Volume 21 Number 2

ACTA CYBERNETICA

Szeged, 2013

Ferenc Gécseg
Department of Computer Algorithms
and Artificial Intelligence
University of Szeged
Szeged, Hungary
gecseg@inf.u-szeged.hu

Jozef Gruska
Institute of Informatics/Mathematics
Slovak Academy of Science
Bratislava, Slovakia
gruska@savba.sk

Tibor Gyimóthy
Department of Software Engineering
University of Szeged
Szeged, Hungary
gyimothy@inf.u-szeged.hu

Helmut Jürgensen
Department of Computer Science
Middlesex College
The University of Western Ontario
London, Canada
helmut@csd.uwo.ca

Zoltan Kato
Department of Image Processing
and Computer Graphics
Szeged, Hungary
kato@inf.u-szeged.hu

Alice Kelemenová
Institute of Computer Science
Silesian University at Opava
Opava, Czech Republic
Alica.Kelemenova@fpf.slu.cz

László Lovász
Department of Computer Science
Eötvös Loránd University
Budapest, Hungary
lovasz@cs.elte.hu

Gheorghe Păun
Institute of Mathematics of the
Romanian Academy
Bucharest, Romania
George.Paun@imar.ro

András Prékopa
Department of Operations Research
Eötvös Loránd University
Budapest, Hungary
prekopa@cs.elte.hu

Arto Salomaa
Department of Mathematics
University of Turku
Turku, Finland
asalomaa@utu.fi

László Varga
Department of Software Technology
and Methodology
Eötvös Loránd University
Budapest, Hungary
varga@ludens.elte.hu

Heiko Vogler
Department of Computer Science
Dresden University of Technology
Dresden, Germany
Heiko.Vogler@tu-dresden.de

Gerhard J. Woeginger
Department of Mathematics and
Computer Science
Eindhoven University of Technology
Eindhoven, The Netherlands
gwoegi@win.tue.nl

Acta Cybernetica **21** (2013) 205–222.

Distinguishing Experiments for Timed Nondeterministic Finite State Machines*

Khaled El-Fakih[†] Maxim Gromov[‡] Natalia Shabaldina[‡] and Nina Yevtushenko[‡]

Abstract

The problem of constructing distinguishing experiments is a fundamental problem in the area of finite state machines (FSMs), especially for FSM-based testing. In this paper, the problem is studied for timed nondeterministic FSMs (TFSMs) with output delays. Given two TFSMs, we derive the TFSM intersection of these machines and show that the machines can be distinguished using an appropriate (untimed) FSM abstraction of the TFSM intersection. The FSM abstraction is derived by constructing appropriate partitions for the input and output time domains of the TFSM intersection. Using the obtained abstraction, a traditional FSM-based preset algorithm can be used for deriving a separating sequence for the given TFSMs if these machines are separable. Moreover, as sometimes two non-separable TFSMs can still be distinguished by an adaptive experiment, based on the FSM abstraction we present an algorithm for deriving an r-distinguishing TFSM that represents a corresponding adaptive experiment.

Keywords: nondeterministic untimed and timed finite state machines, preset and adaptive distinguishing experiments, state identification

1 Introduction

Finite State Machines (FSMs) are widely used for modeling systems in many application domains. For instance, (Mealy) FSMs are used as the underlying models for formal description techniques such as SDL and UML State Diagrams. In many cases, the behavior of a given machine can be considered as a mapping of input sequences (sequences of input symbols) to corresponding output sequences (sequences

*This work was partially supported by AUS FRG-III and Russian ministry of Science and High Education (contract No. 14.B37.21.0622)

[†]American University of Sharjah, Department of Computer Science and Engineering, PO Box 26666, Sharjah, UAE, Tel: (971) 06 5152492, Mobile: (971) 050 3073091 Fax: (971) 6 515 2979, E-mail: kelfakih@aus.edu

[‡]Tomsk State University, 36 Lenin Str., Tomsk, 634050, Russia, E-mail: gromov@sibmail.com, NataliaMailBox@mail.ru, ninayevtushenko@yahoo.com

of output symbols). A machine is *deterministic* if it produces a single output sequence in response to an input sequence and a machine is *nondeterministic* if it can produce several output sequences in response to an input sequence. Nondeterminism may occur due to various reasons such as limited controllability, abstraction level, modeling concurrency and real time systems, etc. [1, 7, 21].

When distinguishing FSMs, we have a machine under test about which we lack some information, and we want to deduce this information by conducting experiments on this machine. An experiment consists of applying input sequences to the machine, observing corresponding output responses and drawing some conclusions about the machine under test. An experiment is *simple* if a single input sequence is applied to a machine under experiment; otherwise, the experiment is referred to as a *multi* experiment. An experiment is *preset* if input sequences are known before starting the experiment and an experiment is *adaptive* if at each step of the experiment the next input is selected based on previously observed outputs. Distinguishing experiments with FSMs are widely used as a basis for solving fundamental testing problems such as the fault detection (or conformance testing) and/or the machine identification problems. For related surveys and algorithms on FSM-based distinguishing experiments, the reader may refer to [2–4, 9, 11–13].

