $\mathcal{G}_\sigma$ and $\mathcal{G}_\pi$, i.e., $\mathcal{G}_{\sigma\cup\pi} = (V, E_{\sigma\cup\pi}, w_{\sigma\cup\pi})$, where $E_{\sigma\cup\pi} = \{(u,v) \in E \mid (u \in V_{\mathrm{Min}} \wedge \pi(u) = v) \vee (u \in V_{\mathrm{Max}} \wedge \sigma(u) = v)\}$, and $w_{\sigma\cup\pi} = w|_{E_{\sigma\cup\pi}}$.

From the existence of optimal positional strategies it follows that $\nu(v)$ for each $v \in V$ is a a fraction with denominator at most $|V|$. It is because the ν values are the mean-weights of certain cycles, namely the cycles in $\mathcal{G}_{\sigma\cup\pi}$, where σ and π are optimal positional strategies for Max and Min, respectively. In $\mathcal{G}_{\sigma\cup\pi}$, each vertex has out-degree exactly one, and so for each $v \in V$, there is a unique cycle reachable from v, and $\nu(v)$ is equal to the mean-weight of that cycle. The fact that the ν values of vertices are mean-weights of cycles also implies that $\nu(v) \in [-W, W]$, for each $v \in V$, where $W = \max_{e \in E} |w(e)|$.

2 Algorithm

Our algorithm solves only the 0-mean partition problem. That is, it divides the vertices of the graph into those with $\nu \geq 0$ and those with $\nu < 0$. How to use the algorithm to compute the exact ν values will be described later in this section.

Our algorithm computes, for each vertex $v \in V$, the value $d_{\geq 0}(v)$ – the minimum value such that Max can ensure that the sum of traversed edges in a play starting from v, plus $d_{\geq 0}(v)$, never goes below 0. The $d_{\geq 0}$ value is finite only for vertices with $\nu \geq 0$, because for plays starting from vertices with negative ν value, Min has a strategy that ensures that all traversed cycles are negative, and so Max is unable to keep the sum of traversed edges nonnegative forever, no matter how high his starting "energy" is. Therefore, for each vertex $v \in V$ such that $\nu(v) < 0$, the algorithm sets $d_{\geq 0}(v) = \infty$.

Chakrabarti et al. [4] proposed a simple algorithm based on value iteration that solves a similar problem. The difference is that the weights are on vertices, not edges. The complexity of their algorithm is $O(|V|^2 \cdot |E| \cdot W)$. We adjusted the algorithm so that it works with weights on edges and improved its complexity to $O(|V| \cdot |E| \cdot W)$.

Our algorithm proceeds in iterations. It starts with $d_0(v) = 0$, for each $v \in V$, and then computes $d_1, d_2, \ldots$ according to the following rules.

$$d_{i+1}(v) = \begin{cases} \min_{(v,u)\in E} \max(0, d_i(u) - w(v,u)) & \text{if } v \in V_{\mathrm{Max}} \\ \max_{(v,u)\in E} \max(0, d_i(u) - w(v,u)) & \text{if } v \in V_{\mathrm{Min}} \end{cases} \tag{1}$$

It is easy to see that for each $v \in V$ and $k \in \mathbb{N}_0$, $d_k(v)$ is the minimum amount of Max's starting energy, that enables him to keep the sum of traversed edges, plus $d_k(v)$, greater or equal to zero in a k-step play. The computation continues until two consecutive d vectors are equal. The last d vector is then the desired vector $d_{\geq 0}$. Please note that the d value of each vertex is non-decreasing, that is, for each $v \in V$, $d_0(v) \leq d_1(v) \leq d_2(v) \leq \cdots$.

This works well if all vertices have $\nu \geq 0$. If some vertex has $\nu < 0$, then its d value never stops increasing and the value iteration does not terminate. Fortunately, there is an upper bound on the d values of vertices with $\nu \geq 0$ and if for some vertex v, $d(v)$ goes past that bound, we know that $\nu(v) < 0$. The said bound is $(|V| - 1) \cdot W$. The reason is the following.

Max has a positional strategy σ such that for each vertex $v \in V$ such that $\nu(v) \geq 0$, all cycles reachable from v in $\mathcal{G}_\sigma$ are non-negative. For the sake of contradiction, let's suppose that Min has a strategy π, not necessarily positional, that for a play starting from some vertex v_0 such that $\nu(v_0) \geq 0$ guarantees that at some point the traversed path has weight less than $-(|V|-1) \cdot W$. Let Max use the strategy σ against π and let $p = (v_0, \ldots, v_k)$ be a path agreeing with σ and π such that $w(p) < -(|V|-1) \cdot W$. Recall that since Max uses the strategy σ, all traversed cycles must be non-negative. Since $w(p) < -(|V|-1) \cdot W$, the number of vertices in p must be greater than $|V|$. Therefore, we can apply the following transformation on p. Start from v_0 and go along the path until an already visited vertex is encountered, then remove the found cycle from p. Since the cycle is non-negative, we get a shorter path p' such that $w(p') \leq w(p) < -(|V| - 1) \cdot W$. We can continue in this fashion until the path has less than $|V|$ vertices and get a contradiction with the fact than no path with less than $|V|$ vertices can have weight less than $-(|V| - 1) \cdot W$. Therefore, if $\nu(v) \geq 0$, then $(|V| - 1) \cdot W$ is a sufficient amount of starting energy to keep the energy level non-negative ad infinitum.

Based on the facts above, we complete the algorithm by adding a test, for each vertex $v \in V$, whether $d(v)$ is greater than $(|V| - 1) \cdot W$ or not. If it is, we set $d(v)$ to ∞, which is then handled in the usual way: $\infty - a = \infty$, where $a \in \mathbb{Z}$. This guarantees the termination of the algorithm. It follows that the d value of each vertex can be improved at most $O(|V| \cdot W)$ times, so the algorithm makes at most $O(|V|^2 \cdot W)$ iterations. Since the complexity of one iteration is $O(|E|)$, we get the overall complexity $O(|V|^2 \cdot |E| \cdot W)$. However, this can be improved to $O(|V| \cdot |E| \cdot W)$. The key ingredients are the following. The first is to keep track of the vertices that increase their d values, so that vertices that cannot bring any improvement are not explored. The second is, for each vertex $v \in V_{\mathrm{Max}}$, to keep track of the number of successors of v that give it the minimum from (1) to be able to quickly determine whether the d value of v has to be improved.

In Figure 1 is a pseudo-code of our improved algorithm. It works with three vectors of size $|V|$, $d_{pre}, d, d' \in \mathbb{N}_0^V$. The vector d contains the current iteration d values of vertices, while d_{pre} and d' contain the previous and the next iteration d values, respectively. The initialization is on lines 2–14. It initializes all the vectors to vectors of zeros, except for the elements of d' that correspond to Max's vertices. For these elements, the algorithm already computes the next iteration d values using (1). For each Max's vertex, it also computes the number of "optimal" edges, the edges which give the vertex the minimum. These numbers are stored in the vector *nopt*. There are also two queues maintained by the algorithm. The queue q contains the vertices with improved (increased) d value. Initially, the queue contains all vertices of the graph. The other queue is q' and it accumulates the vertices with improved d value for the next iteration.

The main loop of the algorithm is on lines 15–43. The termination criterion of the loop is the queue of vertices with improved d values being empty. Each iteration performs the following steps. It loops over the vertices with improved distance on lines 16–35 and for each improved vertex v, it explores all of its predecessors. The following steps depend on whether the predecessor u is Max's or Min's vertex. For $u \in V_{\mathrm{Min}}$, we check if the improvement of the d value of v yields also an improvement of the d value of u. If yes, we update $d'(u)$ and if it is the first improvement of u's d value in this iteration of the main loop, we also put u into the queue of improved vertices for the next iteration, q'. If u is Max's vertex, things are a little bit more complicated.

Since according to (1), minima are computed at Max's vertices, it isn't that easy to determine whether the improvement of v's d value yields an improvement of u's value. This is where the vector *nopt* helps. The d value of u is improved only if all the successors giving it the current minimum increase their d value. The vector *nopt* holds, for each Max's vertex, the number of successors that give the vertex the current minimum. Therefore, if v is one of the vertices that give u its current minimum (condition on line 20), we decrease $nopt(u)$ by one, but only if it no longer gives u the current minimum (condition on line 21). If $nopt(u)$ drops to zero, we recompute the minimum at u, compute the value of $nopt(u)$ for the new minimum, and put u to the queue of improved vertices for the next iteration (lines 22–26). When all vertices from q are explored, we prepare the algorithm for the next iteration of the main loop.

On lines 36–41, we move all the elements from q' to q and update the vectors d_{pre} and d. The current d values of the improved vertices become their previous d values (line 39) and the new d values of these vertices become their current d values (line 40). If a new value is greater than the bound, it becomes infinity. After the queue q' is processed, we increase i and start another iteration, but only if there are some improved vertices.

The complexity analysis of the algorithm is quite simple. The d value of each vertex can be improved at most $O(|V| \cdot W)$ times, so let's determine what is the complexity of one improvement of one vertex. For definiteness, let's denote the vertex by v. If v is improved, we explore it's predecessors in the next iteration and that's $O(indegree(v))$ operations. For $v \in V_{\mathrm{Max}}$, we also have to compute the actual value of the improvement if we detect that v's d value has to be improved. The computation explores all of the successors of v which takes $O(outdegree(v))$ time. So the overall complexity of the algorithm is $O(|V| \cdot |W| \cdot \sum_{v \in V}(indegree(v) + outdegree(v)))$, and since $\sum_{v \in V}(indegree(v) + outdegree(v)) = 2 \cdot |E|$, the overall complexity is $O(|V| \cdot |E| \cdot W)$.

Our algorithm solves only the 0-mean partition problem. It divides the vertices of $\mathcal{G}$ into those with $\nu \geq 0$ (the vertices with finite d values after the termination of the algorithm) and those with $\nu < 0$ (the vertices with infinite d values after the termination). The algorithm can also be used to solve the p-mean

```
proc VI($\mathcal{G} = (V, E, w), V_{\text{Max}}, V_{\text{Min}}$)
  foreach $v \in V$ do
    $d_{pre}(v) := 0$
    $d(v) := 0$
    if $v \in V_{\text{Max}}$ then
      $d'(v) := \max(0, \min_{(v,z)\in E}(0 - w(v, z)))$
      $nopt(v) :=\mid \{(v, z) \in E \mid d'(v) = \max(0, 0 - w(v, z))\} \mid$
      if $d'(v) > 0$ then $q'.enqueue(v)$ fi
    else
      $d'(v) := 0$
    fi
    $q.enqueue(v)$
  od
  $i := 0$
  while $\neg q.empty()$ do
    while $\neg q.empty()$ do
      $v := q.dequeue()$
      foreach $(u, v) \in E$ do
        if $u \in V_{\text{Max}}$ then
          if $i > 0 \ \wedge \ d(u) \geq d_{pre}(v) - w(u, v)$ then
            if $d(v) - w(u, v) > 0$ then $nopt(u) := nopt(u) - 1$ fi
            if $nopt(u) = 0$ then
              $d'(u) := \min_{(u,z)\in E}(d(z) - w(u, z))$
              $nopt(u) :=\mid \{(u, z) \in E \mid d'(u) = d(z) - w(u, z)\} \mid$
              $q'.enqueue(u)$
            fi
          fi
        else
          if $d'(u) < d(v) - w(u, v)$ then
            if $d'(u) = d(u)$ then $q'.enqueue(u)$ fi
            $d'(u) := d(v) - w(u, v)$
          fi
        fi
      od
    od
    while $\neg q'.empty()$ do
      $v := q'.dequeue()$
      $q.enqueue(v)$
      $d_{pre}(v) := d(v)$
      if $d'(v) > (|V|-1) \cdot W$ then $d'(v) := \infty$ fi; $d(v) := d'(v)$
    od
    $i := i + 1$
  od
  return $(\{v \in V \mid d(v) < \infty\}, \{v \in V \mid d(v) = \infty\})$
end
```

Fig. 1. Improved value iteration for solving the zero-mean partition problem

partition problem for arbitrary rational number p, because if we subtract p from all edge-weights, then the ν values of all vertices also decrease by p. The exact ν values of all vertices can then be computed by binary search. The complexity analysis of the resulting algorithm follows.

Since the ν values are fractions with denominators at most $|V|$, we have to solve the p-mean partition problems only for ps with denominators at most $|V|$. This increases the complexity of the 0-mean partitioning algorithm to $O(|V|^2 \cdot |E| \cdot W)$, because the smallest possible increase of a d value of a vertex is not 1 but $1/|V|$. To be able to determine the ν values exactly, we run each branch of the binary search until the size of the search interval is at most $1/|V|^2$. In such a small interval there can be only one rational number with denominator at most $|V|$, and this is the ν value of the vertices in that branch of the binary search. Since the ν value of each vertex is in the interval $[-W, W]$, the depth of the binary search is in $O(\log(|V|^2 \cdot W)) = O(\log(|V| \cdot W))$, and so the overall complexity of the algorithm is $O(|V|^2 \cdot |E| \cdot W \cdot \log(|V| \cdot W))$.

To obtain an algorithm with currently the best complexity we combine our algorithm with the randomized algorithm of Andersson and Vorobyov [1]. It has the complexity $|V|^2 \cdot |E| \cdot e^{2\cdot\sqrt{|V|\cdot\ln(|E|/\sqrt{|V|})}+O(\sqrt{|V|}+\ln|E|)}$, which is better than the complexity of our algorithm for large W. If we interleave the two algorithms and add a stopping criterion which terminates the computation when either of the two algorithms finishes, we get a randomized algorithm with the expected complexity $\min(O(|V|^2 \cdot |E| \cdot W \cdot \log(|V| \cdot W)), |V|^2 \cdot |E| \cdot e^{2\cdot\sqrt{|V|\cdot\ln(|E|/\sqrt{|V|})}+O(\sqrt{|V|}+\ln|E|)})$, which is currently the best expected complexity of an algorithm for solving mean-payoff games. This deserves a little more comments, which the following subsection is dedicated to.

2.1 Comparison with other algorithms

The algorithm of Zwick and Paterson [14] (ZP) has the complexity $O(|V|^3 \cdot |E| \cdot W)$, which is better than the first term of the complexity of our algorithm for very large W. However, it is not better for W up to $2^{O(|V|)}$, and for W in $2^{O(|V|)}$, the second term of our algorithm is already better. Therefore, our algorithm has better complexity than ZP.

We must also compare our algorithm with the algorithm of Björklund and Vorobyov [3] (BV), which has the complexity $\min(O(|V|^3 \cdot |E| \cdot W \cdot \log(|V| \cdot W)), (\log W)\cdot 2^{O(\sqrt{|V|\cdot\log|V|})})$. This can be improved to $\min(O(|V|\cdot(|V|\cdot\log|V| + |E|)\cdot W \cdot \log(|V|\cdot W)), (\log W)\cdot 2^{O(\sqrt{|V|\cdot\log|V|})})$ by the following modifications to the algorithm. The first is to use Dijkstra's algorithm instead of Bellman-Ford's algorithm, which is one of the subroutines of BV. This is made possible by a potential transformation of the edge-weights as described by Schewe [12]. The second modification is to use a technique similar to the key technique of our algorithm that improves its complexity by a factor of $|V|$. However, the first term of the complexity of BV exceeds the first term of the complexity of our algorithm, and the second term of the complexity of BV exceeds the second term of the complexity of our algorithm, even for small W. Therefore, BV is worse. We stress that the improvement of BV outlined above is not straightforward and it is an interesting result per se, but since we achieved better complexity with our algorithm, and also for space reasons, we decided not to give the details of the improvement here.

The algorithm of Svensson and Vorobyov [13] based on linear programming has the complexity $O(|V| \cdot |E| \cdot W)$. However, it solves only the 0-mean partition problem for bipartite games with no zero cycles. The 0-mean partition problem for general games can be reduced to the special case, but the reduction increases edge-weights by a factor of $|V|$. The exact ν values of all vertices can be computed by binary search, but, as already mentioned, it requires solving p-mean partition problems for rational ps, which increases the complexity by another factor of $|V|$. All in all, computation of the exact ν values using the algorithm of Svensson and Vorobyov and binary search has the complexity $O(|V|^3 \cdot |E| \cdot W \cdot \log(|V| \cdot W))$, which exceeds the complexity of our algorithm.

The algorithm of Lifshits and Pavlov [10] has the complexity $O(|V| \cdot |E| \cdot 2^{|V|} \cdot \log(W))$, which is worse than the second term of the complexity of our algorithm.

All the other algorithms for solving MPGs we know of have either the same or worse complexity than the algorithms we have already compared our algorithm with.

3 Conclusion

We designed a deterministic algorithm for solving mean-payoff games with the complexity $O(|V|^2 \cdot |E| \cdot W \cdot \log(|V| \cdot W))$. In combination with the randomized algorithm of Andersson and Vorobyov [1], it is a randomized algorithm with currently the best expected complexity, namely:

$$\min(O(|V|^2 \cdot |E| \cdot W \cdot \log(|V| \cdot W)), |V|^2 \cdot |E| \cdot e^{2 \cdot \sqrt{|V| \cdot \ln(|E|/\sqrt{|V|})} + O(\sqrt{|V|} + \ln |E|)})$$

References

1. D. Andersson and S. Vorobyov. Fast algorithms for monotonic discounted linear programs with two variables per inequality. Technical Report Preprint NI06019-LAA, Isaac Newton Institute for Mathematical Sciences, Cambridge, UK, 2006.
2. H. Björklund, O. Svensson, and S. Vorobyov. Linear complementarity algorithms for mean payoff games. Technical Report DIMACS-2005-13, DIMACS, New Jersey, USA, 2005.
3. H. Björklund and S. Vorobyov. A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. *Discrete Applied Math.*, 155(2):210–229, 2007.
4. A. Chakrabarti, L. de Alfaro, T. Henzinger, and M. Stoelinga. Resource interfaces. In *Proc. Embedded Software*, volume 2855 of *LNCS*, pages 117–133. Springer, 2003.

5. A. Condon. On algorithms for simple stochastic games. In *Advances in Computational Complexity Theory*, volume 13 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 51–73. American Mathematical Society, 1993.
6. V. Dhingra and S. Gaubert. How to solve large scale deterministic games with mean payoff by policy iteration. In *Proc. Performance evaluation methodolgies and tools,* article no. 12. ACM, 2006.
7. A. Ehrenfeucht and J. Mycielski. Positional strategies for mean payoff games. *International Journal of Game Theory*, 8(2):109–113, 1979.
8. V. A. Gurvich, A. V. Karzanov, and L. G. Khachivan. Cyclic games and an algorithm to find minimax cycle means in directed graphs. *USSR Comput. Math. and Math. Phys.*, 28(5):85–91, 1988.
9. M. Jurdziński. Deciding the winner in parity games is in UP $\cap$ co-UP. *Inf. Process. Lett.*, 68(3):119–124, 1998.
10. Y. Lifshits and D. Pavlov. Fast exponential deterministic algorithm for mean payoff games. *Zapiski Nauchnyh Seminarov POMI*, 340:61–75, 2006.
11. A. Puri. Theory of hybrid systems and discrete event systems. Phd thesis, EECS University of Berkeley, Berkeley, CA, USA, 1995.
12. S. Schewe. An optimal strategy improvement algorithm for solving parity and payoff games. In *Proc. Computer Science Logic*, volume 5213 of *LNCS*, pages 369–384. Springer, 2008.
13. O. Svensson and S. Vorobyov. Linear programming polytope and algorithm for mean payoff games. In *Proc. Algorithmic Aspects in Information and Management*, volume 4041 of *LNCS*, pages 64–78. Springer, 2006.
14. U. Zwick and M. S. Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158(1–2):343–359, 1996.

One Size Does Not Fit All – How to Approach Intrusion Detection in Wireless Sensor Networks

Andriy Stetsko and Václav Matyáš

Department of Computer Systems and Communications
Faculty of Informatics, Masaryk University
`{xstetsko, matyas}@fi.muni.cz`

Abstract. A wireless sensor network (WSN) is a highly distributed network of resource constrained and wireless devices called sensor nodes. In the work we consider intrusion detection systems as they are proper mechanisms to defend internal attacks on WSNs. A wide diversity of WSN applications on one side and limited resources on other side implies that "one-fit-all" intrusion detection system is not optimal. We present a conceptual proposal for a suite of tools that enable an automatic design of intrusion detection system that will be (near) optimal for a given network topology, capabilities of sensor nodes and anticipated attacks.

1 Introduction

A wireless sensor network (WSN) consists of sensor nodes – devices that are equipped with sensor(s), microcontroller, wireless transceiver and battery. Each sensor node monitors some physical phenomenons (e.g., humidity, temperature, pressure, light, etc.) inside an area of deployment. The collected measurements are then sent to a base station – a gateway between a WSN and external world (in most cases the Internet).

In the work we consider WSNs that contain hundreds of thousands of nodes distributed over an area of hundreds square kilometers. Communication range of sensor nodes is limited to tens of meters and hence not all of them can directly communicate with a base station. Therefore, data are sent hop-by-hop from one sensor node to another until they reach a base station (see Figure 1).

Sensor nodes are constrained in processing power and energy, whereas a base station is assumed to have laptop capabilities and unlimited energy resources. Crossbow MICAz[1] is an example of average sensor node. It contains Atmel Atmega128L microcontroller, 802.15.4 compliant (250kbps) Texas Instruments CC2420 transceiver and two AA batteries. The microcontroller features 8b processor (operating at 8MHz), 128kB FLASH, 4kB EEPROM and 4kB SRAM. Currently the sensor node is available at price of €110. That eliminates deployment of a large number of sensor nodes. However, it is believed that recent advances in micro-electro-mechanical systems will decrease the cost significantly.

[1] See manufacturer's website http://www.xbow.com/.

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2347

It is expected that WSNs will have many applications in military, ecology, building and industrial automation, energy management, agriculture and even wildlife monitoring. Security becomes an important issue for WSNs and brings new challenges for security engineers.

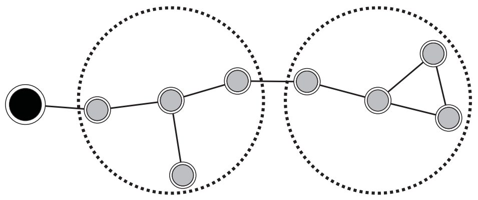


Fig. 1: Wireless sensor network. A base station is depicted as the black filled circle and sensor nodes are depicted as gray ones. We assume that communication ranges (represented by dotted circles) of neighboring sensor nodes are symmetric.

Cryptographic techniques can be used to prevent an external attacker (outsider) [9] from eavesdropping or altering the ongoing communication[2]. Encryption does not solve the problem of jamming attacks, where a malicious node (or other device) purposefully tries to interfere with physical transmission and reception of wireless communication.

An area of deployment is most often not physically protected and an attacker can easily access the area and capture some nodes[3]. Being a legitimate participant of the network the attacker (insider) can launch a variety of internal attacks. In the work we consider: a selective forwarding attack in which an attacker selectively drops packets [5]; a sinkhole attack in which an attacker attracts all traffic from a particular area towards itself, typically by making a compromised node look attractive to neighboring nodes with respect to routing algorithm [5]; a packet alternation attack in which a malicious node modifies packets that it forwards for the neighbors.

Sensor nodes are not tamper-resistant and an attacker can extract cryptographic keys from captured nodes. The attacker can replicate (also known as clone attack) [6] the nodes, deploy them into a network and then launch attacks described above. The attacker can also create nodes with several identities, also known as Sybil nodes [5]. These nodes may have an impact on multipath routing, voting, data aggregation, fair-resource allocation and misbehavior detection.

In this work we consider intrusion detection systems (IDSs) since they are, in comparison to cryptographic techniques, better mechanisms to defend against internal attacks on WSNs. In Section 2 we describe basics of intrusion detection systems for wireless sensor networks – what kinds of audit data can be gathered

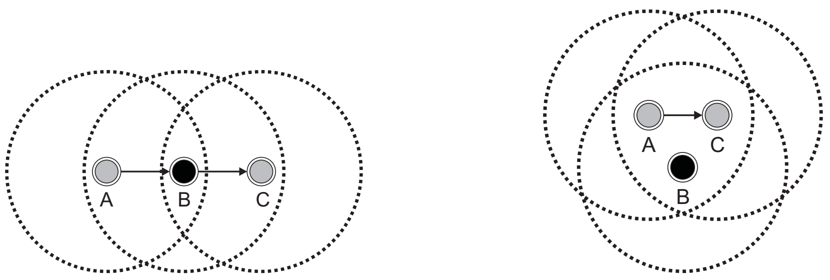
[2] A survey on performance of symmetric/asymmetric cryptographic primitives and hash functions implemented for WSNs is available in [8].

[3] We assume that a number of such nodes is significantly smaller than a total number of sensor nodes in the network.

and for detection of what types of attack they can be used. “One-fit-all” IDS is not optimal because of the wide range of WSN applications and limited resources of sensor nodes. In Section 3 we propose a conceptual architecture of a suite of tools that will provide administrators with an IDS that fits best its purposes.

2 Intrusion detection in wireless sensor networks

In the work we consider a distributed IDS that consists of IDS agents. We assume that every sensor node runs an IDS agent which monitors its neighbors using both local and watchdog monitoring techniques [1]. In the local monitoring technique sensor nodes collect and analyze only data forwarded by themselves (see Figure 2a). In the watchdog technique, sensor nodes collect an analyze data overheard in their neighborhood (see Figure 2b). We assume that sensor nodes employ single-channel transceivers. However, if the multi-channel transceivers are used, it might happen (the worst scenario) that the watchdog technique will be useless and an IDS will have to rely only on the local monitoring technique.



(a) The sensor node B monitors traffic that it forwards from the node A to the node C

(b) The sensor node B monitors in promiscuous mode traffic from the node A to the node C

Fig. 2: Traffic monitoring techniques

A conceptual model of an IDS agent is presented in [10]. Audit data gathered by a local audit data collection module are subsequently analyzed by a local detection module(s). A cooperative detection module is used to propagate intrusion detection state information or/and audit data among neighboring nodes. In case a local detection evidence is weak or inconclusive the cooperative detection module can use information (e.g., audit data) received from other IDS agents to detect an ongoing attack. After the attack is detected local response and global response modules trigger reactions. We assume that the local response module will stop any communication with the malicious node. The global response module will notify an administrator and he/she will remove or reprogram the malicious node. A secure communication module should provide cooperating nodes with a secure communication channel.

We assume that IDS agents will be implemented for TinyOS – the most widely used operating system for wireless sensor networks. It is important to understand what kind of network audit information we can gather. In TinyOS, the basic network abstraction is an active message that includes source and destination addresses [3]. Also it provides synchronous acknowledgements. Hardware independent components (e.g., active message) are built on top of hardware dependent components. Crossbow Imote2, MICAz and TELOSB sensor nodes as well as Sentilla Tmote Sky sensor nodes use TI CC2420 transceiver[4]. The corresponding CC2420 Radio Stack [4] supplies each outgoing message with a unique data sequence number and also provides possibility to read RSSI (received signal strength indication) of each received packet.

The audit data collection module logs source and destination addresses of received, overheard and sent packets. Also it logs: RSSI of received packets as well as information whether a packet passed CRC check or not; information whether an attempt to send a packet was successful or not, whether a packet was received by recipient (acknowledged) and how much time was spent waiting for the channel (carrier sensing time). All this information is gathered for a period of time of duration T and we call it temporal information. Having this information, different temporal statistics can be calculated. Examples of such statistics are presented below.

1. Packet delivery ratio – a ratio of packets that are successfully delivered to a destination compared to a number of packets that were sent by the sender.
2. Packet sending rate – a number of packets sent by a neighboring sensor node.
3. Packet receiving rate – a number of packets received by a neighboring sensor node.
4. Packet dropping rate – a ratio of packets sent by a neighboring sensor node with respect to a number of packets received by that node.
5. Number of neighbors.
6. A function of RSSI (e.g., average, maximum).
7. A function of carrier sensing time (e.g., average, maximum).

Packet delivery ratio can be calculated either at the sender or at the receiver. At the receiver it is calculated as a ratio of a number of packets that passed the CRC check with respect to a number of packets received. At the sender it is calculated as a ratio of a number of ACK received with respect to a number of packets sent.

The calculated statistics can be used to detect different attacks. For example, carrier sensing time, packet delivery ratio and RSSI are used to detect jamming attacks [2]. Selective forwarding attacks can be detected by monitoring packet dropping rate [1]. Packet receiving ratio can be used to detect sinkhole attacks.

[4] We consider these radio chips since they are 802.11.4 compliant and provide hardware support of AES.

3 Intrusion detection system fits to purpose

Density of a network, capabilities of involved sensor nodes, anticipated types of attack and other critical parameters may vary from one application to another. Due to a wide range of WSN applications on one side and limited resources on other side a "one-fit-all" IDS is not optimal. Therefore, we propose to make a two-level optimization.

1. In order to detect different attacks a variety of local detection modules can be implemented. An administrator will specify anticipated attacks and we propose to include only such detection modules in the IDS agent that detect the specified attacks. This approach will save memory and energy that are very important for WSNs. To our best knowledge nobody has yet applied such idea to WSNs.

2. Parameters of detection modules (included in the IDS agent configuration) might not be optimal for a given application. We propose to optimize them for a network topology, sensor nodes capabilities and anticipated types of attack, which all will be specified by a network administrator. In conventional networks, in majority of cases, a trade-off between a number of false positives, a number of false negatives and memory usage is found. However, for WSNs this is not enough. Sensor nodes are energy constrained and if ever depleted they will stop fulfilling their main goal – monitoring of area of deployment. Therefore, for WSNs a trade-off between detection accuracy, memory usage and energy usage should be found. For example, if IDS agents cooperate between themselves it involves communication, which in comparison with computation, consumes significantly more energy [7]. On other side cooperation increases a detection accuracy since a single monitoring node may not have enough information to detect an attack, e.g., due to collisions [1].

We propose a suite of tools that should provide an administrator of a network with a (near) optimal IDS. The suite includes `Framework` and `Simulator` (see Figure 3). A network administrator provides descriptions of network topology, sensor node characteristics and anticipated attacks to the `Framework`. It contains a database of available components which will be used to compose an IDS agent. To make automatic design of an IDS agent possible we should specify types of component that can be used and interfaces between them. The first step is to design a local audit collection module that will gather audit data that might be "ever" required by any detection module. We have undertaken the analysis of state-of-the-art IDSs for WSNs and possible audit data ever met in the studied literature have been described in Section 2. In the worst case, if some detection module needs audit data that are not gathered by the local audit collection module, the collection module should be designed in such way that an administrator will be able to add the required functionality easily.

There are different detection modules among the components in the database. Based on the specified attacks the `Framework` generates a possible configuration of the IDS agent. The configuration will be optimized using the `Simulator` in the following way. The `Framework` sets initial values of parameters and evaluates effectiveness of the configuration using metrics, examples of which we describe further in the section. Based on the evaluation the `Framework` "improves" the values of parameters and repeats the procedure until they become (near) optimal for given network topology, capabilities of sensor nodes and anticipated attacks. Should there are more than one possible configuration each of them is optimized separately. Evaluations of optimized configurations can be used by an administrator to choose the one that fits best its purposes.

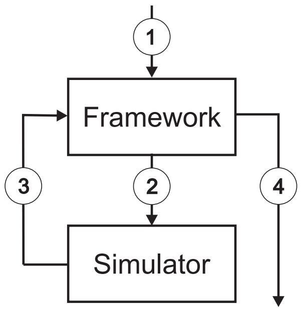


Fig. 3: An architecture of the proposed suite of tools. The arrow with number "1" depicts inputs provided by an administrator. The arrow "2" depicts a configuration that is passed to the `Simulator`. The arrow "3" depicts a feedback on effectiveness of the configuration based on evaluating metrics. The arrow "4" depicts an optimized configuration and its evaluation of effectiveness.

In order to understand how to evaluate an IDS we will firstly analyze attacks and their impacts on a network. Sinkhole attacks result in data receiving delays and additional energy usage because data do not travel along the shortest path. Let us assume that each node shares a cryptographic key with a base station as well as each node sends packet and waits for its acknowledgement. If a packet is modified along the path a base station may drop the packet and request to resend it again. That will cause a packet delay and additional usage of energy. If the packet is modified again and again it will not be ever delivered to the base station. Selective forwarding attack may result in sending the dropped packet again and again. Similarly as in the packet alternation attack, that can cause delays and additional energy usage. If a jamming attack lasts too long a packet will be dropped if a new packet arrives and buffer is full. *To sum up, the considered attacks may cause losses of packets, modifications of packets, delivery delays and additional energy usage.*

The main goal of IDS is to detect ongoing attacks and respond in such a way that the impact of the attacks will be minimal. The presented examples of metrics evaluate the impact of the sinkhole, selective forwarding, packet alternation and jamming attacks as well as an effectiveness of a given IDS. The smaller the measured impact is, the more effective IDS is. We assume that all of the metrics will be measured during a period of time of duration Q ($Q \in R^{+}$). The parameter Q will be chosen by an administrator.

1. *We propose to count the number of lost packets* as a difference between a number of packets sent by sensor nodes and a number of packets successfully received by a base station. The packets may get lost due to collisions, buffer overflows and environment changes. We assume that a number of packets lost in such way is constant for each time interval of duration Q if Q is long enough. It is noteworthy to mention that an increase of number of detected malicious nodes does not necessary mean that a number of successfully received packets will increase as well. Having a smaller number of nodes an attacker can change the strategy and drop/modify/jam more packets than before.

2. *We propose to count the number of modified packets received at the base station.* Packets that had been modified and hence were subsequently dropped are not counted as they are considered as lost. We can extend the metric by introducing a function that will determine how much the original packet differs from the modified one. An administrator will specify the function and thereby specify what packet fields are more critical than others.

3. *We propose to evaluate the total amount of energy used by the network* by summing up energy used by each sensor node. However, the metric does not take into account a distribution of energy usage – some sensor nodes may be depleted soon whereas others may remain fully charged. That may cause partition of a network and measurements from isolated regions of the network will never reach a base station. Such packets will be considered as lost. In order to avoid such situations we propose to use a metric that prioritizes IDSs which detect and respond to attacks in such way that energy consumption is distributed uniformly as much as possible. As an example of the metric we consider $\Theta = \sum_{i=1}^{n} c^{e_i}$, where n is a number of sensor nodes in a network, e_i is an amount of energy consumed by sensor node i, $(1 \leq i \leq n)$ and c $(c \in R, c > 1)$ is a constant that should be specified by an administrator.

4 Conclusions and further work

IDSs are useful for different networks since there are no guaranties that an attacker remains outside a perimeter secured by a firewall. In WSNs risks of being attacked by an insider are higher than in conventional networks since the area of their deployment is most often not physically protected. Due to a distributed nature of WSNs and severe limitations of sensor nodes on energy, memory and computation power, traditional IDSs are not applicable to the WSNs. Moreover, "one-fit-all" IDS is far away from being optimal for a given network topology, capabilities of sensor nodes and set of anticipated attacks in terms of detection accuracy and resources consumption. Therefore, the aim of our work was to propose a conceptual architecture of a suite of tools that will provide a network operator with a (near) optimal IDS for a given application. The optimalization process was divided into two steps. The first will save memory and energy by including into an IDS agent only modules that are used for detection of anticipated attacks. The second will find a trade-off between detection accuracy and resources consumption by setting parameters of detection modules and evaluating the configuration according to the defined metrics using a simulator. The proposed metrics evaluate impact of sinkhole, selective forwarding, packet alternation and jamming attacks by counting a number of lost/modified packets and energy consumed by a network. The smaller the measured impact is, the more effective an IDS is. The list of metrics is not complete and we currently extend it as well as add a classification of types of components that can be used to construct an IDS agent and define interfaces between them. We also plan to work on the selection of a proper optimization algorithm and a proper simulator. Since the space of possible solutions might be too large for exhaustive search, approximation algorithms might be used. Evaluation of the implemented suite of tools will be based on the time needed to find (near) optimal solution and on how close the obtained solution is to the optimal one.

5 Acknowledgement

This work was supported by the project 102/09/H042 "Mathematical and Engineering Approaches to Developing Reliable and Secure Concurrent and Distributed Computer Systems" of the Czech Science Foundation.

Also we would like to thank Petr Švenda for the fruitful discussions.

References

1. Krontiris, I., Dimitriou, T., Freiling, F. C.: Towards Intrusion Detection in Wireless Sensor Networks. In: 13th EuropeanWireless Conference. (2007)
2. Xu, W., Trappe, W., Zhang, Y., Wood, T.: The feasibility of launching and detecting jamming attacks in wireless networks. In: Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing. (2005) 46–57
3. Levis, P.: Packet Protocols. (http://www.tinyos.net/)
4. Levis, P.: CC2420 Radio Stack. (http://www.tinyos.net/, 2007)
5. Karlof, C., Wagner, D.: Secure routing in wireless sensor networks: Attacks and countermeasures. In: First IEEE International Workshop on Sensor Network Protocols and Applications. (2003) 113–127
6. Parno, B., Perrig, A., Gligor, V.: Distributed detection of node replication attacks in sensor networks. In: IEEE Symposium on Security and Privacy. (2005) 49–63
7. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. In: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems. (2000) 93–104
8. Roman, R., Alcaraz, C., Lopez, J.: A survey of cryptographic primitives and implementations for hardware-constrained sensor network nodes. In: Mobile Networks and Applications Journal, Vol. 12, Num. 4. (2007) 231–244
9. Roosta, T., Pai, S., Chen, P., Sastry, S., Wicker, S.: Inherent Security of Routing Protocols in Ad-Hoc and Sensor Networks. In: IEEE Global Telecommunications Conference. (2007) 1273–1278
10. Zhang, Y., Lee, W.: Intrusion detection in wireless ad-hoc networks. In: Proceedings of the 6th annual international conference on Mobile computing and networking. (2000) 275–283

Rewriting Systems over Nested Data Words

Invariance checking for systems with dynamic control and data structures

A. Bouajjani C. Drăgoi Y. Jurski M. Sighireanu
{abou,cezarad,jurski,sighirea}@liafa.jussieu.fr

LIAFA, University of Paris Diderot and CNRS, 75205 Paris 13, France

Abstract. We propose a generic framework for reasoning about infinite state systems handling data like integers, booleans etc. and having complex control structures. We consider that configurations of such systems are represented by nested data words, i.e., words of ... words over a potentially infinite data domain. We define a logic called NDWL allowing to reason about nested data words, and we define rewriting systems called NDW-RS over these nested structures. The rewriting systems are constrained by formulas in the logic specifying the rewriting positions as well as structure/data transformations. We define a fragment Σ_2^* of NDWL with a decidable satisfiability problem. Moreover, we show that the transition relation defined by rewriting systems with Σ_2^* constraints can be effectively defined in the same fragment. These results can be used in the automatization of verification problems such as inductive invariance checking and bounded reachability analysis. Our framework allows to reason about a wide range of concurrent systems including multithreaded programs (with procedure calls, thread creation, global/local variables over infinite data domains, locks, monitors, etc.), dynamic networks of timed systems, cache coherence/mutex/communication protocols, etc.

1 Introduction

Automated verification of modern software systems require reasoning about several complex features such as dynamic creation of concurrent threads, data manipulation, procedure calls, timing constraints, etc. For that, infinite-state models must be considered allowing to capture these features, and algorithmic techniques must be designed allowing to cope with these multiple sources of infinity in the state space.

We introduce in this paper a logic-based framework for reasoning about systems with composite (or nested) data structures such as multi-sets of integers, multi-dimensional arrays of integers, arrays of stacks or queues of integers. Nested data structures are also relevant when reasoning about systems with a complex control structure. For instance, the configuration of a program with dynamic thread creation and procedure calls can be naturally modeled as multi-sets of stacks over some potentially infinite domain of data (which can be themselves composite data structures).

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2356

We consider nested data words (NDW for short[1]) as formal objects for the representation of configurations of such systems with complex control and data. The domain of NDW is parameterized by the domain of (scalar) data used in these objects. We propose a logic, NDWL, for reasoning about such objects. The logic NDWL is parametrized by a data logic (a first order logic over the chosen data domain). NDWL allows to constrain the values of the data located at different nesting levels and at different positions in these nested data words. We consider a fragment of NDWL, called Σ_2^*, which consists of all formulas of the form $\exists^{\leq k}\forall^k\exists^{\leq k-1}\forall^{k-1}\ldots\exists^{\leq 1}\forall^1.\ \varphi$ where the quantifiers are on variables ranging over positions at different nesting levels of the structures, and φ is a formula (in the considered data logic) constraining the data attached to these positions. The satisfiability problem is decidable for Σ_2^* whenever the underlying logic on scalar data is. We show that this result can be used for checking automatically the invariance of assertions w.r.t. relations on NDW which are effectively represented in Σ_2^*.

Then, we introduce a class of rewriting systems over NDW, called NDW-RS. Each rewriting rule in a NDW-RS is constrained by formulas in NDWL specifying the rewriting positions and the structure/data transformation at these positions. We associate with each rewriting rule a NDWL formula characterizing the relation on NDWs induced by this rule. Therefore, we obtain a procedure for checking automatically invariance properties w.r.t. this class of models.

Finally, we show that our framework allows to deal with a large class of systems including distributed mutual exclusion protocols, cache coherence protocols, timed networks. In particular, we provide a systematic modeling for multithreaded programs with procedure calls and synchronization by monitors.

Related work. The verification of dynamic/parametrized networks of infinite-state processes has been addressed in several papers such as [2, 13, 10, 4, 11, 9, 8, 14]. All these works consider only one level nesting of data structures, i.e., collections (multisets, arrays, words) over infinite scalar data. Recently, [1, 15] propose a framework allowing two levels of nesting with a special form: N processes may have local arrays of integers of size exactly N. The verification approach used in these works is upper-approximate backward reachability analysis for a particular class of data constraints (gap order constraints on integers). Our work offers a larger framework for modeling and specification. On the other hand, while our framework allows for automatic inductive invariance checking, [13, 1, 4] allow for more automated verification of safety properties based on abstract analysis.

In comparison with [9, 8], this paper presents several significant and nontrivial extensions. First, we consider a more general framework where composite (nested) data structures can be handled. This allows to deal with classes of systems (such as multithreaded programs with infinite data and unbounded number of monitors/locks, etc.) which cannot be handled in the previous frameworks. Second, we consider here a more general class of rewriting systems (with mixed existential and universal rewriting strategies) allowing to model a larger class of communication and synchronization primitives. For instance, broadcast commu-

[1] NDW are not related with "nested words" in [3]

nication cannot be considered in the frameworks defined in our previous work, and the same holds for timing constraints (which require a global synchronization of the clocks).

Let us finally mention that logics on data words/trees have been proposed for reasoning about XML documents [6, 5, 12]. The considered logics and the obtained results in these works are not comparable with ours.

2 Motivation

We are interested in verifying automatically concurrent recursive programs with dynamic process creation, where the processes use data from infinite domains. The processes synchronize by monitors. The control is changed using sequential composition, conditionals, "while" loops, and procedures calls[2]. In the following we give an example of such a program.

Example The program, given in Fig. 1, is written in a Java-like syntax. An array M of monitors is accessed concurrently by threads created during the execution of the program. (The size of M changes by creating threads therefore monitors; the code for thread creation is omitted). Each thread has a unique identifier id$\geq$ 0 and has the task of creating a monitor and putting it in M at index id. All monitors have the same type, Moni, which has one procedure p, i.e., p shall be executed in mutual exclusion. The procedure chooses a number j strictly smaller than the identifier of its owning monitor and, if the monitor M[j] has been created, it calls its procedure. The property to check on such programs is the absence of deadlock due to the mutual waiting on the monitors. The inter-blocking of threads may appear if a thread i_1 has locked the monitor M[j_1] and it is waiting now to lock the monitor M[j_2], while, in parallel, a thread i_2 has locked the monitor M[j_2] and it is waiting now to lock the monitor M[j_1]. The absence of deadlock can be established by checking the invariance property that the call stacks of all threads are always sorted w.r.t. their integer values.

```
Vector<Moni> M = new Vector<Moni>();
monitor Moni {//Monitors definition
  int id;
  procedure p() {
    int j = value in [0,id);
    if (M.get(j)!=null)
       M.get(j).p();
}
thread T {//Threads definition
  int id;
  procedure run() {
    M.set(id, new Moni(id));
    M.get(id).p();
  }
}
```

Fig. 1. Example of program.

Representing program configurations by nested data words The configurations of the program are given by the configuration of the vector M and the configuration of threads, where each of these threads has an attached unbounded call stack, and each of these stacks contains values over the infinite domain of integers (corresponding to the values of variables id and j). We represent the threads configuration as words where each position denotes a thread. Therefore, each

[2] We do not allow pointer manipulation.

position has attached an integer data and a subword over integers, denoting the identity of the process and its call stack. The vector elements are distinguished positions in this nested data word, that have attached only the identity of the monitor and an empty subword. This structure of word of words of ... words we called it *nested data words* (NDW) over a potentially infinite data domain. Let Σ be a finite alphabet, and let $\mathbb{D}$ be a (infinite) *data domain*. The nested data words domain NDW is the union of the family $\{\mathsf{NDW}_k\}_{k\geq 0}$ where (i) for $k \geq 1$, NDW_k contains all sequences indexed by subsets of $\mathbb{N}$ with values in $\Sigma \times \mathbb{D} \times \mathsf{NDW}_{k-1}$, i.e., $\mathsf{NDW}_k = \{w \mid w : \mathbb{N} \rightharpoonup \Sigma \times \mathbb{D} \times \mathsf{NDW}_{k-1}\}$, (ii) NDW_0 contains only the empty word, denoted ϵ, $\epsilon(i)$ is undefined for all $i \in \mathbb{N}$.

The elements of NDW_k are called *nested data words of level k*. Since any $w \in \mathsf{NDW}$ is a partial function we denote by $dom(w)$ the subset of $\mathbb{N}$ where w is defined. We call *indexes* the natural numbers in the domain of $w \in \mathsf{NDW}$; their level is given by the level of the word they index.

Given a word w in NDW_k ($k > 1$) and $p \in dom(w)$, then $label(w[p])$ (resp. $data(w[p])$, $ndw(w[p])$) denotes the first (resp. second, third) member of the tuple $w[p]$. These notations extend to sequences of indexes, e.g., $label(w[p_1, \ldots, p_j])$ ($1 < j \leq k$) denotes the label attached to index p_j of the inner subword $ndw(w[p_1, \ldots, p_{j-1}])$ ($label(w[p_1, \ldots, p_j]) = label(ndw(w[p_1, \ldots, p_{j-1}])[p_j])$).

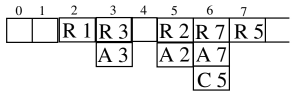


Fig. 2. Element of NDW_2.