Unlike deterministic FSMs, for nondeterministic FSMs, there are a number of distinguishability relations, other than the equivalence relation, such as the *non-reduction*, *separability*, and *r-distinguishability* relations [1, 16, 20]. Two machines can be distinguished by a simple preset experiment if these machines are separable. The separability relation is defined by Starke in [20] and studied in [1] and [19]. Two nondeterministic machines are *separable* if there is an input sequence, called a *separating sequence*, such that the sets of output responses of the machines to the input sequence are disjoint. Thus, two separable machines can be distinguished by applying a separating sequence only once. Two complete non-separable machines still can be distinguished by a simple adaptive experiment if they are *r-distinguishable*, i.e., if they have no common complete reduction [17, 23]. A machine is a *reduction* of another machine if its behavior is contained in the behavior of the other machine.

Currently, models of many systems such as telecommunication systems, plant and traffic controllers etc, take into account time constraints and correspondingly timed FSMs are getting a lot of attention. Merayo et al. [5, 14, 15] consider a timed possibly nondeterministic FSM model where time constrains limit a time elapsed when an output has to be produced after an input has been applied to the FSM. Hierons et al. [8] introduce a timed stochastic FSM model. Gromov et al. [6] consider a timed complete nondeterministic FSM model where transitions are guarded by time constraints over a single clock. The clock is reset at the execution of a transition. In this paper, we consider a model similar to that in [6], yet extended to deal with non-zero output delays sometimes called output timeouts. The considered model can be regarded as a temporal extension of FSMs where a transition is fired only if a given input is given in time (bounded by given lower and upper bounds) that is counted from the moment when a current state is reached. Firing a transition also takes time between the reception of the input and the emission of the output, i.e., the output delay represents the transition execution/processing time. In the

considered model, the identification of input and output time domains of a state can be done independent of time domains of other states, and thus, there are technical benefits in using the considered model for distinguishability and testing.

Given two possibly nondeterministic timed FSMs, we study the problem of deriving an input sequence that distinguishes these machines. At the first step, the TFSM intersection of the given two machines is derived from which an FSM abstraction is then constructed. It is shown that distinguishing experiments for the given timed FSMs can be determined based on the constructed FSM abstraction. In particular, we show how a traditional preset FSM-based method can be adapted for the FSM abstraction of the intersection when deriving a separating sequence for two given timed FSMs. In addition, using the FSM abstraction we present an algorithm for deriving an r-distinguishing TFSM that represents an adaptive distinguishing experiment for the given two TFSMs if the machines are r-distinguishable.

This paper extends a related preliminary work in [6] to TFSMs which can have non-zero output delays. Moreover, the presented work provides a simpler strategy for deriving distinguishing experiments. In particular, in [6] two TFSMs are distinguished based on their intersection using more complex algorithms that inherit ideas from traditional untimed FSM methods mixed with the derivation of appropriate partitions of input domains for handling time constraints. The strategy proposed in this paper is based on a corresponding (untimed) FSM abstraction of the intersection of two TFSMs and this allows simpler adaptation of existing traditional FSM-based methods for distinguishing TFSMs. The methods presented in this paper and in [6] produce experiments of the same length as the FSM abstraction has the same number of states as the TFSM intersection of the given two machines.

We note that another possible strategy for distinguishing two given TFSMs using algorithms for untimed machines is to first build an FSM abstraction for each of the given machines, derive the intersection of the obtained FSM abstractions, and afterwards, tune traditional FSM-based methods for deriving distinguishing sequences and their corresponding timed sequences using the obtained FSM intersection and the given TFSMs. However, in this case, the number of (abstract) inputs and outputs of the FSM abstractions and their intersection are larger than those derived using our proposed strategy. This is due to the fact that in this case the derivation time domains of inputs and outputs has to be carried out considering all the states of the given machines whereas it is sufficient to consider, as in our approach, pairs of states that appear in the intersection of the given machines.

Finally, it is worth stating that in [10] some work has been presented for distinguishing Timed Input/Output Automata (TIOA) with multiple clocks. Given a TIOA and a clock model, the product of the given automaton with the clock model is transformed into a so-called Bisimulation Quotient Graph, and afterwards, the obtained graph is transformed into a special possibly nondeterministic (untimed) Mealy machine which is actually a tranducer over sequences of abstract inputs and outputs written as regular languages. However, a distinguishing sequence derived from the obtained tranducer in [10] cannot be applied to distinguishing states of the original timed machine since the regular languages (corresponding to sequences of

abstract outputs) labeling transitions of the obtained Mealy machine may intersect, and thus, corresponding states of the initial automaton cannot be separated. In addition, the obtained Mealy machine can be non-observable, and thus the traditional FSM method given in [1] cited in [10] cannot be applied.