Fig. 2 provides an example of a nested data word $w \in \mathsf{NDW}_2$ built on the finite alphabet $\Sigma = \{R, A, B, C, D\}$ and the data domain $\mathbb{D} = \mathbb{N}$. This word is a simplified representation for the configurations of the program in Section 1 (the value of j is omitted and also the elements of the array M are omitted). The domain of w is $\{2, 3, 5, 6, 7\}$, i.e., there are five created threads. The labels A,B,C, and D denote the control points at line 14, 8, 7, 4 respectively. These control points are important because they correspond to calls of /returns from the procedure p. Notice that, $label(w[6, 0]) = A$, $data(w[6]) = 7$ and $ndw(w[6]) = [0 \mapsto (A, 7), 1 \mapsto (C, 5)]$ (the integer numbers in $ndw(w[6])$ are the identities of the monitors locked by the thread with the identity $data(w[6]) = 7$).

Reasoning about programs To prove safety properties (e.g., the absence of deadlock) we use invariant checking. Given a set of initial configurations $Init$, a set of safe configurations $Safe$ and a set of configurations Inv, we have to check that Inv is an inductive invariant and that $Inv \subseteq Safe$. Inv is an inductive invariant if (1) $Init \subseteq Inv$, and (2) for every statement st of the program, $post(st, Inv) \subseteq Inv$, where $post(st, Inv)$ denotes the set of configurations obtained by executing st on Inv.

We give a logical framework to specify properties of program configurations and transformations between configurations. We define a multi-sorted second order logic called *nested data word logic*, NDWL. Sets of configurations, like $Init, Inv$, and $Safe$ are modeled by formulas in NDWL, $\varphi_{Init}, \varphi_{Inv}$, resp. φ_{Safe} and the relation between configurations defined by $post(st, \bullet)$ is a formula $\varphi_{post(st)}(\gamma, \gamma')$ where γ and γ' represent the configuration before resp. after the execution of statement st. Then, a formula φ_{Inv} is an *inductive invariant* if

(1) $\varphi_{Init}(\gamma) \wedge \neg\varphi_{Inv}(\gamma)$ is unsatisfiable and (2) for each program statement st, $\varphi_{Inv}(\gamma) \wedge \varphi_{post(st)}(\gamma, \gamma') \wedge \neg\varphi_{Inv}(\gamma')$ is unsatisfiable.

Formulas in NDWL can specify, properties of the global variables and configurations of processes. For example, using NDWL one can specify properties on the call stack of some process, relations between two call stacks, or relations between global and local variables.

3 Nested data word logic NDWL

The logic NDWL is parameterized by a (first-order) logic $\mathsf{FO}(\mathbb{D}, \mathbb{O}, \mathbb{P})$ on the considered data domain $\mathbb{D}$, i.e., by the set of operations $\mathbb{O}$ and by the set of basic predicates (relations) $\mathbb{P}$ allowed on elements of $\mathbb{D}$.

Syntax Consider the following pairwise disjoint sets of variables: (1) D (of elements denoted by $b, c, d, \dots$) is the set of *data variables* taking values in $\mathbb{D}$, (2) Γ (of elements denoted by $\gamma, \gamma', \gamma_1, \dots$) is the set of *nested data word variables* taking values in NDW, (3) I (of elements denoted by $x, y, \dots$) is the set of *index variables* taking values in $\mathbb{N}$, and (4) $\mathcal{I}$ (of elements denoted by $X, Y, \dots$) is the set of *index-set variables* taking values in $2^{\mathbb{N}}$.

Additionally, each variable, which is not a data variable, is indexed by a number from 1 to N called *the level* of the variable. These levels define a partition on $\Gamma, I, \mathcal{I}$: $\Gamma = \bigcup_{1 \le k \le N} \Gamma_k$, resp. $I = \bigcup_{1 \le k \le N} I_k$, $\mathcal{I} = \bigcup_{1 \le k \le N} \mathcal{I}_k$ where Γ_k (resp. I_k, $\mathcal{I}_k$) is the set of nested data word (resp. index, index-set) variables of level k. The syntax of *terms* and *formulas* in NDWL is given by the following grammars:

$$\begin{array}{rl} t ::= & d \mid o(t_1, \dots, t_m) \mid \upsilon(\mathit{tt}[x_0, \dots, x_p]) \qquad \mathit{tt} ::= \gamma \mid \delta(\mathit{tt}[x_0, \dots, x_p]) \\ \varphi ::= & \mathtt{true} \mid r(t_1, \dots, t_m) \mid A(\mathit{tt}[x_0, \dots, x_p]) \mid 0 < x \mid x < x' \mid x \in X \mid \mathsf{idx}(x, \mathit{tt}) \\ & \mid \exists x.\ \varphi \mid \exists d.\ \varphi \mid \neg\varphi \mid \varphi \vee \varphi \end{array}$$

where o is an operation in $\mathbb{O}$ of arity $m \ge 0$, t is a data term and tt is a nested data word term (ndw-term for short), $x, x', x_0, \dots, x_p, (p \ge 0)$ are in I, r is a predicate in $\mathbb{P}$ of arity m, $A \in \Sigma$, 0 is a constant index, $X \in \mathcal{I}$, $d \in D$, and $\gamma \in \Gamma$. The elements generated by this grammar respect the following level constraints: (1) x and x' have the same level in $x < x'$ (2) x and tt have the same level in $\mathsf{idx}(x, \mathit{tt})$ (3) x and X have the same level in $x \in X$ (4) if tt is a ndw-term of level k then $\delta(\mathit{tt}[x_0, \dots, x_p])$ (resp. $\upsilon(\mathit{tt}[x_0, \dots, x_p])$) is a ndw-term of level $k - p - 1$ (resp. a data term) and then x_0, x_1, ... , x_p have levels k, $k-1$, ... , $k-p$ and $k - p \ge 1$; (5) $t_1, \dots, t_m$ are data terms, i.e., they have level 0, in $r(t_1, \dots, t_m)$ and in $o(t_1, \dots, t_m)$; (6) if tt is a ndw-term of level k in $A(\mathit{tt}[x_0, \dots, x_p])$ then $k - p \ge 1$ and x_0, x_1, ... , x_p have levels k, $k-1$, ... , $k-p$.

As usual, conjunction ($\wedge$), implication ($\Rightarrow$), and universal quantification ($\forall$) can be defined in terms of $\neg$, $\vee$, and $\exists$. We also define equality ($=$), disequality ($\neq$) and inequality ($\le$) in terms of $<$ and boolean connectives. To emphasize the level of some quantified variable, we use notations $\exists^k$ (resp. $\forall^k$) instead of $\exists$ (resp. $\forall$).

Notice that the variables in Γ and $\mathcal{I}$ are free in any NDWL formula. We assume w.l.o.g. that in every formula, each variable is quantified at most once.

Semantics Formally, a model of a NDWL formula is a mapping $\mathcal{M} : \mathbb{N}^* \to 2^{\mathbb{N}}$ which gives for each level k the set of positions defined within the model (notation $\mathcal{M}_k = \mathcal{M}(k)$) and valuations of free variables. A valuation of index variables is a partial mapping $\rho \in [I \rightharpoonup \mathbb{N}]$ s.t. variables in I_k take values in $\mathcal{M}_k$. We extend ρ by $\rho(0) = 0$. A valuation of index-set variables is a partial mapping $\nu \in [\mathcal{I} \rightharpoonup 2^{\mathbb{N}}]$ s.t. for any variable $X \in \mathcal{I}_k$, $\nu(X) \subseteq \mathcal{M}_k$. A valuation of data variables is a partial mapping $\beta \in [D \rightharpoonup \mathbb{D}]$. A valuation of NDW variables is a partial mapping $\theta \in [\Gamma \rightharpoonup \mathsf{NDW}]$ s.t. variables in Γ_k take values in NDW_k. Moreover, for all variables $\gamma \in \Gamma_k$, $dom(\theta(\gamma)) \subseteq \mathcal{M}_k$, and so on recursively, i.e., for any subword of $\theta(\gamma)$, $w \in \mathsf{NDW}_\ell$ with $1 \leq \ell < k$, $dom(w) \subseteq \mathcal{M}_\ell$.

The data terms t are interpreted into values in $\mathbb{D}$; they denote the values stored at different positions of some (inner) word. The ndw-terms tt are interpreted into nested data words of the corresponding level. Formulas in NDWL can express ordering relations between indexes ($x < x'$) and the membership relation between an index and an index-set ($x \in X$) or an index and the definition domain of a nested data word ($\mathsf{idx}(x, tt)$). Intuitively, $A(tt[x_0, \ldots, x_p])$ says that tt is interpreted into some word w and the indexes $x_0, \ldots, x_p$ form a valid path to an inner word of w s.t. $label(w[x_0, \ldots, x_p]) = A$. The concept of valid path refers to the fact the x_0 must be defined in w, $x_1 \in dom(ndw(w[x_0]))$ and so forth x_p must be defined in $ndw(w[x_0, \ldots, x_{p-1}])$.

In general, a term $\delta(tt[x_0, \ldots, x_p])$ or $\upsilon(tt[x_0, \ldots, x_p])$ is *well defined* if $x_0, \ldots, x_p$ forms a valid path to an inner word of the word tt interprets into. In the following we consider that γ interprets into w the nested data word pictured in Fig. 2. Then, when $x = 6$ the term $\delta(\gamma[x])$ denotes $ndw(w[6])$ and $\upsilon(\gamma[x])$ interprets into 7. The atomic formulas built over non well defined terms are false. This fact might induce some difficulties when reasoning about nested structures. E.g., the formula $\forall^2 y \exists^1 x.\ \neg A(\gamma[y, x])$ saying that any subword of γ shall have an index not labeled by A, has a model in which γ is interpreted to w. This happens even if all defined positions in $ndw(w[2])$ are labeled by A; in this case, a value for x that satisfies the property is an index of $\mathcal{M}_1$ not defined in $ndw(w[2])$, e.g., $1 \in \mathcal{M}_1$ since $1 \in dom(ndw(w[6]))$ but $1 \notin ndw(w[2])$. To obtain the intuition and reject this model, we must specify that only indexes y in the domain of w are considered: $\forall^2 y \exists^1 x.\ \mathsf{idx}(y, \gamma) \Longrightarrow \mathsf{idx}(x, \gamma[y]) \wedge \neg A(\gamma[y, x])$.

Examples In the following, we consider that γ is interpreted to w, the nested data word w in Fig 2. Then, the following formula states that all threads in the configuration γ are running and their identity is smaller than 10: $\forall^2 y.\ \mathsf{idx}(y, \gamma) \Longrightarrow R(\gamma[y]) \wedge \upsilon(\gamma[y]) \leq 10$. The formula $\forall^2 y \exists^1 x.\ \mathsf{idx}(y, \gamma) \Longrightarrow \mathsf{idx}(x, \gamma[y]) \wedge \upsilon(\gamma[y, x]) \geq 2$ says that all the inner words of w have an index whose data is at least 2. Finally, the next formula says that all the threads in the configuration denoted by γ have their call stack (represented by the inner word) sorted w.r.t. the identity of the owned monitors: $\forall^2 z \forall^1 x, y.\ (x < y \wedge \mathsf{idx}(z, \gamma) \wedge \mathsf{idx}(x, \gamma[z]) \wedge \mathsf{idx}(y, \gamma[z])) \Longrightarrow \upsilon(\gamma[z, x]) > \upsilon(\gamma[z, y])$.

Syntactical forms and fragments A formula is in *prenex normal form* (PNF) if it is of the form $Q_1 z_1 Q_2 z_2 \ldots Q_m z_m.\ \varphi$ where (1) $Q_1, \ldots, Q_m \in \{\exists, \forall\}$, (2)

$z_1, \ldots, z_m \in I \cup D$, and (3) φ is a quantifier-free formula. It can be proved that for every formula φ in NDWL, there exists an equivalent formula in PNF.

We consider $\{\Sigma_2^\ell\}_{\ell \geq 0}$ and $\{\Theta_1^\ell\}_{\ell \geq 0}$ two fragments of NDWL defined by restricting the quantifier alternation over index variables of the same level in PNF formulas. We define $\Sigma_2^* = \bigcup_{\ell \geq 0} \Sigma_2^\ell$ and $\Theta_1^* = \bigcup_{\ell \geq 0} \Theta_1^\ell$ where

$$\Sigma_2^\ell = \{Q_l \ldots Q_2 Q_1.\ \varphi \mid Q_i = \exists^{\leq i} \overrightarrow{x}_i \exists \overrightarrow{d}_i \forall^i \overrightarrow{y}_i,\ 1 \leq i \leq l, \text{ and } \varphi \text{ quantifies over } D\}$$
$$\Theta_1^\ell = \{S_l \ldots S_1.\ \phi \mid S_i = \exists^i \overrightarrow{x} \text{ or } S_i = \forall^i \overrightarrow{x},\ 1 \leq i \leq l, \text{ and } \phi \text{ quantifies over } D\}$$

Notice that, Θ_1^* is a subset of Σ_2^* which is closed under all boolean operations while Σ_2^* is closed only under disjunction and conjunction. All the formulas given as example above are in the Θ_1^* fragment.

4 Application to verification

The satisfiability problem for the full NDWL is not decidable. E.g., [8] proves the undecidability of this problem for a subfragment of NDWL, which allows $\forall^*\exists^*$ quantification over variables of the same level. The next theorem provides a positive result for a fragment of NDWL.

Theorem 1. *Whenever the data logic* $\mathsf{FO}(\mathbb{D}, \mathbb{O}, \mathbb{P})$ *has a decidable satisfiability problem, the satisfiability problem of the fragment* Σ_2^* *of NDWL is also decidable.*

The proof of this theorem is similar to the decidability proof for CSL logic in [7]. The proof gives a decision procedure whose complexity is NP when the number of universally quantified variables is fixed. Moreover, the structure of the nested data words is simpler than the one of the heap graphs in CSL which leads to a more efficient implementation for the decision procedure. The decidability result in Theorem 1 is used to automate invariant checking.

Theorem 2. *Checking that a formula* $\varphi_{Inv} \in \Theta_1^*$ *is an inductive invariant is decidable for specifications with the transition relation* $\varphi_{post(st)}$ *in* Σ_2^* *for every program statement st, when the underlying logic,* $\mathsf{FO}(\mathbb{D}, \mathbb{O}, \mathbb{P})$*, has a decidable satisfiability problem.*

In the next section, we introduce a formalism to specify the transition relation of a system, i.e., φ_{post}, with formulas in Σ_2^*. This formalism handles a large class of complex systems with interesting control structures like rendez-vous, broadcast, procedure call, process creation, locks.

5 Rewriting systems over nested data words

A *nested data words rewriting system* (NDW-RS for short) is a pair $RS = (\Sigma, \Delta)$ where Σ is a finite set of labels, and Δ is a finite set of rewriting rules. Each rule may be an *existential rule*, an *universal rule*, or a general (mixed existential and universal) rule. In the following, we give the syntax and the intuitive semantics of the rewriting rules.
The existential rules have the following syntax:

$$\overrightarrow{A} \hookrightarrow_\sharp \overrightarrow{B} : \varphi_g \ / \ \varphi_a \qquad (1)$$

where $\overrightarrow{A}$ and $\overrightarrow{B}$ are labels in Σ, $\sharp$ is the *rewriting policy* that can be one of multiset ($\sharp = m$), factor ($\sharp = f$), or suffix ($\sharp = s$), and φ_g and φ_a are NDWL formulas.

An existential rule selects the nested data word rewritten using the guard φ_g. On this word, the rule rewrites according to the policy the indexes labeled by $\overrightarrow{A}$ and constrained using φ_g into indexes labeled by $\overrightarrow{B}$ whose data are assigned using φ_a. In φ_g and φ_a, the path to the rewritten word, the indexes labeled by $\overrightarrow{A}$, and those labeled by $\overrightarrow{B}$ are denoted using variables $\overrightarrow{\xi}$, $\overrightarrow{x}$, resp. $\overrightarrow{y}$. Also, the initial (resp. resulting) nested data word is denoted by the NDW variable γ (resp. γ'). Fig. 3 (1) gives an example of a thread creation in the configuration given in Fig. 2 modeled by the existential rule $R1$ below:

$$\text{R1}: \ R \hookrightarrow_m R\ R \ : \ v(\gamma[x_1]) \geq 2 \ / \ v(\gamma'[y_1]) = v(\gamma[x_1]) \wedge \delta(\gamma'[y_1]) = \delta(\gamma[x_1]) \wedge v(\gamma'[y_2]) = 2v(\gamma[x_1])$$

Intuitively, a thread with the identity not smaller than 2 spawns a new thread with the identity doubled. Formally, the rule rewrites γ at an index x_1 labeled by R that stores a value $v(\gamma[x_1]) \geq 2$. The rewriting introduces two positions labeled by R: y_1 is a copy of x_1 and y_2 has attached an integer with the data twice the value of $v(\gamma[x_1])$ and an empty sub-word (in Fig. 3 (1) a thread with the identity 4 is spawned).

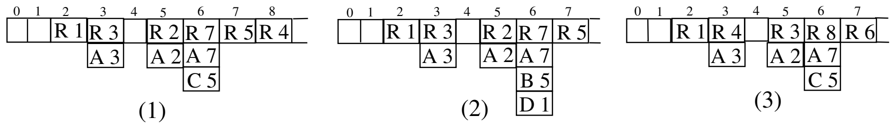


Fig. 3. Applying R1, R2 resp. R3 on the word of Fig. 2.

Note that more than one choice is possible for the indexes that are rewritten, i.e., $\overrightarrow{x}$ and $\overrightarrow{y}$. The rewriting policy refines the selection of these indexes. For example, the multiset policy puts no ordering relation between indexes in $\overrightarrow{x}$ (resp. $\overrightarrow{y}$); it says that rewriting concerns $(min(|\overrightarrow{x}|, |\overrightarrow{y}|))$ positions from γ satisfying φ_g plus some added (resp. removed) positions that were not defined in the initial (resp. resulting) word. For example, in the rule R1 $y_2 \notin dom(\gamma)$ but it is defined in γ', $y_2 \in dom(\gamma')$.

The suffix rewriting policy says that $\overrightarrow{x}$ (and $\overrightarrow{y}$) are the last $|\overrightarrow{x}|$, (resp. $|\overrightarrow{y}|$) consecutive positions of the subword rewritten. The call of the procedure p at line 7 in Fig.1 on a monitor with the identity 1 in a thread with the identity 7 is modeled by an existential rewriting rule with suffix rewriting policy:

$$\text{R2}: \ C \hookrightarrow_s B\ D \ : \ R(\gamma[\xi]) \wedge v(\gamma[\xi]) = 7 / v(\gamma[\xi, x_1]) = v(\gamma'[\xi, y_1]) \wedge v(\gamma'[\xi, y_2]) = 1$$

where C is the control point of the process at the call of p, B is the return point after the call of p, and D is the entry control point of p; the process calling p is selected using φ_g, the local data of the new position (labeled by D) is initialised using φ_a. Existential rewriting rules can model communication by shared, global variables or rendez-vous.

The universal rules have the following syntax:

$$\overrightarrow{C} \mapsto \overrightarrow{D} : \psi_g \;/\; \psi_a \tag{2}$$

where $\overrightarrow{C}$ and $\overrightarrow{D}$ are labels in Σ and ψ_g and ψ_a are NDWL formulas.

A universal rule rewrites *all* indexes labeled by $\overrightarrow{C}$ and satisfying the guard ψ_g by replacing their label with the respective label in $\overrightarrow{D}$ and their data with the data assigned in ψ_a. To refer the positions rewritten we use the set of variables $\overrightarrow{u}$ ($|\overrightarrow{u}| = |\overrightarrow{C}| = |\overrightarrow{D}|$). The formulas ψ_g and ψ_a contain as free variables $\overrightarrow{\xi}$, γ, and γ' with the same semantics as in existential rules. Fig. 3 (3) shows the nested data word resulting by applying the universal rule below on the word of Fig. 2:

$$\text{R3}: \quad R \mapsto R \;:\; \upsilon(\gamma[u]) \geq 2 \;/\; \upsilon(\gamma'[u]) = \upsilon(\gamma[u]) + 1$$

The rule increments the identity of all threads having the identity greater than 2. In this way it will be possible to create later a thread with the identity 2.

The mixed rules combine an existential and an universal rule as follows:

$$\overrightarrow{A} \hookrightarrow_\sharp \overrightarrow{B} \;:\; \varphi_g \;/\; \varphi_a \quad | \quad \overrightarrow{C} \mapsto \overrightarrow{D} \;:\; \psi_g \;/\; \psi_a \tag{3}$$

The subword of γ rewritten (given by $\overrightarrow{\xi}$) is fixed in φ_g and ψ_g. The indexes rewritten by the existential part ($\overrightarrow{x}$ and $\overrightarrow{y}$) may be used in ψ_g and ψ_a to choose the indexes $\overrightarrow{u}$ and their new data, i.e., all the four formulas $\varphi_g, \varphi_a, \psi_g, \psi_a$ share the same index variables in $\overrightarrow{\xi}$, $\overrightarrow{x}$, and $\overrightarrow{y}$. These rules model statements like *notifyAll*(), in synchronization by monitors, or time elapsing in networks where processes manipulate clocks.

Formally, the semantics of rewriting any rule R is given by NDWL formulas denoted $\texttt{reach}_R(\gamma, \gamma')$, where γ denotes the word to be rewritten (it satisfies the guard φ_g/ψ_g) and γ' denotes the word after the rewriting.

We denote by NDW-RS$[\Sigma_2^*]$ the class of NDW-RS where any rewriting rule has the constraints φ_g and φ_a in Σ_2^*, ψ_g and ψ_a in Θ_1^k.

Proposition 1. *For every mixted rule of a rewriting system in NDW-RS$[\Sigma_2^*]$, the associated NDWL formula is in the fragment Σ_2^*.*

Then, the following theorem is a consequence of the results given in Section 4.

Theorem 3. *Checking that a formula $\varphi \in \Theta_1^*$ is an inductive invariant is decidable for any system in NDW-RS$[\Sigma_2^*]$, if the underlying logic* $\mathsf{FO}(\mathbb{D}, \mathbb{O}, \mathbb{P})$ *has a decidable satisfiability problem.*

Notice that all the examples of rewriting rules given in this section belong to a system in NDW-RS$[\Sigma_2^*]$.

6 Conclusion

We have defined a generic framework for reasoning about unbounded networks of processes with complex data and control structures. Various instances of this framework allow to deal in a uniform way with important classes of system models such as dynamic networks of processes with counters, clocks, unbounded/parametric structures (arrays, stacks, queues) over infinite data domains, etc. This is based on generic decidability and closure results for a (useful fragment of a) logic for specifying configurations of such networks as nested data words. Several classes of actions in such networks can be modeled in this logical framework. For example, the process creation, the procedure calls and the *rendez-vous* are modeled by existential rewriting rules while global synchronization between processes (or *broadcast*) is modeled using universal and mixed rewriting rules.

Future work includes extending our framework by developing techniques for compositional verification of concurrent programs.