This paper is organized as follows. Section 2 includes preliminaries and Section 3 presents the FSM abstraction and distinguishability algorithms. Section 4 concludes the paper.

2 Preliminaries

An FSM S[1] is a 5-tuple $\langle S, I, O, \lambda_S, \hat{s}\rangle$, where S, I and O are finite sets of states, inputs and outputs, respectively, $\hat{s}$ is the initial state and $\lambda_S \subseteq S \times I \times O \times S$ is a transition relation. A timed FSM (TFSM) S or simply a timed machine is a 5-tuple $\langle S, I, O, \lambda_S, \hat{s}\rangle$ with the transition relation $\lambda_S \subseteq S \times (I \times \Pi) \times (O \times \aleph) \times S$, where Π is the set of input time guards and $\aleph$ is the set of output time guards for representing output delays. Each guard $g \in \Pi = \lceil min, max\rceil$ (each guard $f \in \aleph = \lceil min, max\rceil$) where min is a nonnegative integer, while max is a nonnegative integer or the infinity, $min \leqslant max$, and $\lceil \in \{(, [\}$ while $\rceil \in \{),]\}$. From the practical point of view, we assume that all the output guards have a finite upper bound $\mathbf{B}$. For every pair $\langle s, i\rangle \in S \times I$, we use $G_{\langle s,i\rangle}$ to denote the collection of input time guards g such that there is a transition $\langle s, \langle i, g\rangle, \langle o, f\rangle, s'\rangle \in \lambda_S$ and for every pair $\langle s, o\rangle \in S \times O$ we use $G_{\langle s,o\rangle}$ to denote the collection of output time guards f such that there is a transition $\langle s, \langle i, g\rangle, \langle o, f\rangle, s'\rangle \in \lambda_S$.

The behavior of a TFSM S can be described as follows. If $\langle s, \langle i, g\rangle, \langle o, f\rangle, s'\rangle \in$ $\in \lambda_S$, where $g = \lceil m_1, m_2\rceil$ and $f = \lceil n_1, n_2\rceil$, we say that TFSM S being at state s *accepts* input i applied at time $t \in \lceil m_1, m_2\rceil$ measured from the moment TFSM S entered state s; the clock then is set to zero, and S *responds* with (or *produces*) output o after t' time units, $t' \in \lceil n_1, n_2\rceil$, and time is set to zero as S enters state s'.

A TFSM S is *observable* if for each two transitions $\langle s, \langle i, \lceil m_1, m_2\rceil\rangle, \langle o, \lceil n_1, n_2\rceil\rangle, s'\rangle \in \lambda_S$ and $\langle s, \langle i, \lceil m_1', m_2'\rceil\rangle, \langle o, \lceil n_1', n_2'\rceil\rangle, s''\rangle \in \lambda_S$ it holds that if $\lceil m_1, m_2\rceil \cap \lceil m_1', m_2'\rceil \neq \varnothing$ and $\lceil n_1, n_2\rceil \cap \lceil n_1', n_2'\rceil \neq \varnothing$, then $o' = o$ implies $s' = s''$. In this paper, we consider only observable TFSMs as similar to untimed FSMs, for every unobservable timed machine there exists an observable timed machine that has the same behavior.

TFSM S is (*time*) *deterministic* if for each two transitions $\langle s, \langle i, \lceil m_1, m_2\rceil\rangle, \langle o, \lceil n_1, n_2\rceil\rangle, s'\rangle \in \lambda_S$, $\langle s, \langle i, \lceil m_1', m_2'\rceil\rangle, \langle o', \lceil n_1', n_2'\rceil\rangle, s''\rangle \in \lambda_S$, $\lceil m_1, m_2\rceil \cap \lceil m_1', m_2'\rceil = \varnothing$. Otherwise, S is (*time*) *nondeterministic*.

TFSM S is *complete* if each input is a defined at each state and for each pair $\langle s, i\rangle \in S \times I$ of S, it holds that the union of all $g \in G_{\langle s,i\rangle}$ equals $[0, \infty)$; otherwise, the machine is called *partial*. A partial machine can be completed by adding appropriate self-loop transitions. In particular, for every time domain g where an input i

[1] If there is no ambiguity we will use the notation S for an FSM and S for its set of states.

at state s is not defined, a self-loop transition $\langle s, \langle i, g\rangle, \langle o, [0, \infty)\rangle, s\rangle$ is added. Consequently, in this paper, we study distinguishing experiments with nondeterministic complete TFSMs.