References

1. P.A. Abdulla, G. Delzanno, and A. Rezine. Parameterized verification of infinite-state processes with global conditions. In *Proc. of CAV*, volume 4590 of *LNCS*, pages 145–157, 2007.
2. P.A. Abdulla and B. Jonsson. Verifying networks of timed processes (extended abstract). In *Proc. of TACAS*, volume 1384 of *LNCS*, pages 298–312, 1998.
3. R. Alur and P. Madhusudan. Adding nesting structure to words. *J.ACM*, 56(3), 2009.
4. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L.D. Zuck. Parameterized Verification with Automatically Computed Inductive Assertions. In *Proc. of CAV*, volume 2102 of *LNCS*, pages 221–234, 2001.
5. M. Bojanczyk, C. David, A. Muscholl, Th. Schwentick, and L. Segoufin. Two-variable logic on data trees and XML reasoning. In *Proc. of PODS*, pages 10–19. ACM, 2006.
6. M. Bojanczyk, A. Muscholl, Th. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *Proc. of LICS*, pages 7–16. IEEE, 2006.
7. A. Bouajjani, C. Drăgoi, C. Enea, and M. Sighireanu. A logic-based framework for reasoning about composite data structures. In *Proc. of CONCUR*, volume 5710 of *LNCS*, pages 178–195, 2009.
8. A. Bouajjani, P. Habermehl, Y. Jurski, and M. Sighireanu. Rewriting Systems with Data. In *Proc. of FCT*, volume 4639 of *LNCS*, pages 1–22, 2007.
9. A. Bouajjani, Y. Jurski, and M. Sighireanu. A generic framework for reasoning about dynamic networks of infinite-state processes. In *Proc. of TACAS*, volume 4424 of *LNCS*, pages 690–705, 2007.
10. M. Bozzano and G. Delzanno. Beyond Parameterized Verification. In *Proc. of TACAS*, volume 2280 of *LNCS*, pages 221–235, 2002.
11. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Proc. of VMCAI*, volume 3855 of *LNCS*, pages 427–442, 2006.
12. C. David. Complexity of data tree patterns over xml documents. In *Proc. of MFCS*, volume 5162 of *LNCS*, pages 278–289, 2008.
13. G. Delzanno. An assertional language for the verification of systems parametric in several dimensions. *Electr. Notes Theor. Comput. Sci.*, 50(4), 2001.
14. C. Ihlemann, S. Jacobs, and V. Sofronie-Stokkermans. On local reasoning in verification. In *Proc. of TACAS*, volume 4963 of *LNCS*, pages 265–281, 2008.
15. A. Rezine. *Parameterized Systems: Generalizing and Simplifying Automatic Verification.* PhD thesis, University of Uppsala, 2008.

Space Effective Model Checking for Component-Interaction Automata

Nikola Beneš[*], Milan Křivánek[**], and Filip Štefaňák[**]

Faculty of Informatics, Masaryk University
Botanická 68a, 602 00 Brno, Czech Republic
`{xbenes3,xkrivan8,xstefan5}@fi.muni.cz`

Abstract. The techniques of component-based development are becoming a common practice in the area of software engineering. One of the crucial issues in the correctness of such systems is the correct interaction among the components. The formalism of component-interaction automata was devised to model various aspects of such interaction, as well as to allow automated verification in the form of model checking with properties expressed in the component-interaction LTL, a variant of the known linear temporal logic. As the state space of a component-based system can grow exponentially with the number of components, it is desirable to employ reduction techniques to make the verification task more feasible. In our work, we describe the implementation of the ample set partial order reduction method within the component-interaction automata verification framework. Due to the state and action-based nature of both the modelling and specification formalisms, the implementation differs from traditional state-based approaches. After describing the implementation, we present some of the results obtained by employing the enhanced verification framework on a case study.

1 Introduction

The demand to shorten the time necessary to develop complex software and to lower its costs encourages employment of new software development techniques. One of such techniques is the component-based development, which builds software systems out of prefabricated autonomous components that are often developed without any knowledge of their deployment context. Therefore a great deal of attention must be paid to their interaction, since correct interaction of the components plays an important role in the correctness of the system as a whole.

Component-interaction automata [1] represent a formalism designed for specification of component-based systems. Such models can be used to verify desirable properties of the system expressed as formulae of a suitable logic. CoIn verification environment [2] is based on DiVinE verification framework [3] and allows model checking of these specifications. The properties are formalized using a variant of state/event LTL [4] that is better suited for component-based

[*] The author has been supported by Grant Agency project no. GD102/09/H042.
[**] The authors have been supported by Academy of Sciences grant no. 1ET400300504.

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2354

systems than pure state-based logics, as we are interested in both the state of the components and their communication.

The verification tool has to cope with exponential growth of the state space that is commonly caused by interleaving of independent actions. Ample set partial-order reduction [5] is one of the state space reduction techniques, which tries to identify redundant states and omit their generation while preserving verification properties of the model.

Traditionally, partial order reduction has been used in connection with state-based models. In [6], we have shown how partial order reduction can be performed on state/event-based models. The goal of this work is to implement this reduction method into the CoIn verification environment, which is an example of such formalism. In order to do that, we have to find effective heuristics to check the conditions for ample sets different from those used in traditional state-based approaches.

2 Foundations

Modelling and Specification Formalisms We start with describing the component-interaction automata formalism [1]. Each component is modelled as a finite labelled transition system equipped with an additional structure on labels and a hierarchy of names representing the architectural structure.

A *component-interaction automaton* (CI automaton for short) is a 5-tuple $\mathcal{C} = (Q, A, \delta, Q_0, H)$ where Q is a finite set of *states*, A is a finite set of *actions*, $\Sigma = ((S_H \cup \{-\}) \times A \times (S_H \cup \{-\})) \setminus (\{-\} \times A \times \{-\})$ is a set of *labels*, $\delta \subseteq Q \times \Sigma \times Q$ is a finite set of *labelled transitions*, $Q_0 \subseteq Q$ is a nonempty set of *initial states*, and H is a hierarchy of component names where the set of component names is denoted by S_H.

The semantics of the labels is input, output, or internal, based on their structure. In the triple, the middle element represents an action name, the first element represents the name of the component that outputs the action, and the third element represents the name of the component that inputs the action.

The CI automata can be composed together using a parametrized composition operator $\otimes^{\mathcal{F}}$. Given a set of *feasible labels* $\mathcal{F}$ and a set of CI automata, the result of the operation is a product automaton with only labels from $\mathcal{F}$ allowed. In the product, the components cooperate either by interleaving of their original transitions, or by simultaneous execution of two complementary transitions (with labels $(n_1, a, -)$, $(-, a, n_2)$) which is represented by a new internal transition (with label (n_1, a, n_2)).

As for the property specification logic, we use a variant of the state/event LTL [4, 6], which is an extension of LTL for reasoning about both properties of states and actions. Currently, the only state atomic propositions we consider are the *enabledness properties*, $Ap = \{\mathcal{E}(l) \mid l \in \Sigma\}$. We say that a state satisfies the property $\mathcal{E}(l)$ if an outgoing transition with label l is enabled in that state. We define a function $\mathcal{L} : Q \to 2^{Ap}$ as $\mathcal{L}(q) = \{\mathcal{E}(l) \mid q \text{ satisfies } \mathcal{E}(l)\}$.

Partial Order Reduction Technique Our approach follows the ample set partial order reduction technique as presented in [7]. The basic idea is to view the verified system as a state transition system in which some of the transition are invisible, and to reduce the system such that all original behaviour is preserved with respect to the ordering of visible transitions.

A *state transition system* is a triple (S, T, S_0) where S is a set of states, S_0 is a nonempty set of initial states and T is a set of transitions such that for each $\alpha \in T$, $\alpha \subseteq S \times S$. Furthermore, for each $\alpha \in T$ and for each state $s \in S$ there is at most one $s' \in S$ such that $(s, s') \in \alpha$. We also write $\alpha(s) = s'$.

In the traditional state-based approach, the *invisible* transitions are those that do not change the state atomic propositions. In the state/event-based approach [6], each transition is further equipped with an action. The property to be verified is then supplied with a set of *interesting actions* Act', and the invisible transitions are those with non-interesting actions that do not change the state atomic propositions.

Two transitions α and β are said to be *independent* if whenever α and β are enabled in s, then also α is enabled in $\beta(s)$, and $\alpha(\beta(s)) = \beta(\alpha(s))$ for all s.

While exploring the state space of the system, the ample set method works by selecting only a subset of outgoing transitions from each state. The original set of outgoing transitions from state s is denoted by $\mathit{enabled}(s)$, the selected subset is denoted by $\mathit{ample}(s)$. To ensure that the reduction is correct, the following four conditions must hold.

C0 - nonemptiness $\mathit{ample}(s) = \emptyset$ if and only if $\mathit{enabled}(s) = \emptyset$.
C1 - dependency Along every path in the full state graph that starts at s, a transition that is dependent on a transition in $\mathit{ample}(s)$ cannot be executed without a transition from $\mathit{ample}(s)$ occurring first.
C2 - invisibility If $\mathit{enabled}(s) \neq \mathit{ample}(s)$ then every $\alpha \in \mathit{ample}(s)$ is invisible.
C3 - cycle A cycle is not allowed if it contains a state in which some transition α is enabled, but is never included in $\mathit{ample}(s)$ for any state s on the cycle.

POR and CI automata In our setting, the system consists of a finite set of *simple* CI automata (numbered 1, …, n) whose state space is described explicitly, composed in a hierarchical way. The hierarchical composition is represented by a number of *composite* CI automata. The hierarchy can be thus represented with a tree, the leaves being the simple automata and the root being the CI automaton representing the whole system.

The (implicit) translation into a state transition systems then works as follows. The states are n-tuples $(s_1, \ldots, s_n)$ of the states of the simple automata. The transitions are then of two kinds: those that represent the progression of only one of the simple automata (*simple transitions*), and those that represent a synchronization of two simple automata (*sync transitions*). The simple transitions can be identified with tuples of the form $\langle i, s_i, s'_i, l\rangle$ and the sync transitions with tuples of the form $\langle i, j, s_i, s'_i, s_j, s'_j, l\rangle$, where i, j are automata numbers, s_i, s'_i, s_j, s'_j their respective states and l is the transition label.

3 Heuristics and Implementation

Overapproximations of the ample set conditions Some of the original ample set conditions are difficult to check, especially given that we want to check them in each state as we build the composite transition system. Therefore we use an overapproximation of these properties, using modified heuristics from [7]. We keep the condition for C3, but use modified versions for C1 and C2.

Firstly, we have to address the issue of selecting the candidates for ample sets, as the result helps us to create more elegant overapproximations. Usually, transitions of a single simple automaton are dependent on each other. Therefore we use the obvious solution of considering ample sets, which for each automaton consist of either all its transitions or none. This approach may not be feasible, though, because the number of subsets of all automata is exponential.

Exploiting the hierarchical structure To further reduce the number of possible candidates, we take advantage of the tree structure of the composition and for ample sets consider only the transitions of a single simple or composite automaton. The selected automaton is denoted by Aut and the set of all simple automata of which it consists by I. The candidate selection starts with the leaves (i.e. simple automata) and progresses towards the root of the hierarchy tree.

Dependency predicate The major problem with original heuristics for checking ample sets [7] is that the overapproximation of the dependency condition (C1) is too restrictive for a system with a lot of synchronization. In fact, only a very small category of systems of CI automata could ever have ample sets that are smaller than the full enabled sets, since it effectively says that an automaton in I that has an enabled action can only ever synchronize with an automaton in I, no matter whether the synchronization is enabled. We relax this condition by allowing synchronizations, that could not be performed by automata in I without them changing their state first.

Definition 1 (dependency condition C1′). *Let $current_i(s)$ be the set of all transitions that could be performed by simple automaton i from state s_i, where $s = (s_1, \ldots, s_n)$. We define C1′ as $\forall i \in I \, \forall \alpha \in current_i(s) \, (automata\ of\ \alpha \subseteq I)$.*

Lemma 1. *Let $ample(s)$ be the set of all enabled transitions that belong to simple automata in I. Then C1′ implies C1.*

Proof. Suppose the opposite: C1′ holds, but C1 does not. Then there is a path from s on which a transition β dependent on $\alpha \in ample(s)$ appears before all transitions from $ample(s)$. That β is dependent on α means that β and α share at least one automaton. As C1′ holds, the automaton or automata of α are in I. Therefore at least one automaton of β, say i, is in I. Clearly, β cannot be enabled in s, as then it would have to be in $ample(s)$.

Thus, if one automaton of β is not in I, the current state of i needs to change before β becomes enabled; otherwise β would violate C1′. If all automata of β are in I, at least one of them has to change its state. However, to change any state of an automaton in I, a transition in *ample*(s) has to be performed first, since all automata in I can currently synchronize only among themselves, which leads to a contradiction. □

Visibility predicate In order to check C2, we need a visibility predicate. As mentioned in the previous section, all transitions with an interesting action and all transitions that change the state atomic propositions have to be considered visible. As usual in the partial order reduction method, we are only interested in the change of those state atomic propositions that appear in the formula that is to be verified. However, as we deal with enabledness propositions, which are properties of the whole state space, we need to use an overapproximation.

Definition 2 (closure). *Let the set of all state labels in the formula be denoted by Ap'. Then an $\mathcal{E}$-closure c of Ap' is defined as:*

$$c(Ap') = Ap' \cup \{\mathcal{E}(m,a,-), \mathcal{E}(-,a,n) \mid \mathcal{E}(m,a,n) \in Ap'\}$$

The *visible* predicate is then defined as follows.

Definition 3 (*visible*). *Let the set of all state labels in the formula be denoted by Ap' and the set of interesting action labels by Act'. Then, if $\alpha = \langle i, s_i, s'_i, l\rangle$ is a simple transition:*

$$\mathit{visible}(\alpha) \iff l \in Act' \vee (\mathcal{L}(s_i) \cap c(Ap') \neq \mathcal{L}(s'_i) \cap c(Ap'))$$

and if $\alpha = \langle i, j, s_i, s'_i, s_j, s'_j, l\rangle$ is a sync transition:

$$\mathit{visible}(\alpha) \iff l \in Act' \vee \exists k \in \{i,j\} : (\mathcal{L}(s_k) \cap c(Ap') \neq \mathcal{L}(s'_k) \cap c(Ap'))$$

Lemma 2 (*visible* is correct). *If a transition $(s,(m,a,n),s')$ in the state space is visible, the predicate visible(α) holds, where α is the corresponding transition of the state transition system.*

Proof. If a transition $(s,(m,a,n),s')$ is visible, then either $(m,a,n) \in Act'$ or $\mathcal{L}(s) \cap Ap' \neq \mathcal{L}(s') \cap Ap'$. In the first case, *visible*(α) clearly holds. For the second case, suppose that there is some $\mathcal{E}(k,b,l) \in \mathcal{L}(s) \cap Ap' \setminus \mathcal{L}(s') \cap Ap'$. That means that there is a transition $s \xrightarrow{(k,b,l)} t$, but there is no (k,b,l) transition enabled in s'. This transition can be either simple or sync. If it is simple, then there is some i such that $s_i \xrightarrow{(k,b,l)} t_i$ and clearly, $s'_i \neq s_i$, otherwise transition (k,b,l) would also be enabled in state s'. Thus $\mathcal{E}(k,b,l) \in \mathcal{L}(s_i) \cap c(Ap')$, but $\mathcal{E}(k,b,l) \notin \mathcal{L}(s'_i) \cap c(Ap')$ and since $s'_i \neq s_i$, the transition α must be local for automaton i and *visible*(α) holds.

If the (k,b,l) transition is sync, then there have to be i,j such that $s_i \xrightarrow{(k,b,-)} t_i$ and $s_j \xrightarrow{(-,b,l)} t_j$. As the (k,b,l) transition is not enabled in s', that means that either α changes the state of i and s'_i has no $(k,b,-)$ transition, or α changes the state of j and s'_j has no $(-,b,l)$ transition. Suppose w.l.o.g. that it is the case with i. Clearly $\mathcal{E}(k,b,-) \in c(Ap')$ by the definition of the closure. But then $\mathcal{E}(k,b,-) \in \mathcal{L}(s_i) \cap c(Ap')$ and $\mathcal{E}(k,b,-) \notin \mathcal{L}(s'_i) \cap c(Ap')$, thus *visible*$(\alpha)$ holds. The other cases and directions are similar. □

Checking of C0, C2 and C3 We also slightly change the original approach to checking the conditions C0, C2 and C3. The first one is trivial to check, since we only have to determine whether I has any enabled transitions. The other two are always invalidated by a counterexample transition, which means that it is sufficient to check them for all enabled transitions once and propagate the invalidation to all composite automata to which it belongs. These conditions are checked for all automata before proceeding with the checking of C1, which is more expensive and therefore only attempted on automata that have passed the first test.

```
begin
    $C_0 \leftarrow \emptyset$;
    $C_{23} \leftarrow$ Set of all automata;
    foreach $\alpha \in$ enabled($s$) do
        $C_0 \leftarrow C_0 \cup$ automata of $\alpha$;
        if visible($\alpha$) $\vee$ inStack($s'$) then $C_{23} \leftarrow C_{23} \setminus$ automata of $\alpha$;
    end
    foreach $i \in$ Set of all composite automata do
        $A \leftarrow$ automata which compose $i$;
        if $A \cap C_0 \neq \emptyset$ then $C_0 \leftarrow C_0 \cup \{i\}$;
        if not $A \subseteq C_{23}$ then $C_{23} \leftarrow C_{23} \setminus \{i\}$;
    end
    Candidate set $\leftarrow C_0 \cap C_{23}$;
end
```

Algorithm 1: Checking of C0, C2 and C3

Checking of C1 We want to determine whether there exists any action in $\bigcup_{i \in I} \mathit{current}_i(s)$ which is a synchronization with an automaton not in I. To do so, we take names of all input (resp. output) actions from each $s_i, i \in I$ and then compare them with names of output (resp. input) actions from all states of all simple automata not in I. Any matching couple is a counterexample for the ample set.

When we find a set that satisfies all four conditions, we accept it as an ample set and use it for the verification instead of the set of all enabled actions.

```
begin
    $C_{\mathsf{in}}, C_{\mathsf{out}}, O_{\mathsf{in}}, O_{\mathsf{out}} \leftarrow \emptyset$;
    foreach $i \in$ simple automata in $I$ do
        $C_{\mathsf{in}} \leftarrow C_{\mathsf{in}} \cup$ names of input actions of $i$ from $s_i$;
        $C_{\mathsf{out}} \leftarrow C_{\mathsf{out}} \cup$ names of output actions of $i$ from $s_i$;
    end
    foreach $i \in$ simple automata not in $I$ do
        $O_{\mathsf{in}} \leftarrow O_{\mathsf{in}} \cup$ names of all input actions of $i$;
        $O_{\mathsf{out}} \leftarrow O_{\mathsf{out}} \cup$ names of all output actions of $i$;
    end
    return $(C_{\mathsf{in}} \cap O_{\mathsf{out}} = \emptyset) \wedge (C_{\mathsf{out}} \cap O_{\mathsf{in}} = \emptyset)$;
end
```

Algorithm 2: Checking of C1

4 Case Study

To provide some evidence for the effectiveness of the partial order method, we have implemented the method within the CoIn verification environment and applied it on a case study. Our previous experience with verification of this model, which has uncovered the need of a partial order reduction method for state/event systems, is reported upon in [8].

The modelled system, the *Trading System*, serves to handle sales in a chain of supermarkets. Its functionality includes the interaction with the cashier at the cash desk, as well as accounting the sale at the inventory. The system is open, designed to interact with external components representing users of the system (cashiers and managers) and a bank application. The model of the system consists of 140 simple CI automata, composed hierarchically into 34 composite automata up to 6 levels of depth. The behaviour of the model features a high degree of independent interleaving of actions, it can be thus expected to achieve a fair amount of state space reduction using the partial order reduction method.

The results obtained by using the method are summarized in Table 1. The various models the verification was performed on were created by complementing the Trading Systems with various components depicting the users of the system.

Table 1. Experimental results (the reduction ratio relates the number of states of the model to the number of states obtained after applying partial order reduction)

Model		without POR			with POR			reduction ratio
		# states	RAM (MB)	time (s)	# states	RAM (MB)	time (s)	
C	2	749 340	139	498	30 618	11	40	24 : 1
C	5	1 498 679	274	1 010	61 771	17	66	24 : 1
C	9	750 684	139	499	30 774	11	40	24 : 1
SC	2	29 341	9	19	2 959	5	4	10 : 1
SC	5	58 681	15	39	6 013	5	6	10 : 1
SC	9	29 629	9	20	2 995	5	4	10 : 1
SCM	2	22 745 391	4 045	21 656	2 016 210	494	2 619	11 : 1
SCM	5	—	—	—	4 084 764	987	4 367	—
SCM	9	22 915 023	4 076	21 864	2 037 002	499	2 640	11 : 1
SCR	2	2 994 016	570	2 119	28 633	11	39	105 : 1
SCR	5	5 988 032	1 128	4 631	58 078	17	67	103 : 1
SCR	9	3 034 336	578	2 150	29 006	10	40	105 : 1
SCSM	2	6 369 598	1 135	4 692	542 794	139	688	12 : 1
SCSM	5	12 739 195	2 263	10 434	1 098 699	273	1 144	12 : 1
SCSM	9	6 413 518	1 143	4 725	548 094	140	694	12 : 1
TSC	2	1 356 277	245	934	37 398	13	48	36 : 1
TSC	5	2 712 553	484	1 936	76 219	22	83	36 : 1
TSC	9	1 373 653	248	948	37 888	13	48	36 : 1

Here, C, SC, SCM, SCR, SCSM and TSC stand for the different user components composed with the system. For each of this variants, three properties were verified, those correspond with properties 2, 5 and 9 as described in [8].

The table shows the number of states of each model combined with each property and the memory and time that was needed to generate the state space, both with and without employing the partial order reduction. A dash (—) in the table means that the information is not available, as the process of state space generation exceeded the maximum of 4 GB of memory. Our experience with applying the partial order reduction on this case study is very positive. In all cases, the state space has been reduced to at least one tenth of the original size and there have been cases where the reduction ratio surpassed one hundred.

5 Conclusion

In our work we present an implementation of the partial-order reduction for state/event LTL in the framework of CI automata. We explain the necessity of modification of the original heuristics for computing ample sets as well as our method of choosing candidate sets, which takes advantage of the hierarchical structure of CI automata. The case study shows how much time and space can be saved using POR in particular cases.

References

1. Brim, L., Černá, I., Vařeková, P., Zimmerova, B.: Component-Interaction automata as a verification-oriented component-based system specification. In: Proceedings of SAVCBS'05, ACM (2005) 31–38
2. Beneš, N., Brim, L., Černá, I., Sochor, J., Vařeková, P., Zimmerova, B.: The CoIn Tool: Modelling and Verification of Interactions in Component-Based Systems. In: Proceedings of FACS'08. (2008) 221–225
3. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DiVinE – a tool for distributed verification. In: Proceedings of CAV'06. Volume 4144 of LNCS., Springer-Verlag (2006) 278–281
4. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/event-based software model checking. In: Proceedings of IFM'04. Volume 2999 of LNCS., Springer-Verlag (2004) 128–147
5. Peled, D.: All from one, one from all: on model checking using representatives. In: Proceedings of CAV'93. Volume 697 of LNCS., Springer-Verlag (1993) 409–423
6. Beneš, N., Brim, L., Černá, I., Sochor, J., Vařeková, P., Zimmerova, B.: Partial order reduction for state/event LTL. In: Proceedings of iFM'09. Volume 5423 of LNCS., Springer (2009) 307–321
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. Cambridge, London, MIT Press (1999)
8. Beneš, N., Černá, I., Sochor, J., Vařeková, P., Zimmerova, B.: A case study in parallel verification of component-based systems. In: Proceedings of PDMC'08. (2008) 35–51

The Parameterized Complexity of Oriented Colouring

Robert Ganian[⋆]

Faculty of Informatics, Masaryk University
Botanická 68a, 602 00 Brno, Czech Republic
ganian@mail.muni.cz

Abstract The oriented colouring problem is intuitive and related to undirected colouring, yet remains NP-hard even on digraph classes with bounded traditional directed width measures. Recently we have also proved that it remains NP-hard in otherwise severely restricted digraph classes. However, unlike most other problems on directed graphs, the oriented colouring problem is not directly transferable to undirected graphs. In the article we look at the parameterized complexity of computing the oriented colouring of digraphs with bounded undirected width parameters, whereas the parameters "forget" about the orientations of arcs and treat them as edges. Specifically, we provide new complexity results for computing oriented colouring on digraphs of bounded undirected rank-width and a new algorithm for this problem on digraphs of bounded undirected tree-width.