Given TFSMs $S = \langle S, I, O, \lambda_S, \hat{s}\rangle$ and $P = \langle P, I, O, \lambda_P, \hat{p}\rangle$, the *intersection* $S \cap P$ is the largest connected submachine of the TFSM $\langle S \times P, I, O, \lambda_{S\cap P}, \langle \hat{s}, \hat{p}\rangle\rangle$ where $\langle\langle s, p\rangle, \langle i, \lceil m_1, m_2\rceil\rangle, \langle o, \lceil n_1, n_2\rceil\rangle, \langle s', p'\rangle\rangle \in \lambda_{S\cap P}$ if and only if there are transitions $\langle s, \langle i, \lceil m_1', m_2'\rceil\rangle, \langle o, \lceil n_1', n_2'\rceil\rangle, s'\rangle \in \lambda_S$ and $\langle p, \langle i, \lceil m_1'', m_2''\rceil\rangle, \langle o, \lceil n_1'', n_2''\rceil\rangle, p'\rangle \in \lambda_P$ such that $\lceil m_1', m_2'\rceil \cap \lceil m_1'', m_2''\rceil = \lceil m_1, m_2\rceil$ and $\lceil n_1', n_2'\rceil \cap \lceil n_1'', n_2''\rceil = \lceil n_1, n_2\rceil$. As a running example, consider TFSM S (Figure 1) with the initial state 1 (hereafter denoted S_1) and the TFSM S with the initial state 3 (hereafter denoted S_3). In the figures, a transition $\langle s, \langle i, \lceil m_1, m_2\rceil\rangle, \langle o, \lceil n_1, n_2\rceil\rangle, s'\rangle$ is depicted as s (column), i (row), and corresponding entry $(\lceil m_1, m_2\rceil), s'/\langle o, \lceil n_1, n_2\rceil\rangle$. The intersection $Q = S_1 \cap S_3$ is shown in Figure 2.

S	1	2	3	4
i_1	$(t \leqslant 2), 1/\langle o_1, t < 3\rangle$ $(t \leqslant 3), 2/\langle o_2, 0 \leqslant t < 5\rangle$ $(t > 2), 3/\langle o_1, 0 \leqslant t < 5\rangle$	$(t \leqslant 2), 1/\langle o_1, 0 \leqslant t < 5\rangle$ $(2 < t \leqslant 3), 2/\langle o_1, 0 \leqslant t < 5\rangle$ $(t > 3), 3/\langle o_1, 0 < t < 5\rangle$	$(t \leqslant 2), 3/\langle o_1, t > 2\rangle$ $(t > 3), 1/\langle o_1, 0 \leqslant t < 5\rangle$ $(2 < t \leqslant 3), 2/\langle o_1, t < 2\rangle$ $(2 < t \leqslant 3), 4/\langle o_2, 0 \leqslant t < 5\rangle$	$(t \leqslant 3), 3/\langle o_2, 0 \leqslant t < 5\rangle$ $(t > 3), 1/\langle o_1, 0 \leqslant t < 5\rangle$
i_2	$(t \leqslant 2), 1/\langle o_1, 0 \leqslant t < 5\rangle$ $(t > 2), 3/\langle o_1, 0 \leqslant t < 5\rangle$	$(t \leqslant 1), 1/\langle o_2, 0 \leqslant t < 5\rangle$ $(1 < t < 2), 2/\langle o_2, 0 \leqslant t < 5\rangle$ $(t \geqslant 2), 4/\langle o_2, 0 \leqslant t < 5\rangle$	$(t \leqslant 2), 3/\langle o_1, 0 \leqslant t < 5\rangle$ $(t > 2), 1/\langle o_1, 0 \leqslant t < 5\rangle$	$(t \leqslant 1), 3/\langle o_2, 0 \leqslant t < 5\rangle$ $(t > 1), 2/\langle o_2, 0 \leqslant t < 5\rangle$

Figure 1: TFSM S, TFSM S_1 is S with initial state 1, and TFSM S_3 is S with initial state 3

$S_1 \cap S_3$	$\langle 1, 3\rangle$	$\langle 3, 2\rangle$	$\langle 2, 4\rangle$	$\langle 2, 2\rangle$
i_1	$(t \leqslant 2), \langle 1, 3\rangle/\langle o_1, 2 < t < 3\rangle$ $(2 < t \leqslant 3), \langle 3, 2\rangle/\langle o_1, t < 2\rangle$ $(t > 3), \langle 3, 1\rangle/\langle o_1, 0 \leqslant t < 5\rangle$ $(2 < t \leqslant 3), \langle 2, 4\rangle/\langle o_2, 0 \leqslant t < 5\rangle$	$(t \leqslant 2), \langle 3, 1\rangle/\langle o_1, 0 < t < 5\rangle$ $(2 < t \leqslant 3), \langle 2, 2\rangle/\langle o_1, t < 2\rangle$ $(t > 3), \langle 1, 3\rangle/\langle o_1, 0 \leqslant t < 5\rangle$		$(t \leqslant 2), \langle 1, 1\rangle/\langle o_1, 0 \leqslant t < 5\rangle$ $(2 < t \leqslant 3), \langle 2, 2\rangle/\langle o_1, 0 \leqslant t < 5\rangle$ $(t > 3), \langle 3, 3\rangle/\langle o_1, 0 \leqslant t < 5\rangle$
i_2	$(t \leqslant 2), \langle 1, 3\rangle/\langle o_1, 0 \leqslant t < 5\rangle$ $(t > 2), \langle 3, 1\rangle/\langle o_1, 0 \leqslant t < 5\rangle$		$(t \leqslant 1), \langle 1, 3\rangle/\langle o_2, 0 \leqslant t < 5\rangle$ $(1 < t < 2), \langle 2, 2\rangle/\langle o_2, 0 \leqslant t < 5\rangle$ $(t \geqslant 2), \langle 4, 2\rangle/\langle o_2, 0 \leqslant t < 5\rangle$	$(t \leqslant 1), \langle 1, 1\rangle/\langle o_2, 0 \leqslant t < 5\rangle$ $(1 < t < 2), \langle 2, 2\rangle/\langle o_2, 0 \leqslant t < 5\rangle$ $(t \geqslant 2), \langle 4, 4\rangle/\langle o_2, 0 \leqslant t < \infty\rangle$

Figure 2: The intersection TFSM $Q = S_1 \cap S_3$

Given a TFSM S, a pair $\langle i, t\rangle/\langle o, t'\rangle$, where $i \in I$, $o \in O$, t and t' are non-negative rational numbers, is a *timed input-output* pair where $\langle i, t\rangle$ is a *timed input* that states that input i is applied at time t measured from the moment when the machine entered its current state and $\langle o, t'\rangle$ is a *timed output* that states that output o is produced at time t' measured from the moment when the timed input $\langle i, t\rangle$ has been applied.

Consider a TFSM S and a timed input-output pair $\langle i, t\rangle/\langle o, t'\rangle$. Given a state s, there is a *clocked transition* $\langle s, \langle i, t\rangle, \langle o, t'\rangle, s'\rangle$ in S if λ_S has a transition $\langle s, \langle i, g\rangle, \langle o, f\rangle, s'\rangle \in \lambda_S$ such that $t \in g$ and $t' \in f$. A timed input-output pair $\langle i, t\rangle/\langle o, t'\rangle$ is a timed input-output pair at state s if there exists a clocked transition $\langle s, \langle i, t\rangle, \langle o, t'\rangle, s'\rangle$ in S.

A sequence of timed input-output pairs is a *timed trace*. A *timed trace* $\alpha/\beta = \langle i_1, t_1\rangle/\langle o_1, t'_1\rangle, \ldots, \langle i_k, t_k\rangle/\langle o_k, t'_k\rangle$ is a *timed trace* at state s if there exist states $s_1, \ldots, s_{k+1}$ such that $s_1 = s$ and for each $j = 1, \ldots, k$, there exists a clocked transition $\langle s_j, \langle i_j, t_j\rangle, \langle o_j, t'_j\rangle, s_{j+1}\rangle$ in S.

By the above definition, given a timed trace $\alpha/\beta = \langle i_1, t_1\rangle/\langle o_1, t'_1\rangle, \ldots, \langle i_k, t_k\rangle/\langle o_k, t'_k\rangle$ at state s, we assume that the input sequence α is applied to the TFSM in the following way. For each j, $1 \leqslant j \leqslant k$, the input i_j is applied at the time instance t_j measured from the time when the TFSM entered the state s_j, the clock starts advancing from 0 and the output o_j is produced at time t'_j.

A timed input sequence α is *defined* at state s if and only if at state s there exists a timed trace α/β for some timed output sequence β.

A TFSM $S = \langle S, I, O, \lambda_S, \hat{s}\rangle$ is a *submachine* of TFSM $P = \langle P, I, O, \lambda_P, \hat{p}\rangle$ if $S \subseteq P$, $\hat{s} = \hat{p}$, and each clocked transition $\langle s, \langle i, t\rangle, \langle o, t'\rangle, s'\rangle$ of S is a clocked transition of P.

Two complete TFSMs S and P are *separable* if there exists a timed input sequence for both TFSMs such that the sets of timed output responses to this input sequence do not intersect and in addition, S and P are *r-distinguishable* if for each complete TFSM M it holds that there exists a timed input sequence α such that the set of output responses of M to α is not a subset of responses of S to α or of responses of P to α.

3 Distinguishing Timed Finite State Machines

Given two TFSMs S and P, in order to distinguish these machines, as usual, we first derive the TFSM intersection $Q = S \cap P$. Given the intersection Q, an abstract FSM $A(Q)$ is then constructed for which we can apply the traditional FSM distinguishability algorithms when deriving distinguishing sequences over abstract inputs; the distinguishing sequences are then transformed into timed sequences over timed inputs using the established correspondence between Q and $A(Q)$.