1 Introduction

The study of undirected colourings of graphs has become the focus of many authors and lead to a number of interesting results. However, only in the last decade has this been extended to directed graphs. The notion of oriented colouring was first introduced by Courcelle [2]. Oriented colouring has been studied by several authors, see e.g. the work of Nešetřil and Raspaud [11] or the survey by Sopena [13].

Similarly to undirected colouring, computing the oriented chromatic number (*OCN* in brief) and deciding oriented colourability of digraphs are both NP-hard problems. However, while undirected colouring becomes easy if we restrict the input to the graph class of trees, even deciding oriented colourability by 4 colours (also referred to as OCN_4) remains NP-hard on directed acyclic graphs (further referred to as DAGs) [3]. And since the vast majority of digraph parameters have low, fixed values on DAGs, this alone means that they would not be useful for computing *OCN*.

[⋆] supported by the research grants GAČR 201/09/J021 and MUNI/E/0059/2009

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2350

Bi-rank-width (first introduced by Kanté [9]), the digraph equivalent to rank-width, is an exception since it can have high values for DAGs – we have recently shown that deciding OCN_k is in FPT on digraphs of bounded bi-rank-width [7] (FPT stands for fixed parameter tractable, meaning that the time complexity is not only polynomial for any fixed value of the parameter, but also the degree of the polynomial does not depend on the parameter). Unfortunately, the case of computing OCN is worse than for OCN_k: there is no known parameterized algorithm for computing OCN utilizing a digraph parameter. But what about undirected graph parameters?

Most hard problems on directed graphs can be directly translated to undirected graphs. Consider c-Path, Hamiltonian Path, Hamiltonian Cycle, Directed Steiner Tree, Directed Dominating Set, Directed Feedback Vertex Set – all of these directed problems have also been extensively studied on undirected graphs. OCN is different; its definition only makes sense on digraphs. Nevertheless, we show that it is still possible to naturally and intuitively use well-known undirected width parameters for computing OCN on directed graphs. In the article we present new complexity results and a new parameterized algorithm for OCN on digraphs restricted by undirected width parameters.

2 Preliminariess

We assume that the reader is familiar with all basic definitions related to undirected and directed graphs. Keep in mind that digraph stands for directed graph and DAG stands for directed acyclic graph.

Let G, H be digraphs. A *homomorphism* of G to H is a mapping $f : V(G) \to V(H)$ such that for all $(a, b) \in E(G)$, it holds $(f(a), f(b)) \in E(H)$. The k-oriented chromatic number (OCN_k) problem is then defined as follows: Given a digraph G, is there a homomorphism from G to H, where H is some (irreflexive antisymmetric) orientation of edges of the complete graph on k vertices? OCN is the optimization problem of finding the minimum k for a given digraph such that OCN_k is true.

For simplicity, we will sometimes say that a set of vertices of G have the same colour – meaning that they all map into the same vertex of H. Notice that such vertices with the same colour can never have an arc between them, and that if there is an arc from a vertex coloured A to a vertex coloured B, then there can never be an arc from a vertex coloured B to a vertex coloured A. This is a useful and intuitive way of looking at oriented colouring. Next, we will need the notions of tree-width and rank-width – both being very successful width parameters of undirected graphs.

Tree-width: A *tree decomposition* of an undirected graph $G = (V, E)$ is a tree T together with a collection of subsets $T_x \subseteq V$ (called bags) labeled by the vertices x of T such that $\bigcup_{x \in T} T_x = V$ and (1) and (2) below hold:
(1): For every edge uv of G, there is some x such that $\{u, v\} \subseteq T_x$.
(2): (Interpolation Property) If y is a vertex on the unique path in T from x to z, then $T_x \cap T_z \subseteq T_y$.

The width of a tree decomposition is the maximum value of $|T_x| - 1$ taken over all the vertices x of the tree T of the decomposition. We then say that a graph G has *tree-width* k if G has a tree-decomposition of width k.

Branch-width and rank-width: A set function $f : 2^M \to \mathbb{Z}$ is called *symmetric* if $f(X) = f(M \setminus X)$ for all $X \subseteq M$. A tree is *subcubic* if all its nodes have degree at most 3. For a symmetric function $f : 2^M \to \mathbb{Z}$ on a finite set M, the branch-width of f is defined as follows.

A *branch-decomposition* of f is a pair (T, μ) of a subcubic tree T and a bijective function $\mu : M \to \{t : t \text{ is a leaf of } T\}$. For an edge e of T, the connected components of $T \setminus e$ induce a bipartition (X, Y) of the set of leaves of T. The *width* of an edge e of a branch-decomposition (T, μ) is $f(\mu^{-1}(X))$. The *width* of (T, μ) is the maximum width over all edges of T. The *branch-width* of f is the minimum of the width of all branch-decompositions of f. (If $|M| \leq 1$, then we define the branch-width of f as $f(\emptyset)$.)

Natural applications of this definition include not only rank-width (introduced by Oum [10]) but also its directed counterpart bi-rank-width (Kanté, [9]) and the branch-width of graphs (Robertson and Seymour, [12]). In the case of rank-width we consider the vertex set $V(G) = M$ of a graph G as the ground set.

For a graph G, let $\boldsymbol{A}_G[U, W]$ be the bipartite adjacency matrix of a bipartition (U, W) of the vertex set $V(G)$ defined over the two-element field GF(2) as follows: the entry $a_{u,w}$, $u \in U$ and $w \in W$, of $\boldsymbol{A}_G[U, W]$ is 1 if and only if uw is an edge of G. The *cut-rank* function $\rho_G(U) = \rho_G(W)$ then equals the rank of $\boldsymbol{A}_G[U, W]$ over GF(2). A *rank-decomposition* and *rank-width* of a graph G is the branch-decomposition and branch-width of the cut-rank function ρ_G of G on $M = V(G)$, respectively.

Another notion we will need later on is *b*i-rank-width. For a digraph G, let $\boldsymbol{A}_G[U, W]^+$ ($\boldsymbol{A}_G[U, W]^-$) be the bipartite adjacency matrix of a bipartition (U, W) of the vertex set $V(G)$ defined over the two-element field GF(2) as follows: the entry $a_{u,w}$, $u \in U$ and $w \in W$, of $\boldsymbol{A}_G[U,W]^+$ ($\boldsymbol{A}_G[U,W]^-$) is 1 if and only if $(u,w) \in E(G)$ ($(w,u) \in E(G)$). The *bi-cutrank* function of G is defined as the sum of the ranks of these two matrices $brk_G(X) = \mathrm{rank}(\boldsymbol{A}_G[U,W]^+) + \mathrm{rank}(\boldsymbol{A}_G[U,W]^-)$ over the binary field $GF(2)$. A *bi-rank-decomposition* and *bi-rank-width* of a graph G is then the branch-decomposition and branch-width of this bi-cutrank function brk_G.

We have mentioned that for the purposes of this paper, we will apply undirected width measures on directed graphs. So, unless otherwise specified, by tree-width and rank-width we will mean the undirected variants of these measures, even when speaking of digraphs. Formally, given a digraph $G = (V,E)$, we consider an undirected graph $G' = (V(G), E(G'))$ where $E(G') = \{\{a,b\} : (a,b) \in E(G)\}$, and by restricting G to bounded tree-width or rank-width we actually restrict the values of these parameters on G'. Informally this means that we “forget” about the orientations of arcs when computing tree-width and rank-width.

3 OCN on digraphs of bounded rank-width

Although rank-width is not as restrictive as tree-width, in a certain sense bounding rank-width means limiting the complexity of the structure of the graph, and this can be exploited to design powerful parameterized algorithms. For example, computing the “usual” undirected chromatic number can be done in polynomial time (XP to be precise) on graphs of bounded rank-width (see [6]), and deciding colourability can even be done in FPT time ([5]). Unfortunately, despite its successes with undirected colouring, it turns out that rank-width is not useful for computing the more complicated OCN – even on digraphs of bounded rank-width the problem is DET-hard, i.e. as hard as general graph isomorphism. DET is the class of decision problems reducible in logarithmic space to the problem of computing the determinant of an n-by-n matrix of n-bit integers.

Theorem 3.1. *Computing the oriented chromatic number of digraphs is DET-hard even when restricted to digraphs of bounded undirected rank-width.*

Proof. We employ a reduction from a problem involving tournaments. A tournament is, simply put, a complete graph with arbitrary orientation of edges – more precisely, a digraph with precisely one arc between every pair of distinct vertices. The isomorphism of two tournaments has recently been proved to be DET-hard by Wagner [14].

The reduction works as follows: Given two tournaments G_1, G_2 with n vertices each, we construct G as the disjoint union of G_1 and G_2. Note that the rank-width of G is 1, yet we could still solve the problem of isomorphism of G_1 and G_2 by solving OCN on G.

First, assume that the OCN of G is n. Notice that each of G_i contains exactly n vertices and no colour can appear more than once in each G_i. We know that there exists a colouring of G which uniquely identifies each vertex of G_1 with a vertex of G_2 of the same colour. What remains is to argue that such a bijection $f : G_1 \mapsto G_2$ is an isomorphism. Consider any arc $(a, b) \in E(G_1)$. We need to show $(f(a), f(b)) \in E(G_2)$, but by the definition of tournaments either $(f(a), f(b))$ or $(f(b), f(a))$ must be present, and the latter would contradict the oriented colouring of G.

Now assume that G_1 is isomorphic to G_2 by an isomorphism $f : G_1 \mapsto G_2$. We need to show that G can be coloured by n colours. By definition this means proving that there exists a homomorphism from G to some tournament H on n vertices. Choose $H \cong G_2$, $h : G_2 \mapsto H$ being the isomorphism, and consider the following homomorphism: all $v \in G_2$ map to $h(v)$ and all $v \in G_1$ map to $h(f(v))$. Any arc $(a, b) \in E(G)$ must now be also present in $E(H)$, proving that G is orientedly n-colourable. This concludes our proof.

∎

4 *OCN* on digraphs of bounded tree-width

The introduction of tree-width was a breakthrough in the field, and it still remains the most popular graph parameter to this day. Tree-width exploits the fact that almost every problem is easy on the class of trees, and parameterizes the graph by how "tree-like" it is. Powerful tools now exist for designing algorithms on graphs of bounded tree-width, however these are not capable of handling OCN. Nevertheless it still turns out that it is possible to compute OCN on digraphs of bounded tree-width in FPT time. First, we will need a few known results:

Corollary 4.1 ([8], 6.45). *Graphs with bounded degree, or tree-width, or genus have bounded oriented chromatic number.*

More precisely, the authors of [1] have proved that the "acyclic chromatic number" of graphs with tree-width t is at most 2^{t+1}. Hell and

Nešetřil in [8] obtained a bound on the oriented chromatic number of at most $k \cdot 2^{k-1}$, with k as the acyclic chromatic number of the graph. So altogether we get a bound on OCN of $b(t) = 2^{t+1} \cdot 2^{2^{t+1}-1}$ on digraphs of tree-width at most t.

Now, all that remains is to find an FPT algorithm on tree-width which would decide OCN_k. Unfortunately, no such direct algorithm is known, but we have recently developed an FPT algorithm doing just that running on the bi-rank-width of digraphs [7]. What we need to do now is prove that tree-width also bounds bi-rank-width, allowing us to use the aforementioned algorithm. This theorem is of independent interest – the proof that tree-width bounds rank-width does not immediately translate to bi-rank-width, and no result on the relationship of these two parameters has been previously known.

Theorem 4.2. *The bi-rank-width of a digraph G with tree-width t is at most $2 \cdot (t+1)$.*

Proof. We start by normalizing the tree-decomposition of G in a similar way as in [4, Theorem 6.72]:

1. First, we make the decomposition sub-cubic, i.e. bounding the degrees of nodes to 3. This is accomplished by duplicating the nodes of higher degree and inserting them as subdivisions of incident edges. Thus, nodes with high degrees will be duplicated several times.
2. Next, we make all the sets in the tree decomposition uniform of size $t + 1$ by adding new vertices to the node if necessary. This can be accomplished by adding vertices from neighbours.
3. We ensure that neighbouring sets differ by at most one. This can be achieved by adding interpolating nodes where necessary.
4. Now we make sure all sets of leaves have bags of size 1. This is done by adding a path to each former leaf and reducing the size of each consecutive bag on the path by one, omitting a random vertex. Notice that this cannot break the interpolation property. The sets on these paths will be smaller than $t + 1$, and will be exempt from step 2.
5. Finally, for each node of degree 3 in the decomposition, we create an *attachment node* with the same set by subdividing any of its incident edges.

Now we will transform this tree-decomposition into a bi-rank-decomposition, and argue that such a bi-rank-decomposition has bounded bi-rank-width. First, we perform a Depth-first search starting from any leaf of the tree-decomposition. Every time we come across a new, previously unvisited vertex in a bag at some node, we add it as a pendant vertex to the node if the node has degree at most 2. The decomposition must remain subcubic, so if the degree is already 3, we add it to the node's attachment vertex. In this way, all the vertices previously in bags will be added to the decomposition as leafs in the same order as they appeared in bags.

What remains is to argue that such a bi-rank-decomposition truly has bounded bi-rank-width. Consider any edge of the decomposition. The edges incident to leaves of the decomposition can have a bi-rank-width of at most 2, due to the matrices having a single row or column. All other edges were already present in the tree-decomposition, and due to the nature of tree-decompositions (particularly the interpolation property), only at most $t+1$ vertices could occur in bags on both "sides" of the edge. This means that of all the vertices in the rows of $\boldsymbol{A}_G^+$ (those on one "side" of the edge), only at most $t+1$ could have ever met with the vertices in the columns of $\boldsymbol{A}_G^+$ (i.e. those on the other "side" of the edge) in a bag – and since every edge must be present in some bag, we immediately get that all rows or columns other than those of these $t+1$ vertices will only contain zeros. The same of course holds for the other matrix $\boldsymbol{A}_G^-$. Thus $\mathrm{rank}(\boldsymbol{A}_G^+) + \mathrm{rank}(\boldsymbol{A}_G^-) \leq 2 \cdot (t+1)$.

∎

Recall that on digraphs of tree-width at most t, OCN is bounded by $b(t) = 2^{t+1} \cdot 2^{2^{t+1}-1}$. On the other hand, the algorithm for OCN_k on digraphs of bi-rank-width at most r runs in time $O(2^{k^2} \cdot (2^{kr(r+1)/2} \cdot kr^3 \cdot |V(G)|))$ – the runtime is not written explicitly in [7], however it is based on first considering all orientations of arcs of the tournament on k vertices (2^{k^2} possibilities), and for each it is possible to straightforwardly apply our algorithm for deciding unoriented k-colourability on rank-width [6] which runs in time $(2^{kr(r+1)/2} \cdot kr^3 \cdot |V(G)|)$.

So, to compute OCN, it suffices to simply run through all $b(t)$ admissible colours and for each colour compute OCN_k by the bi-rank-width algorithm. The number of tested colours can be trivially improved to $\log b(t)$ in the same way as one can find a number between 1 and k by using only $O(\log k)$ greater/less-or-equal queries. Altogether, we get:

Corollary 4.3. *OCN can be computed on digraphs of tree-width at most t in time $O\big(\log b(t) \cdot 2^{b(t)^2} \cdot 2^{b(t)\cdot(t+1)(2t+3)} \cdot b(t)(2t+2)^3 \cdot |V(G)|\big)$.*

5 Conclusion

In the article we have introduced new positive as well as negative results for computing the oriented chromatic number of digraphs. The positive result on tree-width is of particular interest. This is the first polynomial parameterized algorithm for OCN. Future research should focus on utilizing undirected width measures on other digraph problems, especially those which do not translate directly to undirected graphs. Another direction for future research would be studying OCN on digraphs of bounded bi-rank-width. If it indeed turns out to be hard, would it be possible to find a new powerful directed measure capable of dealing with such hard problems as OCN?

References

1. Michael O. Albertson, Glenn G. Chappell, H. A. Kierstead, Andr Kndgen, Radhika Ramamurthi: Coloring with no 2-colored P4's. *Electron. J. Combin.*, 11: paper R26, 2004.
2. B. Courcelle: The monadic second order-logic of graphs VI : on several representations of graphs by relational structures. *Discrete Appl. Math.*, 54:117-149, 1994.
3. J.-F. Culus and M. Demange: Oriented coloring: Complexity and approximation. In *SOFSEM'06*, volume 3831 of LNCS, pages 226236. Springer, 2006.
4. Downey, R. and Fellows, M.: Parameterized complexity. Springer, 1999.
5. Ganian, R. and Hliněný, P.: On Parse Trees and Myhill–Nerode–type Tools for handling Graphs of Bounded Rank-width. Manuscript, to appear. Extended abstract in *IWOCA08*, LNCS. Springer, 2008.
6. R. Ganian and P. Hliněný: Better Polynomial Algorithms on Graphs of Bounded Rank-width. Manuscript, to appear. Extended abstract in *IWOCA09*, LNCS. Springer, 2009.
7. R. Ganian, P. Hliněný, J. Kneis, A. Langer, J. Obdržálek and P. Rossmanith: On Digraph Width Measures in Parameterized Algorithmics. In *IWPEC2009*, to appear.
8. P. Hell and J. Nešetřil: Graphs and Homomorphisms. Oxford University Press, 2004.
9. Kanté, M.: The rank-width of directed graphs. arXiv:0709.1433v3 (2008).
10. Oum, S.: Rank-width and vertex-minors. *J. Combin. Theory Ser. B* **95**(1) (2005) 79–100.
11. J. Nešetřil and A. Raspaud: Colored Homomorphisms of colored mixed graphs. *Journal of Combinatorial Theory*, Series B, 80(1):147-155, 2000.
12. Robertson, N. and Seymour, P.: Graph minors. X. Obstructions to tree-decomposition. *J. Combin. Theory Ser. B* **52**(2) (1991) 153–190.
13. É. Sopena: Oriented Graph Coloring. *Discrete Math.*, 229:359-369, 2001.
14. Wagner, F.: Hardness Results for Tournament Isomorphism and Automorphism. Mathematical foundations of computer science, 572-583, 2007.

Towards Comparing the Robustness of Synchronous and Asynchronous Circuits by Fault Injection

Marcus Jeitler and Jakob Lechner

Institute of Computer Engineering,
Vienna University of Technology,
Vienna, Austia
`{jeitler, lechner}@ecs.tuwien.ac.at`
`http://ti.tuwien.ac.at`

Abstract. As transient error rates are growing due to smaller feature sizes, designing reliable synchronous circuits becomes increasingly challenging. Asynchronous logic design constitutes a promising alternative with respect to robustness and stability. In particular, delay-insensitive asynchronous circuits provide interesting properties, like an inherent resilience to delay-faults.
This paper presents a new approach for comparing the robustness of synchronous and asynchronous logic. In order to ensure comparability we have developed a tool to automatically transform synchronous designs into their asynchronous counterparts while preserving structural and functional equivalence. Using a saboteur-based fault injection technique, the robustness assessment of both synchronous and asynchronous circuits can then be performed.
At the example of a small-sized test design, this paper demonstrates the capabilities of the proposed approach and, based on these first results, briefly investigates the different behavior of synchronous and asynchronous circuits in the presence of faults.

1 Introduction

The common principle of all asynchronous circuits is a request/acknowledge handshaking protocol. This mechanism regulates the data flow based on the actual speed of the circuit rather than on pessimistic timing assumptions needed in synchronous circuits. Asynchronous circuits can be distinguished by the delay model they employ. In terms of robustness, delay-insensitive circuits are of particular interest since they do not rely on any timing assumptions at all.

Delay-insensitivity in asynchronous circuits is typically implemented using a dual-rail handshake protocol in combination with a certain encoding of data words which allows a completion detection at the recipient. This mechanism

This work is partially funded by the FFG Bridge program: Project “RADIAL” Project Nr: 815458

not only masks delay-faults, the dual-rail encoding with its implicit redundancy also helps in mitigating other fault types in the value domain. While the fault-tolerance properties of asynchronous circuits have been investigated [1], little attention has been paid to the direct comparison of delay-insensitive designs and fault-tolerant synchronous logic so far. Therefore, the aim of the RADIAL project is to compare a synchronous fault-tolerant processor (TMR) with its non-fault tolerant asynchronous counterpart. In addition to a theoretical analysis of the circuit's structure, fault injection experiments will be conducted in the course of the project. The experimental results are expected to provide a detailed insight into the weaknesses and strengths of asynchronous logic wrt. robustness.

The first part of this paper introduces our framework for conducting fault injection experiments in synchronous and asynchronous circuits. The tool-chain consists of two key components: A software tool for automated transformation of synchronous circuits into asynchronous counterparts as well as a powerful fault injection framework that supports the insertion of saboteur units and executes experiments by simulation or hardware emulation. The conversion process for asynchronous circuits will be briefly presented in Section 2. Subsequently, Section 3 gives an overview of the fault injection methodology.

In the second part of the paper a first robustness assessment using our fault injection environment is conducted. In order to keep complexity low, the program counter of a simple processor was used for performing fault injection experiments. Section 4.4 presents the results of these experiments. The paper concludes with Section 5 and provides an outlook on future work.

2 Tranformation of Synchronous Circuits

Since the aim of the RADIAL project is to assess the robustness of synchronous and asynchronous circuits, a common design flow for both circuit types is necessary. Therefore, we have recently developed a software tool for transforming synchronous circuits into *Four State Logic* (FSL) counterparts [2]. FSL is a delay-insensitive, dual-rail encoded implementation style for asynchronous circuits with 2-phase handshaking. The employed data encoding is based on the *Level-Encoded two-phase Dual-Rail* (LEDR) scheme [3]. LEDR represents binary data words in two alternating phases. Phase changes allow to safely separate successive data words and enable the recipient to perform the necessary completion detection. Table 1 shows the codewords for both phases, φ_0 and φ_1.

	φ_0	φ_1
0	00	01
1	11	10

Table 1: LEDR encoding scheme.

An important property of our FSL tool-chain is that the conversion process not only preserves the logical function but also retains the structure of the original circuit at gate-level. This behavior is essential for obtaining comparable results from fault injection experiments. The starting point of the FSL design flow is a conventional synchronous synthesis tool, e.g., *Synopsys*, which translates the clocked circuit description into a gate-level netlist. The latter can subsequently be processed by the FSL conversion tool: Flip-flop components are identified and grouped into registers. A directed graph which represents the data dependencies among the registers is derived from the netlist. Tokens, placed on the graph's edges, can be used to illustrate the data flow. These tokens denote a phase difference between a source and a sink register, which means that a new data word is available for the sink register. Thus, a register may switch if its input edges carry a token and the register's own output tokens have been consumed by all successor stages [4].

The initial token assignment plays a crucial role for the function of an FSL circuit. If the behavior of the original synchronous circuit should be preserved, this assignment is straightforward: In a synchronous system all registers pass their inital value to the successor stages. Thus, all registers of the resulting FSL circuit have to produce initial output tokens. However, in the presence of feedback paths this token configuration causes deadlocks because all registers hold tokens, which need to be consumed first. In order to resolve these deadlocks empty buffer registers have to be inserted into the feedback path (see Figure 1). This deadlock resolution is handled automatically by our FSL tool.

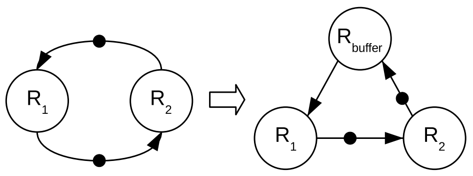


Fig. 1: Initial token assignment, deadlock removal.