3.1 Deriving an FSM Abstraction

Given TFSM $Q = S \cap P$, an FSM abstraction $A(Q)$ of Q is derived as follows. For each input $i \in I$ of Q, the collection G_i of time guards over all states with an input i and the corresponding partition Π_i over $[0, \infty)$ is constructed. There is an input $\langle i, g\rangle$ in the abstraction if and only if $g \in \Pi_i$. More precisely, given input $i \in I$, let $G = \{j_1 = 0, j_2, \ldots, j_m\}$, $j_a < j_{a+1}$, $a = 1, \ldots, m-1$, be the finite ordered set of boundaries of guards of collection G_i. The finite set Π_i is defined as the (finite) set $\{(j_1, j_2), \ldots, (j_{m-1}, j_m), (j_m, \infty), \{j_1\}, \{j_2\}, \{j_3\}, \ldots \{j_m\}\}$, i.e., the set Π_i has singletons all boundaries and all (infinite) domains with consecutive boundaries of the set G. For each state $q \in Q$ and each $g_j \in \Pi_i$, the abstraction $A(Q)$ has a transition from state q under abstract input $\langle i, g_j\rangle$ if and only if it holds that there exists a transition $\langle q, \langle i, g\rangle, \langle o, f\rangle, q'\rangle \in \lambda_Q$ such that g contains g_j. For our running

example, Π_{i_1} of TFSM Q in Figure 2 equals $\{\{0\},(0,2),\{2\},(2,3),\{3\},(3,\infty)\}$ and $\Pi_{i_2} = \{\{0\},(0,1),\{1\},(1,2),\{2\},(2,\infty)\}$.

Proposition 1. *Given a TFSM $Q = \langle Q, I, O, \lambda_Q, \hat{q}\rangle$, an input $i \in I$ and a set Π_i of time domains for the input i, let $g \in \Pi_i$ and $t_1, t_2 \in g$. For each $q \in Q$, there is a clocked transition $\langle q, \langle i, t_1\rangle, \langle o, f\rangle, q'\rangle \in \lambda_Q$ if and only if there is a clocked transition $\langle q, \langle i, t_2\rangle, \langle o, f\rangle, q'\rangle \in \lambda_Q$.*

Similarly, the partition Π_o of output guards is derived. For each output $o \in O$ of Q, the collection F_o based on the collections $F_{\langle q,o\rangle}$ over all states where the output o can be produced is derived. An output *o can be produced at time instances $t \in f$* if and only if there exists a state q and pair $\langle i, g\rangle$ such that $\langle q, \langle i, g\rangle, \langle o, f\rangle, q'\rangle \in$ $\in \lambda_Q$. Let now $F = \{j_1 = 0, j_2, \ldots, j_m\}$, $j_a < j_{a+1}$, $a = 1, \ldots, m-1$, be the finite ordered set of boundaries of guards of the collection F_o. Based on F the (finite) set $\Pi_o = \{(j_1, j_2), \ldots, (j_{m-1}, j_m), (j_m, \mathbf{B}), \{j_1\}, \{j_2\}, \{j_3\}, \ldots, \{j_m\}\}$ is built, i.e., the set Π_o has singletons for all boundaries and all (infinite) domains with consecutive boundaries of the set F where the output o can be produced. In our running example, Π_{o_1} of TFSM Q (Figure 2) equals $\{\{0\},(0,2),\{2\},(2,3),\{3\},(3,5)\}$ and $\Pi_{o_2} = \{\{0\},(0,5)\}$.

Proposition 2. *Given a TFSM $Q = \langle Q, I, O, \lambda_Q, \hat{q}\rangle$, an output $o \in O$ and a set Π_o of output domains for the output o, let $f \in \Pi_o$ and $t', t'' \in f$. For each $q \in Q$ and a timed input $\langle i, t\rangle$, either TFSM Q cannot produce both timed outputs $\langle o, t'\rangle$ and $\langle o, t''\rangle$ at state q under $\langle i, t\rangle$ or there is a clocked transition $\langle q, \langle i, t\rangle, \langle o, t'\rangle, q'\rangle \in \lambda_Q$ if and only if there is a clocked transition $\langle q, \langle i, t\rangle, \langle o, t''\rangle, q'\rangle \in \lambda_Q$.*

Given TFSMs S and P, the TFSM intersection $Q = \langle Q, I, O, \lambda_Q, \hat{q}\rangle$ of S and P, and partitions Π_i and Π_o, a corresponding abstract FSM $A(Q) =$ $= \langle Q, I_{A(Q)}, O_{A(Q)}, \lambda_A, \hat{q}\rangle$ of the intersection can be derived as follows. The FSM $A(Q)$ has the same set of states and the same initial state as Q, and $A(Q)$ has (abstract) inputs $I_{A(Q)} = \{\langle i, g\rangle : i \in I, g \in \Pi_i\}$, (abstract) outputs $O_{A(Q)} =$ $= \{\langle o, f\rangle : o \in O, f \in \Pi_o\}$ and transition relation λ_A; there is a transition $\langle q, \langle i, g\rangle, \langle o, f\rangle, q'\rangle$ in λ_A if and only if there is a transition $\langle q, \langle i, g'\rangle, \langle o, f'\rangle, q'\rangle \in \lambda_Q$ such that $g \subseteq g'$ and $f \subseteq f'$. Considering the running example, abstract inputs of $A(Q)$ are the pairs from $\{i_1\} \times \Pi_{i_1}$ and $\{i_2\} \times \Pi_{i_2}$ and abstract outputs are the pairs from $\{o_1\} \times \Pi_{o_1}$ and $\{o_2\} \times \Pi_{o_2}$. A fragment of $A(Q)$ for the TFSM Q in Figure 2 is shown in Figure 3.