Finally, an FSL netlist of the circuit is generated by replacing synchronous flip-flop components with FSL latches and by adding the required acknowledge signals between successive registers. Furthermore, all single-rail signals are converted to dual-rail signals. The asynchronous netlist can then be processed by conventional place & route tools (e.g., Altera QuartusII) for mapping the asynchronous circuit onto FPGA platforms.

3 Fault Injection Framework

In this section we will introduce FuSE, a hardware accelerated HDL-based fault injection environment which supports arbitrary synchronous or asynchronous (with respect to FSL) VHDL circuits.

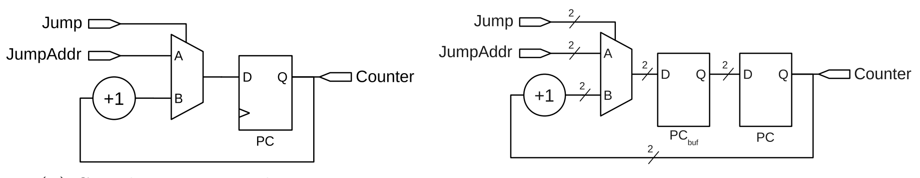
In order to overcome the drawbacks of current simulation- and emulation-based fault injection approaches [5], the FuSE concept integrates both methods in a single tool and allows the user to switch between these modes as required: At an early design stage the user can benefit from executing fault injection experiments within the preferred simulation environment for maximum observation capability. However, when some modules are completed, they can be synthesized and moved to the FPGA, which considerably improves the simulation speed while preserving the visibility of internal signals. This co-simulation support has been achieved by integrating FuSE into the SEmulator® engine – a hardware accelerator for HDL simulations. The resulting environment consists of three core components: The user front-end called Hpe_desk, an HDL simulator (ModelSim in our case) and a rather low cost hardware environment consisting of a prototyping FPGA board equipped with a proprietary PCI-express interface. A more detailed description concerning the SEmulator® and the FuSE integration can be found in [6].

To facilitate fault injection, FuSE uses a source modification approach based on saboteur devices, which currently supports stuck-at 0/1 and bit-flip faults. A saboteur can be activated or deactivated at runtime so that permanent as well as transient faults can be emulated. During the set-up phase the design under test (DUT) is enhanced with the saboteurs and the corresponding control ports. For observation and evaluation of the experiments, additional observation ports can be added. The required modification can either be specified via stylized comments, which are simply added to the source code, or via the Hpe_desk scripting interface. After the configuration the transformation of the source code is automatically performed by the Hpe_desk software.

4 Case Study

In order to demonstrate the unified fault injection platform for both synchronous and asynchronous circuits, this section presents some basic experiments with a rather simple test design. The program counter of our future design under test, the SPEAR processor, was chosen as an example, because it is more transparent with respect to the transformation process and the analysis of the conducted experiments than a complex fault-tolerant synchronous processor.

A schematic representation of the program counter (PC) is presented in Figure 2. The circuit consists of the register storing the current address of the PC, an adder and a multiplexer. The multiplexer selects the incremented counter address provided by the adder or an external jump address input. Unless the jump input is active, the program counter is incremented with every clock cycle in the synchronous implementation (Figure 2a). The asynchronous PC, which has been derived from the synchronous version, produces a new counter output with every phase transition of the circuit. Figure 2b shows the corresponding FSL implementation. In order to retain a correct and equivalent behavior the circuit requires an additional buffer register (PC_{buf}) in its feedback path as explained in Section 2.



(a) Synchronous Implementation (b) Asynchronous Implementation

Fig. 2: Program Counter.

4.1 Experiment Setup

For the fault injection experiments saboteur devices have been inserted in both versions of the program counter. The locations were determined in the synchronous VDHL description and therefore automatically transferred by the FSL transformation process[1]. The respective locations are marked in Figure 3.

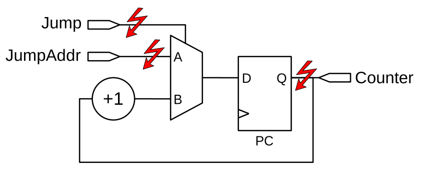


Fig. 3: Injection Location.

Due to the characteristics of the VHDL code, no saboteurs could directly be placed at the output of the adder and the multiplexer. However, faults at these locations can be simulated with the available injection points. As the counter only has one output that could transport an error to the rest of the circuit, the evaluation of the experiments is based on a comparison between this output and a reference output obtained from a fault free execution. The workload, a simple testbench, was configured to increment the counter for 10 cycles, then jump to address 4 and increment from there on.

4.2 Synchronous Fault Injection

In synchronous circuits one or multiple faults can be injected with every active clock edge. Using the inserted saboteur devices each injection point can be configured as a “stuck-at-0”, “stuck-at-1” or “bit-flip” fault during the execution of the experiment. Due to the simple structure of our DUT, the outcome of the experiments was not unexpected: Every injected fault has the potential to

[1] Note, that the FSL implementation has a saboteur on each rail of a disturbed signal, doubling the number of saboteur devices.

manifest itself. While "stuck-at" faults can be masked by a matching current state of the respective signal, "bit-flips" always lead to an error. Furthermore, the duration of a fault usually has a different impact on the result. A special case is the bit-flip in the feedback loop which can neutralize itself if the faulty value is affected again in the subsequent clock cycle.

4.3 Asynchronous Fault Injection

As explained in Section 2 FSL uses a dual-rail coding scheme which encodes every binary data value in two phases: φ_0 and φ_1. Due to this encoding, the introduced fault types have different effects in an FSL circuit than in a synchronous circuit. Table 2 lists all possible single-rail and dual-rail faults marked with their observed characteristic.

FSL State		↑		↓		↕		↑↑	↓↓	↕↕	↑↓	↓↑
φ_0	00	01^P	10^P	00^M	00^M	01^P	10^P	11^V	00^M	11^V	10^P	01^P
	11	11^M	11^M	10^P	01^P	10^P	01^P	11^M	00^V	00^V	10^P	01^P
φ_1	01	01^M	11^P	00^P	01^M	00^P	11^P	11^P	00^P	10^V	10^V	01^M
	10	11^P	10^M	10^M	00^P	11^P	00^P	11^P	00^P	01^V	10^M	01^V
		single-rail faults						dual-rail faults				

Fault Type: ↑. . . stuck-at-1, ↓. . . stuck-at-0, ↕. . . bit-flip
Characteristic: M. . . masked, P. . . phase change, V. . . value change

Table 2: FSL Fault Characteristics.

The FSL coding scheme describes a valid state transition as the change of a single-rail. As a result, all single-rail faults are either masked or change the phase of the signal. In theory, both outcomes should not lead to an error.

In contrast to single-rail faults, dual-rail faults can also change the value of a signal, which can become an error in our circuit if it is stored in the register. As shown in Table 2, not only fault constellations that are rather unlikely, e.g. both rails of a signal are affected in different ways, but also ↑↑ and ↓↓ faults can lead to a value change. However, this behavior only occurs in specific cases with values encoded in φ_0 (00, 11), otherwise the fault has no effect.

During our exhaustive experiments every proposed fault constellation was injected in order to confirm the theoretical assumptions. The corresponding results will be presented in the following section.

4.4 Experimental Results

Single-Rail Faults Although the experiments with single-rail faults did not cause an error, the observed behavior complements the theoretical characteristics presented in the previous section. A fault which is masked in one phase can keep a rail from switching, thereby delaying the execution of the subsequent phase. As a result a masked fault is transformed into a delay fault. This behavior also offers an easy way to generate and investigate delay faults in asynchronous circuits.

If a single-rail fault causes a phase shift the execution of the circuit is not necessarily stopped at once. Depending on a specific fault activation window the inconsistent phase either affects the current phase or the subsequent phase. In the first case, the circuit is halted at once. In the latter case the phase shift can either halt the circuit in the subsequent phase or become masked. However, neither case did lead to an error.

Dual-Rail Faults The considered dual-rail faults either affect both rails the same way or in different ways. As our experiments showed, if a dual-rail fault is masked then it will definitely block the execution of the next phase until the fault is removed again.

If a dual-rail fault results in a phase change, the execution is either blocked or continues with the next phase where the original fault becomes masked or introduces a value change and therefore an error. Nevertheless, if the fault is still active, the further execution will be blocked until the fault is removed again.

The third fault characteristic is an instant value change. If it hits the fault activation window it will cause an error first before the execution is halted. The only exception to this rule is the dual-rail bit-flip fault: Execution will continue and the introduced error will be removed every other cycle if the faulty value is affected again.

4.5 Beyond Experimental Results

Due to the limited complexity of our test circuit certain temporal effects, as e.g. the skew between the signal rails, were not distinctive enough to influence the execution of our experiments. This section will therefore address possible results which depend on a more intricate timing.

Skew-affected Single-Rail Faults As explained in Section 4.4 the investigated program counter is resilient to single-rail faults, which is a direct result of the implemented coding scheme. However, the assumption that single-rail faults can be tolerated mainly depends on the the skew between the inputs of an FSL register. If the skew is small, i.e., all signal rails almost switch simultaneously, then a single-rail fault will either be masked or causes an inconsistent input vector (refer to Table 2). If the skew is sufficiently large, then even a single-rail fault may introduce a value error due to a premature phase change. This behavior is illustrated in Figure 4.

The presented example shows an FSL register with two inputs *A*,*B* and an output *C*. Due to some delay at input *A*, *B* is always assigned a new value (represented by a phase change) prior to *A* in the dataflow diagram. As the inputs are inconsistent during this period, the FSL register has to retain its last

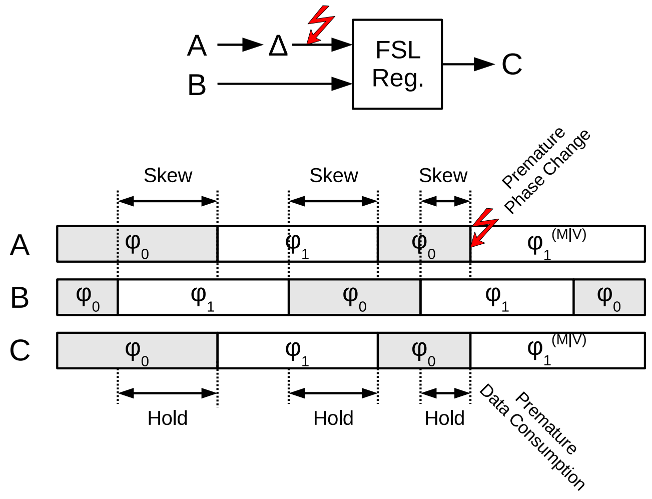


Fig. 4: Skew-affected Single-Rail Fault.

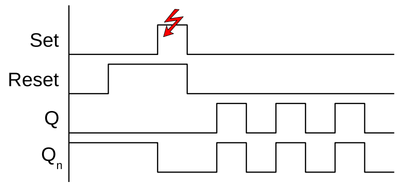
valid output. This "hold" window makes the circuit susceptible for faults, as the register only waits for the inputs to become consistent again. If a single-rail fault causes a premature phase change at A within the "hold" window, possibly wrong data is consumed and a value error might propagate.

By extending this consideration to an arbitrary number of inputs, we can derive the following properties for a single-rail error:

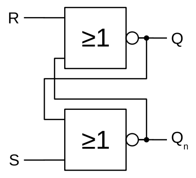
– The fault activation window is determined by the skew of the two slowest inputs.
– A single-rail fault has to hit the slowest rail pair in order to cause a premature data consumption.

Considering the current theoretical analysis we conclude, that the skew, respectively the fault activation window within the test circuit is too small so that all injected single-rail faults become masked according to the coding scheme. Future experiments should therefore be conducted on more complex circuits in order to qualitatively assess the inherent robustness of asynchronous logic.

Metastability Effects While single-rail faults only produce premature phase changes, dual-rail faults can additionally cause invalid transitions in the FSL coding scheme. These may lead to timing issues within the storage elements which are extensively used by FSL designs. Basically all FSL components, sequential registers as well as basic combinational gates, contain RS-latches for holding a data value or preserving a stable state during ongoing input transitions. Although this latch type is very convenient for building FSL gates, first fault injection experiments have uncovered an unpleasant vulnerability: If a fault causes the Set and the Reset line to be active at the same time and subsequently both signals are released almost simultaneously, the output of the latch starts to oscillate. The RS-latch basically behaves like a JK-latch in toggle mode. Figure 5a shows a waveform of this behavior.



(a) Upset on Set signal



(b) RS-latch

Fig. 5: Fault disrupting the function of an RS latch.

RS-latches can be built from two cross-coupled NOR-gates (see Figure 5b). If the *Set* and the *Reset* input are active at the same time, Q and Q_n are both forced low, which violates the equation $Q = not\ Q_n$. When the *Set* and the *Reset* inputs are released again, the NOR-gates form a loop which may start to oscillate.

5 Conclusion and Outlook

The fault injection experiments presented in this paper outline the inherent fault tolerance capabilities of an FSL circuit with respect to arbitrary rail faults. While simple, well-balanced designs are resilient to single-rail faults, more complex architectures might yet be error-prone. In this context, the duration of the "hold" window has been identified as a source for the propagation of single-rail faults. A dual-rail fault, however, does not depend on a specific activation window. Although it can cause a value error at any instant, it only affects the circuit once during its occurrence because the execution will be blocked afterwards. Therefore, the duration of such a fault has no additional effect on the amount of possible errors which is a major difference compared to a synchronous circuit where a fault can become activated periodically.

In future experiments we plan to investigate the propagation of single-rail faults as well as the necessary requirements for the fault activation. In this context, the temporal resolution for the injection will be improved in order to variate the occurrence of a fault within this critical period. The gathered results should then let us develop appropriate mitigation strategies. The improved architecture will be used for the comparison of the synchronous fault-tolerant processor and its non-fault tolerant asynchronous counterpart which has been recently developed [7].

References

1. LaFrieda, C., Manohar, R.: Fault detection and isolation techniques for quasi delay-insensitive circuits. In: DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks, Washington, DC, USA, IEEE Computer Society (2004) 41
2. Lechner, J.: Implementation of a Design Tool for Generation of FSL Circuits. Master's thesis, Vienna University of Technology, Austria (2008)
3. McAuley, A.J.: Four State Asynchronous Architectures. IEEE Transactions on Computers **41**(2) (1992) 129–142
4. Sparsø, J., Furber, S., eds.: Principles of Asynchronous Circuit Design: A Systems Perspective. Kluwer Academic Publishers (2001)
5. Benso, A., Prinetto, P.: Fault Injection Techniques and Tools for Embedded Systems. Kluwer Academic Publishers, Norwell, MA, USA (2003)
6. Jeitler, M., Delvai, M., Reichor, S.: FuSE - A Hardware Accelerated HDL Fault Injection Tool. In: 5th Southern Conference on Programmable Logic, 2009. SPL. (2009) 89–94
7. Jeitler, M., Lechner, J.: Speeding up Fault Injection for Asynchronous Logic by FPGA-based Emulation. to be published at: International Conference on ReConFigurable Computing and FPGAs, 2009. ReConFig. (2009)

Undecidability of Coverability and Boundedness for Timed-Arc Petri Nets with Invariants

Lasse Jacobsen, Morten Jacobsen and Mikael H. Møller

Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300, 9220 Aalborg Øst, Denmark
`{lassejac,mortenja,mikaelhm}@cs.aau.dk`

Abstract. Timed-Arc Petri Nets (TAPN) is a well studied extension of the classical Petri net model where tokens are decorated with real numbers that represent their age. Unlike reachability, which is known to be undecidable for TAPN, boundedness and coverability remain decidable. The model is supported by a recent tool called TAPAAL which, among others, further extends TAPN with invariants on places in order to model urgency. The decidability of boundedness and coverability for this extended model has not yet been considered. We present a reduction from two-counter Minsky machines to TAPN with invariants to show that both the boundedness and coverability problems are undecidable.

1 Introduction

Time-dependent models have been extensively studied due to increasing demands on the reliability and safety of embedded software systems. *Timed automata* [11] and various time-extensions of *Petri nets* (e.g. [4]) are among the most studied time-dependent models. A recent paper by Srba [14] provides a comparative overview of these models.

Timed-Arc Petri Nets (TAPN's) [4] is a popular time-extension of Petri Nets [10] in which each token is assigned an age (a real number), and time intervals on arcs restrict the ages of tokens that can be used to fire a transition. The reachability problem has been shown undecidable for TAPN [12]. In particular, a TAPN cannot correctly simulate a test for zero on a counter [3]. However, other problems, like boundedness and coverability remain decidable [2][1].

Recent work on the verification tool TAPAAL by Byg et al. [5] have, among other things, introduced invariants on places into the TAPN model as a way to represent urgency. However, urgency alone does not allow a TAPN to correctly simulate a test for zero on a counter. Nevertheless, we show that invariants on places makes the coverability and boundedness problem undecidable. We adopt the main idea from [12] (see also [6] for a similar proof technique for another time extension of Petri nets), in which a two-counter Minsky machine (2-CM) is weakly simulated by a TAPN. In contrast to their reduction, the extension of invariants allows us to detect when the net incorrectly simulates the 2-CM. Further, our reduction allows us to prove the undecidability of both the coverability and boundedness problems.

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2346

2 Basic Definitions

Many of the definitions in this section are following [13]. The set of all *time intervals* $\mathcal{I}$ and the set of time intervals for invariants $\mathcal{I}_{Inv}$ are defined according to the following abstract syntaxes where $a \in \mathbb{N}_0$, $b \in \mathbb{N}$ and $a < b$:

$$I ::= [a,a] \ | \ [a,b] \ | \ [a,b) \ | \ (a,b] \ | \ (a,b) \ | \ [a,\infty) \ | \ (a,\infty)$$
$$I_{Inv} ::= [0,0] \ | \ [0,b] \ | \ [0,b) \ | \ [0,\infty)$$

We define the predicate $r \in I$ for $r \in \mathbb{R}_0^+$ in the expected way.

Definition 1 (Timed-Arc Petri Net with Invariants). *A* Timed-Arc Petri Net with Invariants *(ITAPN) is a 5-tuple $N = (P, T, F, c, \iota)$ where P is a finite set of places, T is a finite set of transitions such that $P \cap T = \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, $c : F|_{P \times T} \to \mathcal{I}$ is a function assigning time intervals to arcs from places to transitions, and $\iota : P \to \mathcal{I}_{Inv}$ is a function assigning invariants to places.*

We let $\mathcal{B}(\mathbb{R}_0^+)$ denote the set of finite multisets over $\mathbb{R}_0^+$. For a $B \in \mathcal{B}(\mathbb{R}_0^+)$ and some $d \in \mathbb{R}_0^+$, we let $B + d = \{b + d \mid b \in B\}$. Notationally, we use multisets as ordinary sets with the operations $\cup, \setminus, \subseteq, \in$ implicitly interpreted over multisets.

Let us now define a marking on a ITAPN.

Definition 2 (Marking). *A* marking *M on a ITAPN $N = (P, T, F, c, \iota)$ is a function $M : P \to \mathcal{B}(\mathbb{R}_0^+)$, such that for every place $p \in P$ it holds that for every token $x \in M(p)$, $x \in \iota(p)$. The set of all markings over N is denoted $\mathcal{M}(N)$.*

A *marked* ITAPN is a pair (N, M_0) where N is a ITAPN and M_0 is the initial marking. We only allow initial markings in which all tokens have age 0.

The *preset* of a transition t is $^\bullet t = \{p \in P \mid (p,t) \in F\}$ and the *postset* of t is $t^\bullet = \{p \in P \mid (t,p) \in F\}$.

Definition 3 (Firing rule). *Let $N = (P, T, F, c, \iota)$ be a ITAPN, M some marking on it and $t \in T$ be a transition of N.*

We say that t is enabled *if and only if $\forall p \in {^\bullet t}.\ \exists x \in M(p).\ x \in c(p,t)$, i.e. there is a token with an appropriate age at every place in the preset of t.*

If t is enabled in M, it can be fired*, whereby we reach a marking M' defined by $\forall p \in P.\ M'(p) = \big(M(p) \setminus C_t^-(p)\big) \cup C_t^+(p)$ (note that all operations are on multisets and there may be multiple choices for the sets $C_t^-(p)$ and $C_t^+(p)$ for each p. We simply fix the sets before firing t), where*

- $C_t^-(p) = \begin{cases} \{x\} & \textit{if } p \in {^\bullet t} \wedge x \in M(p) \wedge x \in c(p,t) \\ \emptyset & \textit{otherwise} \end{cases}$

- $C_t^+(p) = \begin{cases} \{0\} & \textit{if } p \in t^\bullet \\ \emptyset & \textit{otherwise} \end{cases}$

i.e. from each place $p \in {^\bullet t}$ we remove a token with an appropriate age, and we add a new token with age 0 to every $p \in t^\bullet$.

Definition 4 (Time delays). *Let $N = (P, T, F, c, \iota)$ be a ITAPN and M some marking on it. A time delay $d \in \mathbb{R}_0^+$ is allowed if and only if $(x + d) \in \iota(p)$ for all $p \in P$ and $x \in M(p)$, i.e. by delaying d time units no token violates the invariants. By delaying d time units we reach a marking M', defined as $M'(p) = M(p) + d$ for all $p \in P$.*

A marked ITAPN (N, M_0) is said to be *k-bounded* if the number of tokens in each place does not exceed k for any marking reachable from M_0. A marked ITAPN is *bounded* if it is k-bounded for some $k \in \mathbb{N}$.

Problem 1 (Boundedness). Given a marked ITAPN is it bounded?

A marking M on a ITAPN (N, M_0) is said to be *coverable* if there exists a marking M', reachable from M_0, s.t. $M'(p) \supseteq M(p)$ for each place p in the net.

Problem 2 (Coverability). Given a marked ITAPN (N, M_0) and some marking M, is M coverable?

3 Undecidability of Boundedness and Coverability

In this section we will prove the undecidability of boundedness and coverability by reduction from two-counter Minsky machines.

Definition 5. *A* Two-Counter Minsky Machine (2-CM) *with two non-negative registers r_1 and r_2 is a sequence of instructions $(I_1 : Ins_1;\ I_2 : Ins_2;\ \ldots\ I_{e-1} : Ins_{e-1};\ I_e : HALT)$ where for every j, $1 \leq j < e$, Ins_j is one of the two types:*

- $r_i := r_i + 1;$ *goto* I_k; *where $i \in \{1, 2\}$ and $k \in \{1, 2, \ldots, e\}$ (Increment).*
- *if* $r_i > 0$ *then* $r_i := r_i - 1;$ *goto* I_k; *else goto* I_ℓ; *where $i \in \{1, 2\}$ and $k, \ell \in \{1, 2, \ldots, e\}$ (Test and decrement).*

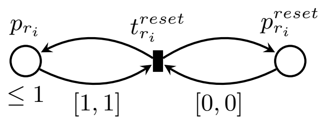
The last instruction is always the HALT instruction. A configuration *of a 2-CM is a triple (j, v_1, v_2) where $j \in \{1, 2, \ldots, e\}$ is the index of instruction I_j to be executed and v_1 and v_2 are the values of the registers r_1 and r_2, respectively.*

The computational step relation of a 2-CM is defined as expected and we use the notation $(j, v_1, v_2) \to (j', v_1', v_2')$ to denote that we perform the current instruction I_j with values v_1 and v_2 in the registers, resulting in the configuration (j', v_1', v_2').

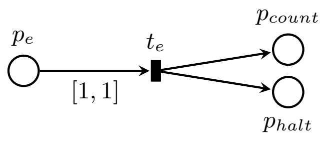
Definition 6 (The Halting Problem for 2-CM). *Given a 2-CM, is it possible to reach the halt instruction from the initial configuration $(1, 0, 0)$, i.e. $(1, 0, 0) \to^* (e, v_1, v_2)$ for some $v_1, v_2 \in \mathbb{N}_0$?*

Theorem 1 (Minsky [9]). *The halting problem for 2-CM is undecidable.*

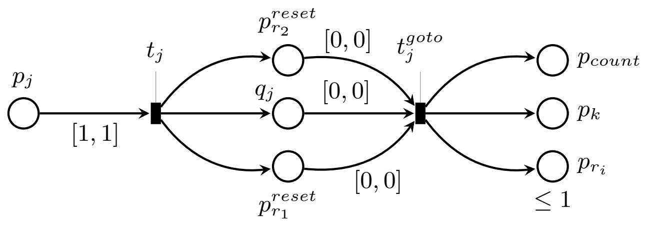
We will now describe the reduction from 2-CM to ITAPN. Given a 2-CM $(I_1 : Ins_1;\ I_2 : Ins_2;\ \ldots\ I_{e-1} : Ins_{e-1};\ I_e : HALT)$ we construct a ITAPN (P, T, F, c, ι) where



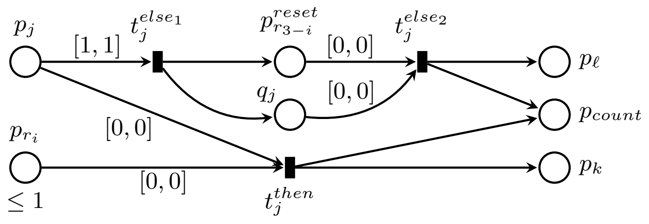
(a) Simulation of a register.



(b) Simulation of Halt instruction.



(c) Simulation of $I_j : r_i := r_i + 1;$ goto I_k.



(d) Simulation of $I_j :$ if $r_i > 0$ then $r_i := r_i - 1;$ goto $I_k;$ else goto I_ℓ.

Fig. 1: ITAPN models for 2-CM simulation.

- $P = \{p_j, q_j \mid 1 \leq j < e\} \cup \{p_{r_1}, p_{r_1}^{reset}, p_{r_2}, p_{r_2}^{reset}\} \cup \{p_{count}\} \cup \{p_e, p_{halt}\}$
- $T = \{t_{r_1}^{reset}, t_{r_2}^{reset}\} \cup \{t_j, t_j^{goto} \mid Ins_j \text{ is of type increment}\} \cup$
  $\left\{t_j^{else_1}, t_j^{else_2}, t_j^{then} \mid Ins_j \text{ is of type test and decrement}\right\} \cup \{t_e\}$

The number of tokens in p_{r_1} and p_{r_2} correspond to the values of r_1 and r_2, the number of tokens in p_{count} remembers the number of computation steps which have been simulated in the net and $p_1, \ldots, p_e$ corresponds to the instructions $Ins_1, \ldots, Ins_e$ such that the place p_j contains one token if and only if the current instruction is Ins_j. For the flow relation we will split it into 4 parts.

- F_1 contains the arcs for the registers. For each register $r_i,\ i \in \{1,2\}$, we add the following arcs to F_1

  $$(p_{r_i}, t_{r_i}^{reset}),\ (t_{r_i}^{reset}, p_{r_i}),\ (p_{r_i}^{reset}, t_{r_i}^{reset}),\ (t_{r_i}^{reset}, p_{r_i}^{reset}) \text{ where}$$
  $$c((p_{r_i}, t_{r_i}^{reset})) = [1,1],\ \ c((p_{r_i}^{reset}, t_{r_i}^{reset})) = [0,0] \text{ and } \iota(p_{r_i}) = [0,1]\ .$$

  This is illustrated in Figure 1a. The number of tokens on p_{r_i} indicates the value of the register. Notice the invariant on the register which disallows tokens with an age greater than 1. Placing a token on $p_{r_i}^{reset}$ allows us to reset the age of all tokens of age 1 in the register.

– F_2 contains the arcs for the increment instructions. For each increment instruction $I_j : r_i := r_i + 1;$ $\mathsf{goto}\ I_k;$, we add the following arcs to F_2

$$(p_j, t_j),\ (t_j, p^{reset}_{r_2}),\ (t_j, q_j),\ (t_j, p^{reset}_{r_1}),\ (p^{reset}_{r_2}, t^{goto}_j),\ (q_j, t^{goto}_j),$$
$$(p^{reset}_{r_1}, t^{goto}_j),\ (t^{goto}_j, p_{count}),\ (t^{goto}_j, p_k),\ (t^{goto}_j, p_{r_i}) \text{ where } c((p_j, t_j)) = [1, 1],$$
$$c((p^{reset}_{r_2}, t^{goto}_j)) = [0, 0],\ \ c((q_j, t^{goto}_j)) = [0, 0] \text{ and } c((p^{reset}_{r_1}, t^{goto}_j)) = [0, 0]\ .$$

This is illustrated in Figure 1c. Notice that we require a delay of one time unit before firing t_j. Because of this, we allow tokens in each register to be reset (by placing tokens on $p^{reset}_{r_1}$ and $p^{reset}_{r_2}$). Following this, by firing t^{goto}_j a token is added to p_{count}, register r_i is incremented by adding a token to p_{r_i} and control is given to the next instruction I_k by placing a token on p_k.

– F_3 contains the arcs for the test and decrement instructions. For each test and decrement instruction I_j : $\mathsf{if}\ r_i > 0\ \mathsf{then}\ r_i := r_i - 1;\ \mathsf{goto}\ I_k;\ \mathsf{else\ goto}\ I_\ell;$, we add the following arcs to F_3

$$(p_j, t^{else_1}_j),\ (p_j, t^{then}_j),\ (p_{r_i}, t^{then}_j),\ (t^{else_1}_j, q_j),\ (t^{else_1}_j, p^{reset}_{r_{3-i}}),\ (q_j, t^{else_2}_j),$$
$$(p^{reset}_{r_{3-i}},\ t^{else_2}_j),\ (t^{else_2}_j, p_\ell),\ (t^{else_2}_j, p_{count}),\ (t^{then}_j, p_{count}),\ (t^{then}_j, p_k) \text{ where}$$
$$c((p_j, t^{else_1}_j)) = [1, 1],\ \ c((p_j, t^{then}_j)) = [0, 0],\ \ c((p_{r_i}, t^{then}_j)) = [0, 0],$$
$$c((p^{reset}_{r_{3-i}}, t^{else_2}_j)) = [0, 0] \text{ and } c((q_j, t^{else_2}_j)) = [0, 0]\ .$$

This is illustrated in Figure 1d. Notice that when we follow the else branch (firing transition $t^{else_1}_j$), we can only reset the ages of tokens in the register on which we are not testing for emptyness.

– F_4 contains the arcs for the HALT instruction. Formally it is defined as

$$F_4 = \{(p_e, t_e), (t_e, p_{count}), (t_e, p_{halt})\} \text{ where } c((p_e, t_e)) = [1, 1]\ .$$

This is illustrated in Figure 1b. Again we require a time delay of one time unit before t_e can be fired and a token placed at p_{halt}.

– The flow relation F can then be defined as the union of the four parts, i.e. $F = F_1 \cup F_2 \cup F_3 \cup F_4$ and we let $\iota(p) = [0, \infty)$ for all $p \in P \setminus \{p_{r_1}, p_{r_2}\}$.

We define the initial marking M_0 such that $M_0(p_1) = \{0\}$ and $M_0(p) = \emptyset$ for all $p \in P \setminus \{p_1\}$.

Let (N, M_0) be the marked ITAPN simulating a given 2-CM. Notice that every place in the net except for $p_{r_1}, p_{r_2}, p_{count}$ is *1-safe* (i.e. contains at most one token). In a *correct* simulation of the 2-CM by our net, a configuration (j, v_1, v_2) of the 2-CM corresponds to any marking M where

$$M(p_j) = \{0\},\ \ M(p_{r_i}) = \underbrace{\{0, 0, \ldots, 0\}}_{v_i \text{ times}} \text{ for } i \in \{1, 2\}, \tag{1}$$

$|M(p_{count})| = n$ where $n \in \mathbb{N}_0$ and $M(p) = \emptyset$ for all $p \in P \setminus \{p_j, p_{r_1}, p_{r_2}, p_{count}\}$.

We will now describe how to simulate the three types of instructions of a 2-CM in a *correct* way. Assume there is a token of age 0 in p_j.

If I_j is an increment instruction, we need to delay for one time unit in order to enable t_j (see Figure 1c). Because we delayed one time unit, all tokens in the registers are now of age 1. In a correct simulation, we fire repeatedly transitions $t_{r_1}^{reset}$ and $t_{r_2}^{reset}$ until all tokens in p_{r_1} and p_{r_2} are of age 0. Note that it is possible to *cheat* in the simulation, as it is possible to leave some tokens of age 1 in p_{r_1} or p_{r_2} when firing t_j^{goto}.

If I_j is a test and decrement instruction there are two possibilities (see Figure 1d). If there is a token of age 0 at p_{r_i}, we fire t_j^{then} in order to decrement the number of tokens in register r_i, and hand over the control to I_k by placing a token on p_k. Otherwise, in the correct simulation we delay one time unit before firing $t_j^{else_1}$. Then we reset the age of all the tokens in the other register, $p_{r_{3-i}}$ to 0. We then proceed by firing $t_j^{else_2}$. This will hand over control to instruction I_ℓ by placing a token on p_ℓ. Again note that it is possible to *cheat* in the simulation, either by leaving tokens of age 1 at $p_{r_{3-i}}$ when proceeding to the next instruction or by taking the *else*-branch even though there is a token at p_{r_i} (because the net does not force us to fire transition t_j^{then} when it is enabled).

If I_j is the halt instruction, we delay one time unit before we fire the last transition t_e and add a token to p_{halt}.

After every instruction one token is added to p_{count}. We will now prove a lemma detailing what happens if we cheat.

Lemma 1. *Let (j, v_1, v_2) be the current configuration of a 2-CM CM, (N, M_0) the associated ITAPN and M a marking corresponding to (j, v_1, v_2) (see Equation 1). If the net cheats then during the simulation of CM in the next computation step it is not possible to simulate an increment instruction, go to the halt state, nor to take the else-branch of a test and decrement instruction. Further, the net can do at most $v_1 + v_2$ decrements before getting stuck.*

Proof. We can perform an incorrect simulation in two ways:

- If all tokens in p_{r_1} and p_{r_2} are not reset to age 0 in an increment or test and decrement instruction before going to the next instruction.
- In a test and decrement instruction, the net can fire the transition $t_j^{else_1}$ even if there is a token of age 0 in p_{r_i}. This is possible by delaying 1 time unit to enable the transition. However, this will result in the tokens in p_{r_i} having age 1 and these can not be reset before going to the next instruction.

In both cases we end up in a marking M' where there is at least one token of non-zero age in either p_{r_1} or p_{r_2}. Observe that the simulation of increment, halt and the else-branch of a test and decrement instruction all require a delay of 1 time unit (see Figure 1) which would violate the invariants $\iota(p_{r_1})$ or $\iota(p_{r_2})$. Thus, the only possibility is to take the then-branch of a test and decrement instruction. However, this is only possible as long as there are tokens of age 0 in p_{r_1} or p_{r_2}. There are v_1 and v_2 tokens in p_{r_1} and p_{r_2}, repectively. Thus, the net can do at most $v_1 + v_2$ decrements before getting stuck. $\square$

3.1 Undecidability Results

First we prove the undecidability of the boundedness problem.

Lemma 2. *Given a 2-CM CM and the associated ITAPN* (N, M_0)*, CM halts if and only if* N *is bounded.*

Proof. We start by proving that if N is bounded then CM halts. Assume that N is k-bounded. Further, assume by contradiction that CM does not halt. After simulating $k+1$ computational steps of CM correctly, the net will be in a marking M where $|M(p_{count})| = k+1$. This is a contradiction to the assumption that N is k-bounded.

Now we prove the implication in the other direction. Assume that CM halts in n steps. We will show that N is $2n$-bounded. If we simulate CM correctly, there will be at most n tokens at the registers, and exactly n tokens at p_{count}. Hence, the net must cheat in order to become unbounded. In the worst case, it cheats at the last step, when there are at most $n-1$ tokens in the registers and $n-1$ tokens at p_{count}. Then we have that the net is $2n$-bounded since there will be at most $2(n-1)$ tokens at p_{count} by Lemma 1. □

From Lemma 2 we conclude the following theorem.

Theorem 2. *The boundedness problem is undecidable for ITAPN.*

We now prove the undecidability of the coverability problem.

Lemma 3. *Let* M *be a marking such that* $M(p_{halt}) = \{0\}$ *and* $M(p) = \emptyset$ *for all* $p \in P \setminus \{p_{halt}\}$*. Given a 2-CM CM and the associated marked ITAPN* (N, M_0)*, as defined above, CM halts if and only if* M *is coverable from* M_0*.*

Proof. First we prove that if CM halts then M is coverable from M_0. Assume that the CM halts. By simulating CM correctly in N, we can easily see that we reach a marking M', with a token in p_{halt}, hence $M'(p) \supseteq M(p)$ for all $p \in P$.

Now we prove that if M is coverable from M_0 then CM halts. Assume that M is coverable from M_0. By assumption there exists a reachable marking M' such that $M'(p) \supseteq M(p)$ for all $p \in P$. By definition of coverability, it holds that $0 \in M'(p_{halt})$ and by Lemma 1 this is only possible if we simulate CM correctly in the net, hence CM halts. □

From Lemma 3 we conclude the following theorem.

Theorem 3. *The coverability problem is undecidable for ITAPN.*

4 Conclusion

We proved that coverability and boundedness is undecidable for Time-Arcs Petri Nets with Invariants by reduction from two-counter Minsky machines. The following table shows a summary of known results about Petri Nets (PN). Our results are emphasized.

	Reachability	Boundedness	Coverability
PN	decidable [8]	decidable [7]	decidable [7]
TAPN	undecidable [12]	decidable [2]	decidable [1]
ITAPN	undecidable [12]	*undecidable*	*undecidable*

Acknowledgements

We would like to thank Jiří Srba, Mads Chr. Olesen, Kenneth Y. Jørgensen and the anonymous reviewers for their comments and suggestions.

References

[1] P. A. Abdulla and A. Nylén. Timed Petri Nets and BQOs. In *Proc. of ICATPN*, volume 2075 of *LNCS*, pages 53–70. Springer, 2001.

[2] P. A. Abdulla, P. Mahata, and R. Mayr. Dense-Timed Petri Nets: Checking Zenoness, Token liveness and Boundedness. *Logical Methods in Computer Science*, 3(1):1–61, 2007.

[3] T. Bolognesi and P. Cremonese. The Weakness of some Timed Models for Concurrent Systems. Technical Report CNUCE C89-29, CNUCE-C.N.R., Oct. 1989.

[4] T. Bolognesi, F. Lucidi, and S. Trigila. From Timed Petri Nets to Timed LOTOS. In *Proc. of IFIP WG 6.1 PSVT X*, pages 395–408, Ottawa, Canada, 1990. North-Holland Publishing.

[5] J. Byg, K. Y. Jørgensen, and J. Srba. An Efficient Translation of Timed-Arc Petri Nets to Networks of Timed Automata. In *Proc. of ICFEM '09*, volume 5799 of *LNCS*. Springer, Dec 2009. To appear (available at author's website).

[6] Neil D. Jones, Lawrence H. Landweber, and Y. Edmund Lien. Complexity of some problems in petri nets. *Theor. Comput. Sci.*, 4(3):277–299, 1977.

[7] R. M. Karp and R. E. Miller. Parallel Program Schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.

[8] E. W. Mayr. An Algorithm for the General Petri Net Reachability Problem. In *Proc. of STOC '81*, pages 238–246, Milwaukee, WI, USA, 1981. ACM.

[9] M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.

[10] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Darmstadt, 1962.

[11] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[12] V. V. Ruiz, F. C. Gomez, and D. de Frutos-Escrig. On Non-Decidability of Reachability for Timed-Arc Petri Nets. In *Proc. of PNPM '99*, pages 188–196, Washington DC, USA, 1999. IEEE Computer Society.

[13] J. Srba. Timed-Arc Petri Nets vs. Networks of Timed Automata. In *Proc. of ICATPN'05*, volume 3536 of *LNCS*, pages 385–402. Springer, 2005.

[14] J. Srba. Comparing the Expressiveness of Timed Automata and Timed Extensions of Petri Nets. In *Proc. of FORMATS'08*, volume 5215 of *LNCS*, pages 15–32. Springer, 2008.

Weighted Dynamic Pushdown Networks

Alexander Wenner

Institut für Informatik, Fachbereich Mathematik und Informatik
Westfälische Wilhelms-Universität Münster
alexander.wenner@uni-muenster.de

Abstract. We develop a generic framework for the analysis of programs with recursive procedures and dynamic process creation. To this end we combine the approach of weighted pushdown systems (WPDS) with the model of dynamic pushdown networks (DPN). Weighted dynamic pushdown networks (WDPN) describe processes running in parallel. Each process may perform pushdown actions and spawn new processes. Transitions are labelled by weights to carry additional information. We derive a method to determine meet-over-all-paths values for the paths from a starting configuration to a regular set of configurations of a WDPN.

1 Introduction

The interest in writing parallel programs has increased in recent years. However parallel programming is notoriously difficult and error-prone. Thus static analysis of parallel programs has become more and more important. The goal of this paper is to present a generic framework for the analysis of parallel programs, especially in the presence of recursive procedures and dynamic process creation. We base our framework on DPN [1] and WPDS [2]. DPN precisely model procedures and process creation and have been studied for reachability analyses. Since the analysis of recursive procedures and synchronisation is undecidable [3], DPNs do not model synchronisation between processes. However, through the addition of weights we will be able to analyse some interaction between processes. WPDS extend pushdown systems (PDS) by labelling transitions with weights and solving the generalised pushdown predecessor (GPP) problem, which is the meet-over-all-paths solution for paths from a starting configuration into a regular set of target configurations. The weights can be used to formulate a wide range of analysis problems. The GPP problem formulation allows for a specific query depending on the shape of the entire call-stack, in contrast to standard dataflow techniques, where typically all information at the topmost program point is merged. Analogous to WPDS we extend DPN to WDPN by annotating weights to transitions and study the GPP problem. Even though a WPDS is then simply a WDPN with one process, adapting the approach to solve the GPP problem from WPDS to WDPN is problematic. In general a path of a DPN is an interleaving of the transitions of arbitrary many parallel processes. Results from [1] show, that such a set of paths can not be described using a constraint system. We avoid these problems by introducing a branching semantics for DPN

Mathematical and Engineering Methods in Computer Science (MEMICS), Znojmo, Czech Republic, 2009.
Petr Hliněný, Vashek Matyáš, Tomáš Vojnar (Eds.)
Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/DROPS.MEMICS.2009.2344

similar to the tree semantics in [4]. Transitions of newly spawned processes are no longer mixed with the transitions of the creating process, but contained in their own branch. This results in executions which are tree shaped for single processes and form hedges, which contain a tree for each process, for configurations with multiple processes. We introduce an extended weight domain to abstract these trees, and study the analogous branching GPP (BGPP) problem, which is the meet-over-all-hedges solution, for these branching WDPN (BWDPN). We show, that if the weight domain of a WDPN and the extended weight domain of a BWDPN, based on the same DPN, are related, the solution for the GPP problem of the WDPN can be derived from the solution of the corresponding BGPP problem of the BWDPN. The BGPP problem can be solved using an approach adapted from WPDS.

Up to this point our framework of WDPN and BWDPN can solve the bitvector problems for DPNs formulated in [1], the more general KILL/GEN analyses described in [5] and the shortest path analysis from [2]. In [6] a different approach to generalize WPDS to parallel programs is presented, by introducing a context bound. This leads to an underapproximation, whereas our approach handles unbounded context switches precisely.

The remainder of the paper is organised as follows: Section 2 presents the intuitive extension of WPDS to DPN called WDPN and defines the GPP. Section 3 introduces BWDPN. We formulate the BGPP problem and present the relation to the GPP problem. Section 4 presents two applications and Section 5 presents the approach to solve the BGPP problem for BWDPN.

2 Weighted Dynamic Pushdown Networks

A DPN [1] is a model for parallel programs with multiple processes and dynamic process creation. Each process is modeled as a PDS, where the rules are extended to allow creation of new processes. Formally a DPN is a tuple $\mathcal{M} = (P, \Gamma, \Delta)$, where P is a finite set of control states, Γ is a finite set of stack symbols, with $P \cap \Gamma = \emptyset$, and Δ is a finite set of transition rules of the form $p\gamma \hookrightarrow c$ with $p \in P$, $\gamma \in \Gamma$ and $c \in (P\Gamma^*)^*P\Gamma^*$. The right side of a rule consists of the new control state and stacktop of the original process in the rightmost position and the control states and stacks of all processes spawned by this rule to the left. Configurations of the DPN are words from $\mathsf{Conf} = (P\Gamma^*)^*$. The empty configuration is written as ε. For the rest of the paper we fix a DPN $\mathcal{M} = (P, \Gamma, \Delta)$, a configuration c and a regular set $C \subseteq \mathsf{Conf}$.

An execution of a DPN is represented by a path. A path is defined as a sequence of rules $\rho = r_1 \dots r_n$ with $r_i \in \Delta$. The empty path is denoted by ε_ρ and Paths is the set of all paths. The execution of a path is modeled by the labelled transition relation $\longrightarrow \subseteq \mathsf{Conf} \times \mathsf{Paths} \times \mathsf{Conf}$, where for $c, c' \in \mathsf{Conf}$, $p \in P$, $\gamma \in \Gamma$, $u \in (P\Gamma^*)^*$ and $v \in \Gamma^*(P\Gamma^*)^*$:

$$[\mathsf{empty}]\ c \xrightarrow{\varepsilon_\rho} c \qquad [\mathsf{rule}]\ up\gamma v \xrightarrow{r\rho} c \ \text{ if } r = p\gamma \hookrightarrow c' \text{ and } uc'v \xrightarrow{\rho} c$$

Application of a rule replaces the control state and top symbol of one stack by the new control state and stacktop specified by the rule and inserts the newly created processes with their initial stacks to the left. We call this the interleaving semantics of the DPN, since the rules of all processes are mixed together. We are interested in the set $\mathsf{Paths}(c, C) = \{\rho \in \mathsf{Paths} \mid \exists c' \in C \text{ with } c \xrightarrow{\rho} c'\}$ of connecting paths from c to C.

In order to abstract from the set of connecting paths to the aspects which are relevant to the desired analysis, we assign a weight to each transition of the DPN. The structure of the weight domain is captured by a complete idempotent semiring, which supports the necessary operators $\odot$ for concatenation of weights along a path and $\oplus$ for combination of weights of different paths. A complete idempotent semiring is a tuple $\mathcal{S} = (D, \oplus, \odot, 0, 1)$, where D is a set of elements with $0, 1 \in D$ and $\oplus, \odot$ are binary operators on D with:

- $(D, \oplus)$ is a commutative monoid with neutral element 0 and $\oplus$ is idempotent
- $(D, \odot)$ is a monoid with neutral element 1 and 0 annihilates $\odot$
- $(D, \sqsubseteq)$ is a complete lattice, where $\sqsubseteq$, with $d_1 \sqsubseteq d_2 :\Leftrightarrow d_1 \oplus d_2 = d_1$ for $d_1, d_2 \in D$, is the partial order induced by $\oplus$
- $\odot$ distributes over arbitrary $\oplus$, i.e. $\bigoplus D_1 \odot \bigoplus D_2 = \bigoplus\{d_1 \odot d_2 \mid d_i \in D_i\}$ for $D_1, D_2 \subseteq D$

Furthermore we assume, that a weight function $f : \Delta \to D$ is given. The weight function assigns a weight to each transition of our DPN and depends on the analysis, since it describes how the transitions of the DPN are connected to the analysed information represented by the semiring. We fix the tuple $\mathcal{W} = (\mathcal{M}, \mathcal{S}, f)$, with $\mathcal{S} = (D, \oplus, \odot, 0, 1)$, called a WDPN, for the rest of the paper and define an abstraction function $\alpha : \mathsf{Paths} \to D$ for paths:

$$[\mathsf{empty}]\ \alpha(\varepsilon_\rho) = 1 \quad [\mathsf{rule}]\ \alpha(r\rho) = f(r) \odot \alpha(\rho)$$

Overloading it for sets of paths with $\alpha(M) = \bigoplus\{\alpha(\rho) \mid \rho \in M\}$, we can formulate the GPP problem for WDPN as computing $\delta(c, C) = \alpha(\mathsf{Paths}(c, C))$.