Based on the above construction, the following statements can be established.

Proposition 3. *The following statements hold.*

1. *(a) If TFSMs S and P are observable then TFSM $Q = S \cap P$ is observable. (b) TFSM Q is observable if and only if FSM $A(Q)$ is observable.*

2. *Given a state q of TFSM Q, a timed input-output pair $\langle i, t\rangle/\langle o, t'\rangle$ is defined at state q if and only if there exists a transition $\langle q, \langle i, g\rangle, \langle o, f\rangle, q'\rangle$ in the abstract FSM such that $t \in g$ and $t' \in f$. Moreover, given a defined (abstract)*

input-output pair $\langle i, g\rangle/\langle o, f\rangle$ *at state* q *of the FSM* $A(Q)$, $t_1, t_2 \in g$, $t'_1, t'_2 \in f$, *there is a clocked transition* $\langle q, \langle i, t_1\rangle, \langle o, t'_1\rangle, q'\rangle \in \lambda_Q$ *if and only if there is a clocked transition* $\langle q, \langle i, t_2\rangle, \langle o, t'_2\rangle, q'\rangle \in \lambda_Q$.

3. *Given an abstract input-output sequence* $\langle i_1, g_1\rangle/\langle o_1, f_1\rangle \ldots \langle i_k, g_k\rangle/\langle o_k, f_k\rangle$ *at state* q *of the FSM* $A(Q)$, *each timed input-output sequence* $\langle i_1, t_1\rangle/\langle o_1, t'_1\rangle \ldots \langle i_k, t_k\rangle/\langle o_k, t'_k\rangle$ *such that* $t_j \in g_j$, $t'_j \in f_j$, $j = 1, \ldots, k$, *is a timed input-output sequence at state* q *of TFSM* Q, *and vice versa, given a timed trace* $\langle i_1, t_1\rangle/\langle o_1, t'_1\rangle \ldots \langle i_k, t_k\rangle/\langle o_k, t'_k\rangle$ *at state* q *of TFSM* Q *there always exists a defined input sequence* $\langle i_1, g_1\rangle/\langle o_1, f_1\rangle \ldots \langle i_k, g_k\rangle/\langle o_k, f_k\rangle$ *at state* q *of the FSM* $A(Q)$ *such that* $t_j \in g_j$, $t'_j \in f_j$, $j = 1, \ldots, k$.

4. *TFSM* Q *has a timed trace* $\langle i_1, t_1\rangle/\langle o_1, t'_1\rangle \ldots \langle i_k, t_k\rangle/\langle o_k, t'_k\rangle$ *at state* q *if and only if the FSM* $A(Q)$ *has a trace* $\langle i_1, g_1\rangle/\langle o_1, f_1\rangle \ldots \langle i_k, g_k\rangle/\langle o_k, f_k\rangle$ *such that* $t_j \in g_j$, $t'_j \in f_j$, $j = 1, \ldots, k$, *at state* s.

Proof. 1. (a) If TFSMs S and P are observable, then for every two timed transitions $\langle s, \langle i, t\rangle, \langle o, t'\rangle, s'\rangle \in \lambda_S$, $\langle s, \langle i, t\rangle, \langle o, t'\rangle, s''\rangle \in \lambda_S$ (or $\langle p, \langle i, t\rangle, \langle o, t'\rangle, p'\rangle \in \lambda_P$, $\langle p, \langle i, t\rangle, \langle o, t'\rangle, p''\rangle \in \lambda_P$) it holds that $s' = s''$ (or correspondingly $p' = p''$). Thus, there are no timed transitions $\langle\langle s, p\rangle, \langle i, t\rangle, \langle o, t'\rangle, \langle s', p'\rangle\rangle \in \lambda_Q$ and $\langle\langle s, p\rangle, \langle i, t\rangle, \langle o, t'\rangle, \langle s'', p''\rangle\rangle \in \lambda_Q$ such that $\langle s', p'\rangle \neq \langle s'', p''\rangle$.