3 Branching Weighted Dynamic Pushdown Networks

It follows from results in [1] that the set $\mathsf{Paths}(c, C)$ can not be characterised as least solution of a constraint system. Therefore we can not compute the solution for the GPP problem directly by an abstract interpretation [7] of such a constraint system. To avoid these problems we consider an alternative interpretation of an execution of a DPN in form of a tree or hedge, first introduced in [4]. The set of connecting hedges can then be described using a constraint system.

We recursively define the sets Trees and $\mathsf{Hedges} = \mathsf{Trees}^*$ of execution trees and hedges. The empty hedge is written as ε_σ. The empty tree ε_τ consisting of a single leaf node, representing a finished execution, is a tree. $r(\sigma\tau)$ is a tree with a root node labelled with a rule $r \in \Delta$, describing the first step of the execution, and an ordered list of subtrees $\sigma\tau \in \mathsf{Hedges}$, representing the executions σ of spawned processes and the rest of the execution τ of the spawning process. We define the **;** operator to concatenate a tree to the last tree of a hedge:

$$[\mathsf{hedge}]\ (\sigma\tau)\,\mathbf{;}\,\tau' = \sigma(\tau\,\mathbf{;}\,\tau') \quad [\mathsf{empty}]\ \varepsilon_\tau\,\mathbf{;}\,\tau' = \tau' \quad [\mathsf{rule}]\ r(\sigma)\,\mathbf{;}\,\tau' = r(\sigma\,\mathbf{;}\,\tau')$$

Appending a tree replaces the rightmost leaf of the hedge with that tree. Thus concatenation of trees is concatenation of the rightmost branches. Since the rightmost branch represents the execution of the initial process, this will later be used to assemble execution trees from partial executions of an initial process. The execution of a hedge is modeled by the labelled transition relation $\Longrightarrow \subseteq \mathsf{Conf} \times \mathsf{Hedges} \times \mathsf{Conf}$, where for $c, c', \tilde{c} \in \mathsf{Conf}$, $p \in P$, $\gamma \in \Gamma$ and $w \in \Gamma^*$:

$$\begin{array}{llll} [\mathsf{none}] & \varepsilon \overset{\varepsilon_\sigma}{\Longrightarrow} \varepsilon & [\mathsf{tree}]\ cpw \overset{\sigma\tau}{\Longrightarrow} c'\tilde{c} & \text{if } c \overset{\sigma}{\Longrightarrow} c' \text{ and } pw \overset{\tau}{\Longrightarrow} \tilde{c} \\ [\mathsf{empty}] & pw \overset{\varepsilon_\tau}{\Longrightarrow} pw & [\mathsf{rule}]\ p\gamma w \overset{r(\sigma)}{\Longrightarrow} c & \text{if } r = p\gamma \hookrightarrow c' \text{ and } c'w \overset{\sigma}{\Longrightarrow} c \end{array}$$

We call this the branching semantics of the DPN, since each process has its own branch in the execution. We are interested in the set $\mathsf{Hedges}(c, C) = \{\sigma \in \mathsf{Hedges} \mid \exists c' \in C \text{ with } c \overset{\sigma}{\Longrightarrow} c'\}$ of connecting hedges.

To abstract hedges we define an extended complete idempotent semiring, which contains the additional $\bar{\otimes}$ operator for parallel combination of weights. An extended complete idempotent semiring $\mathcal{E} = (E, \bar{\oplus}, \bar{\odot}, \bar{\otimes}, \bar{0}, \bar{1})$ is a tuple, where E is a set of values and $\bar{\oplus}, \bar{\odot}, \bar{\otimes}$ are binary operators on E with:

- $(E, \bar{\oplus}, \bar{\odot}, \bar{0}, \bar{1})$ is a complete idempotent semiring
- $(E, \bar{\otimes})$ is a semigroup, $\bar{1} \,\bar{\otimes}\, e = e$ for $e \in E$ and $\bar{0}$ annihilates $\bar{\otimes}$
- $\bar{\otimes}$ distributes over arbitrary $\bar{\oplus}$, i.e. $\bar{\bigoplus} E_1 \,\bar{\otimes}\, \bar{\bigoplus} E_2 = \bar{\bigoplus}\{e_1 \,\bar{\otimes}\, e_2 \mid e_i \in E_i\}$ for $E_1, E_2 \subseteq E$
- $(e_1 \,\bar{\otimes}\, e_2) \,\bar{\odot}\, e_3 = e_1 \,\bar{\otimes} (e_2 \,\bar{\odot}\, e_3)$, for $e_1, e_2, e_3 \in E$

The fourth property ensures, that **;** is abstracted by $\bar{\odot}$, by always appending weights to the rightmost weight of a parallel combination. In this regard the $\bar{\otimes}$ operator differs from the interleaving operator $\otimes$ introduced in [8], since weights that are concatenated after an interleaving need to be considered as well.

Furthermore we assume, as with WDPN, that a weight function $\bar{f} : \Delta \to E$ is given. We fix the tuple $\mathcal{B} = (\mathcal{M}, \mathcal{E}, \bar{f})$, with $\mathcal{E} = (E, \bar{\oplus}, \bar{\odot}, \bar{\otimes}, \bar{0}, \bar{1})$, called BWDPN, for the rest of the paper and define an abstraction function $\beta : \mathsf{Hedges} \to E$ for hedges:

$$\begin{array}{llll} [\mathsf{none}] & \beta(\varepsilon_\sigma) = \bar{1} & [\mathsf{tree}]\ \beta(\sigma\tau) & = \beta(\sigma) \,\bar{\otimes}\, \beta(\tau) \\ [\mathsf{empty}] & \beta(\varepsilon_\tau) = \bar{1} & [\mathsf{rule}]\ \beta(r(\sigma)) & = \bar{f}(r) \,\bar{\odot}\, \beta(\sigma) \end{array}$$

Overloading it for sets of hedges with $\beta(M) = \bar{\bigoplus}\{\beta(\sigma) \mid \sigma \in M\}$, we define the BGPP problem for BWDPN as computing $\theta(c, C) = \beta(\mathsf{Hedges}(c, C))$.

There is a strong connection between the interleaving and branching semantics of a DPN. A hedge represents of a set of paths, which can be constructed by interleaving the branches and trees of the hedge. In [4] it was shown, that if we take a function $\psi : \mathsf{Hedges} \to 2^{\mathsf{Paths}}$ that computes the set of interleavings of a hedge, and overload it for sets of hedges, we have:

Theorem 1. $\mathsf{Paths}(c, C) = \psi(\mathsf{Hedges}(c, C))$

A similar result can be shown for the solutions of the GPP and BGPP problems, if the semiring of the WDPN is related to the extended semiring of the BWDPN. We describe the necessary relation by an extension. An extension is a tuple $(\mathcal{S}, \mathcal{E}, \iota, \eta)$, containing embedding and projection functions $\iota : D \to E$ and $\eta : E \to D$, where for $d, d_i \in D, e, e_i \in E$ the following conditions must hold:

- E is the smallest set with $\iota(D) \subseteq E$, closed under $\bar{\odot}, \bar{\otimes}$ and arbitrary $\bar{\oplus}$
- $\iota(0) = \bar{0}$, $\iota(1) = \bar{1}$ and $\eta(\iota(d)) = d$
- η distributes over arbitrary $\bar{\oplus}$, i.e. $\eta(\bar{\bigoplus} M) = \bigoplus\{\eta(e) \mid e \in M\}$ for $M \subseteq E$
- $\eta(\iota(d) \mathbin{\bar{\odot}} e) = d \odot \eta(e)$
- $\eta(e_1 \mathbin{\bar{\otimes}} \ldots \mathbin{\bar{\otimes}} e_n) = \eta(e_{i_1} \mathbin{\bar{\otimes}} \ldots \mathbin{\bar{\otimes}} e_{i_m})$ with $e_i = \bar{1}$ for $i \notin \{i_1, \ldots, i_m\}$
- $\eta(e_1 \mathbin{\bar{\otimes}} \ldots \mathbin{\bar{\otimes}} e_n) = \bigoplus_{i=1}^{n} d_i \odot \eta(e_1 \mathbin{\bar{\otimes}} \ldots \mathbin{\bar{\otimes}} e'_i \mathbin{\bar{\otimes}} \ldots \mathbin{\bar{\otimes}} e_n)$ with $e_i = \iota(d_i) \mathbin{\bar{\odot}} e'_i$

The first three points ensure, that every weight of the original semiring has a corresponding weight in the extended semiring. The fourth point guarantees, that a simple concatenation of extended weights is mapped to the corresponding concatenation of weights. The last two points ensure, that the parallel combination of extended weights is mapped to the meet over all interleavings of the weights they are constructed from. For the rest of the paper, we assume that the semiring and extended semiring are connected by the extension $(\mathcal{S}, \mathcal{E}, \iota, \eta)$.

If $\bar{f}(r) = \iota(f(r))$, for all $r \in \Delta$, i.e. the analysis of the WDPN is embedded in the BWDPN, we can proof $\alpha(\psi(\sigma)) = \eta(\beta(\sigma))$ for all $\sigma \in \mathsf{Hedges}$ by induction on σ. Consequently with Theorem 1 we have:

Theorem 2. $\delta(c, C) = \eta(\theta(c, C))$.

4 Applications

Since the existence of an extended semiring and a matching extension for a given semiring is not self-evident, we first give two examples of semirings, for which an extended semiring and a corresponding extension can be constructed, before describing the approach to solve the BGPP problem in Section 5.

The shortest path analysis assigns a positive integer weight to all transitions. The weight of a path is the sum of the weights of the transitions occurring on the path. The goal is to find the weight of the path with the smallest weight. We use the semiring $\mathcal{S} = (\mathbb{N} \cup \{0, \infty\}, min, +, \infty, 0)$ introduced in [2]. Since $+$ is commutative and associative, the order in which transitions occur and are combined on a path is irrelevant. Thus $+$ can be used as the interleaving operator $\bar{\otimes}$ in an extended semiring. The semiring in combination with the interleaving operator fulfills all necessary conditions for an extended semiring $\mathcal{E} = (\mathbb{N} \cup \{0, \infty\}, min, +, +, \infty, 0)$ and the matching extension is simply $(\mathcal{S}, \mathcal{E}, id, id)$.

Bitvector Analyses analyse a property represented by a single bit. For lack of space, we consider only forward may bitvector analysis. Backward or must analyses can be handled similarly. The transitions of the DPN are annotated with transformers, that change the current state of the bit. We use the semiring

$\mathcal{S} = (D, \oplus, \odot, \mathsf{zero}, \mathsf{id})$, where $D = \{\mathsf{gen}, \mathsf{id}, \mathsf{kill}, \mathsf{zero}\}$. Here gen represents the transformer setting the bit to 1, id is the identity and kill sets the bit to 0. The artificial weight zero is introduced to represent the zero element of the ring. $\odot$ is reversed functional concatenation extended to include zero. $\oplus$ is a meet operator inducing the ordering $\mathsf{gen} \sqsubseteq \mathsf{id} \sqsubseteq \mathsf{kill} \sqsubseteq \mathsf{zero}$. In [8] it was shown, that the operator $\otimes$, defined as $f \otimes g = (f \odot g) \oplus (g \odot f)$, is an interleaving operator on the path level. However the semiring in combination with the interleaving operator can not be used as extended semiring, since it does not fulfill the property $(f \otimes g) \odot h = f \otimes (g \odot h)$ for all $f, g, h \in D$. Especially for $f = \mathsf{gen}, g = \mathsf{id}$ and $h = \mathsf{kill}$, the terms $(f \otimes g) \odot h = \mathsf{kill}$ and $f \otimes (g \odot h) = \mathsf{gen}$ evaluate differently. This is caused by the fact, that a gen occurring in a parallel process can always be executed last in an interleaving and reset the bit. However the operator $\otimes$ does only consider the gen to be parallel to g and not the later appended h. We solve this problem by introducing a new weight $\overline{\mathsf{gen}}$, that stores the information, that a gen weight was encountered in parallel. This leads to the extended semiring $(\{\overline{\mathsf{gen}}, \mathsf{gen}, \mathsf{id}, \mathsf{kill}, \mathsf{zero}\}, \bar{\oplus}, \bar{\odot}, \bar{\otimes}, \mathsf{zero}, \mathsf{id})$ and extension $(\mathcal{S}, \mathcal{E}, \iota, \eta)$, where $\bar{\oplus}$ induces the ordering $\overline{\mathsf{gen}} \sqsubseteq \mathsf{gen} \sqsubseteq \mathsf{id} \sqsubseteq \mathsf{kill} \sqsubseteq \mathsf{zero}$ and:

$$f \mathbin{\bar{\odot}} g = \begin{cases} f \odot g & \text{if } f, g \neq \overline{\mathsf{gen}} \\ \overline{\mathsf{gen}} & \text{if } f = \overline{\mathsf{gen}} \text{ or } g = \overline{\mathsf{gen}} \end{cases} \qquad f \mathbin{\bar{\otimes}} g = \begin{cases} f \mathbin{\bar{\odot}} g & \text{if } f \notin \{\overline{\mathsf{gen}}, \mathsf{gen}\} \\ \overline{\mathsf{gen}} & \text{if } f \in \{\overline{\mathsf{gen}}, \mathsf{gen}\} \end{cases}$$

$$\eta(f) = \begin{cases} f & \text{if } f \in \{\mathsf{gen}, \mathsf{id}, \mathsf{kill}, \mathsf{zero}\} \\ \mathsf{gen} & \text{if } f = \overline{\mathsf{gen}} \end{cases} \qquad \iota(f) = f$$

5 Solving the BGPP Problem for BWDPN.

We use $\mathcal{M}$- and $\mathcal{M}^*$-automata, adapted from [1], as a compact representation for the target set. A $\mathcal{M}^*$-automaton is a finite automaton $\mathcal{A}^* = (S, P \cup \Gamma, \delta, \dot{s}, F)$ that satisfies the following additional conditions:

- $S_c, S_p \subseteq S$, where for all $s \in S_c, p \in P$ exists a unique and distinguished state $s_p \in S_p$
- $\delta = \delta_P \cup \delta_\Gamma$ where $\delta_P = \{(s, p, s_p) \mid s \in S_c, p \in P\}$ and $\delta_\Gamma \subseteq S \times (\Gamma \cup \{\varepsilon\}) \times S$
- $\mathcal{L}(\mathcal{A}^*) \subseteq \mathsf{Conf}$

A $\mathcal{M}$-automaton $\mathcal{A}$ is a $\mathcal{M}^*$-automaton, where $\dot{s} \in S \setminus S_p$ and the transition relation δ satisfies the stronger condition $\delta_\Gamma \subseteq S \times (\Gamma \cup \{\varepsilon\}) \times (S \setminus S_p)$. We write $s \xrightarrow{\lambda}_\delta s'$ for $(s, \lambda, s') \in \delta$ and $s \xrightarrow{c}{}^{*}_{\delta} s'$ for the transitive closure. For the rest of the paper we fix an $\mathcal{M}$-automaton $\mathcal{A} = (S, P \cup \Gamma, \delta, \dot{s}, F)$ describing the set C.

Now consider an execution hedge in $\mathsf{Hedges}(c, C)$. Each tree of the hedge transforms a stack in c into a configuration containing the transformed original stack and stacks of spawned processes. Analogous to the approach in [2], we can split each tree into several phases along the rightmost branch, each transforming a stacksymbol of the corresponding initial stack. During these transformation new processes may be spawned and transformed themselves. The idea is to compute for each symbol of the starting configuration the set of trees, that transform the stack symbol into a configuration, that is part of a configuration in C.

To this end we take a closer look at the saturation procedure used in [1] to construct the set $PRE^*(C) = \{c \mid \exists c' \in \mathsf{Conf}, \sigma \in \mathsf{Hedges} \text{ with } c \overset{\sigma}{\Longrightarrow} c'\}$ of all predecessor configurations. The saturation procedure works by adding new transitions to the automaton $\mathcal{A}$, thus allowing more configurations to be accepted. The result is a $\mathcal{M}^*$-automaton $\mathcal{A}^* = (S, P \cup \Gamma, \delta', \dot{s}, F)$, with $\delta' = \delta_P \cup \delta'_\Gamma$, where δ'_Γ is the smallest set fulfilling the conditions:

$$\begin{array}{lrl} [\mathsf{init}] & t \in \delta'_\Gamma & \text{if } t \in \delta_\Gamma \\ [\mathsf{step}] & (s_p, \gamma, s') \in \delta'_\Gamma & \text{if } r = p\gamma \hookrightarrow c \in \Delta, s \in S_c \text{ and } s \xrightarrow{c}{}^{*}_{\delta'} s' \end{array}$$

A transition is added, if there is a rule transforming the symbol into a configuration which can be read by previously existing transitions. If these transitions were also added by the saturation, they themselves have a rules, which transform their symbols. If we follow this recursion and assemble the rules into a tree, we have a tree that transform the symbol of the newly added transition into a configuration that can be read using only transition of $\mathcal{A}$ and therefore is part of a configuration in C. We extend the saturation procedure to keep track of these trees by constructing a constraint system L over $(2^{\mathsf{Trees}}, \cup)$. The variables of the constraint system $\mathsf{L}[t]$ with $t \in \delta'_\Gamma$ can be seen as annotations to the transitions of the saturated automaton. Additionally we define a function $\pi_\mathsf{L} : S \times \mathsf{Conf} \times S \to 2^{\mathsf{Hedges}}$ that constructs a set of hedges for a configuration by reading the annotations from the automaton:

$$\begin{array}{lll} [\mathsf{empty}] & \pi_\mathsf{L}(s, \varepsilon, s') & = \begin{cases} \{\varepsilon_\sigma\} & \text{if } s \xrightarrow{\varepsilon}{}^{*}_{\delta'} s' \\ \emptyset & \text{else} \end{cases} \\ [\mathsf{control}] & \pi_\mathsf{L}(s, cp, s') & = \bigcup\{\pi_\mathsf{L}(s, c, \tilde{s})\varepsilon_\tau \mid s \xrightarrow{c}{}^{*}_{\delta'} \tilde{s} \xrightarrow{p}_{\delta'_P} \hat{s} \xrightarrow{\varepsilon}{}^{*}_{\delta'} s'\} \\ [\mathsf{stack}] & \pi_\mathsf{L}(s, c\gamma, s') & = \bigcup\{\pi_\mathsf{L}(s, c, \tilde{s}) \,;\, \mathsf{L}[(\tilde{s}, \gamma, \hat{s})] \mid s \xrightarrow{c}{}^{*}_{\delta'} \tilde{s} \xrightarrow{\gamma}_{\delta'_\Gamma} \hat{s} \xrightarrow{\varepsilon}{}^{*}_{\delta'} s'\} \end{array}$$

ε transitions do not contribute any information and are simply skipped. If a control state is encountered a new empty tree is added to the current hedges for the following new process stack. In case of a stack symbol, the trees which transform the stack symbol are appended to the current hedges. By appending the trees of the individual stack symbols, we get a single tree, that transforms the whole stack.

We construct a set of constraints in a similar way the saturation procedure adds transitions to the automaton. Here $r(\cdot) : 2^{\mathsf{Hedges}} \to 2^{\mathsf{Trees}}$ are operators generating new trees out of a root node labelled with $r \in \Delta$ and lists of subtrees from a given set:

$$\begin{array}{lll} [\mathsf{init}] & \mathsf{L}[t] \supseteq \{\varepsilon_\tau\} & \text{if } t \in \delta_\Gamma \\ [\mathsf{step}] & \mathsf{L}[(s_p, \gamma, s')] \supseteq r(\pi_\mathsf{L}(s, c, s')) & \text{if } r = p\gamma \hookrightarrow c \in \Delta, s \in S_c \text{ and } s \xrightarrow{c}{}^{*}_{\delta'} s' \end{array}$$

If we annotate the transitions of $\mathcal{A}^*$ with the least solution $\mathsf{lfp}(\mathsf{L})$ of L we can proof, by induction on the length of c, that the solution of the constraint system can be used to describe the set of all connecting hedges:

Theorem 3. $\mathsf{Hedges}(c, C) = \bigcup\{\pi_{\mathsf{lfp}(\mathsf{L})}(\dot{s}, c, s) \mid s \in F\}$.

To compute the weight of the hedges, we construct a constraint system $\mathsf{L}^{\#}$ and a function $\pi^{\#}_{\mathsf{L}^{\#}}$ over the weight domain by replacing the operators and constants in the constraint system L and the function π_{L}, with the corresponding operators and constants according to the abstraction function β. By standard results from abstract interpretation [7], we get $\mathsf{lfp}(\mathsf{L}^{\#}) = \beta(\mathsf{lfp}(\mathsf{L}))$ and $\pi^{\#}_{\mathsf{lfp}(\mathsf{L}^{\#})} = \beta(\pi_{\mathsf{lfp}(\mathsf{L})})$ and with Theorem 3 we have:

Theorem 4. $\theta(c, C) = \bar{\bigoplus}\{\pi^{\#}_{\mathsf{lfp}(\mathsf{L}^{\#})}(\dot{s}, c, s) \mid s \in F\}$.

Thus we can solve the BGPP problem by solving for $\mathsf{lfp}(\mathsf{L}^{\#})$ using standard techniques and evaluating $\pi^{\#}_{\mathsf{lfp}(\mathsf{L}^{\#})}$. Theorem 2 states, that we get the solution to the GPP problem by applying η.

6 Conclusion

We presented the GPP problem for a WDPN, which is a model for parallel programs with dynamic process creation and recursive procedures. The GPP problem is a general problem formulation, which can, for example, be used to capture basic dataflow analysis problems. Since the GPP problem can not be solved directly, our approach is based on an alternative branching semantics for DPN. The resulting tree shaped executions can be characterised using a constraint system, which can then be solved over an abstract domain to get a solution for the BGPP problem for BWDPN. If the weight domains for the BWDPN and WDPN are connected through an extension, the solution for the GPP problem can be derived from the corresponding BGPP problem. We have shown how the results can be used to solve basic dataflow analysis problems like bitvector analyses or shortest path problems.

References

1. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: CONCUR. LNCS 3653, Springer (2005)
2. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comp. Prog. **58**(1-2) (2005)
3. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. **22**(2) (2000)
4. Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor sets of dynamic pushdown networks with tree-regular constraints. In: CAV. LNCS 5643, Springer (2009)
5. Lammich, P., Müller-Olm, M.: Precise fixpoint-based analysis of programs with thread-creation and procedures. In: CONCUR. LNCS 4703 (2007)
6. Lal, A., Touili, T., Kidd, N., Reps, T.W.: Interprocedural analysis of concurrent programs under a context bound. In: TACAS. LNCS 4963, Springer (2008)
7. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, ACM Press (1977)
8. Seidl, H., Steffen, B.: Constraint-based inter-procedural analysis of parallel programs. Nordic J. of Computing **7**(4) (2000)