(b) TFSM Q is observable if and only if for every two timed transitions $\langle q, \langle i, t\rangle, \langle o, t'\rangle, q'\rangle \in \lambda_Q$ and $\langle q, \langle i, t\rangle, \langle o, t'\rangle, q''\rangle \in \lambda_Q$ it holds that $q' = q''$. Correspondingly, by construction of the FSM $A(Q)$, for each defined input $\langle i, g\rangle$ at state q of the FSM $A(Q)$ it holds that there are no two transitions $\langle q, \langle i, g\rangle, \langle o, f\rangle, q'\rangle \in \lambda_A$ and $\langle q, \langle i, g'\rangle, \langle o, f'\rangle, q''\rangle \in \lambda_A$ such that $g \cap g' \neq \varnothing$, $f \cap f' \neq \varnothing$ while $q' \neq q''$, i.e., FSM $A(Q)$ is observable if and only if TFSM Q is observable.

2. Statement 2 of the above proposition is a direct corollary to the definition of time domains.

3. Statement 3 can be shown by induction on the length of a defined input sequence.

4. Statement 4 is implied by the definition of the FSM $A(Q)$ and Statement 3. □

We recall that an abstract FSM $A(Q)$ and TFSM Q have the same number of states, while, $A(Q)$ has more transitions as it has more inputs. However, the number of transitions of an $A(Q)$ is polynomial w.r.t. the number of transitions of Q as it mainly depends on the number of (abstract) inputs $I_{A(Q)}$ which is of order $|I| \cdot m$ where m is the maximum number of items of partitions Π_i.

3.2 Deriving an r-distinguishing TFSM

In order to check whether nondeterministic machines S and P can be distinguished by an adaptive experiment a so-called r-distinguishing machine can be used. The derivation of such a machine is described in [5,16] for complete untimed FSMs and in [6] for complete TFSMs S and P without output delays. In this paper, such a machine is derived based on the abstraction $A(Q)$ for TFSMs S and P with output delays.

Similar to FSMs [5,16,17], an adaptive experiment is represented by a special acyclic so-called single-input output-complete TFSM. Given complete observable TFSMs $S = \langle S, I, O, \lambda_S, \hat{s}\rangle$ and $P = \langle P, I, O, \lambda_P, \hat{p}\rangle$, let $R = \langle R, I, O, \lambda_R, \hat{r}\rangle$ be an acyclic initially connected TFSM such that the set R of states has two designated deadlock states called r_S and r_P. If after the experiment the machine R reaches state r_S then the TFSM under experiment is S while if the final state is r_P then the TFSM under experiment is P. Only one timed input $\langle i, t\rangle$ is defined at each other state of R with all possible outputs, i.e., TFSM R represents an adaptive experiment with a TFSM over input alphabet I and output alphabet O. TFSM R is an r-distinguishing TFSM $R_{(S,P)}$ of S and P (or TFSM $R_{(S,P)}$ r-distinguishes TFSM S and P) if for each state $\langle s, r\rangle$ of the intersection $S \cap R_{(S,P)}$ it holds that $r \neq r_P$ and for each $\langle p, r\rangle$ of the intersection $P \cap R_{(S,P)}$ it holds that $r \neq r_S$.

Similar to FSMs [16], here, we define the notion of *k-undefined* states in order to derive $R(S, P)$ using $A(Q)$. Given (complete observable) TFSMs S and P, $Q = S \cap P$, and FSM abstraction $A(Q)$, state $q = \langle s, p\rangle$ of $A(Q)$ is 1-undefined if there exists an undefined (abstract) input $\langle i, g\rangle$ at state q. Consider $k > 1$ and assume that all $(k-1)$-undefined states of $A(Q)$ are determined. State $q = \langle s, p\rangle$ of $A(Q)$ is k-undefined if q is $(k-1)$-undefined or there exists an abstract input $\langle i, g\rangle$ defined at state q such that for each transition $\langle q, \langle i, g\rangle, \langle o, f\rangle, q'\rangle \in \lambda_A$, each state q' is $(k-1)$-undefined. It can be shown as in [16], that given complete observable TFSMs S and P, these TMSMs are r-distinguishable iff there exists an integer k such that the initial state of the abstraction $A(Q)$ is k-undefined for some $k > 0$.

We use Algorithm 1 in order to derive an r-distinguishing TFSM for two given TFSMs S and P based on the abstract FSM $A(Q)$ of $Q = S \cap P$. If an r-distinguishing FSM over abstract inputs of $A(Q)$ is derived, then the machine is converted to corresponding timed inputs in order to represent an r-distinguishing TFSM for TFSMs S and P.

Based on the TFSM $R_{(S,P)}$ an adaptive experiment for distinguishing TFSMs S and P can be performed in the following way. Given TFSM under test, which is either TFSM S or P, the experiment starts at the initial state $\hat{r} = \hat{q}$ of TFSM $R_{(S,P)}$. At any state of $R_{(S,P)}$ only one timed input $\langle i, t\rangle$ is defined, in addition, any state of $R_{(S,P)}$ is always reached at time $t = 0$. Thus, when reaching a current state r of $R_{(S,P)}$ the clock advances from 0 and the only defined input $\langle i, t\rangle$ is applied to a TFSM under test. In response, the TFSM under test produces a timed output $\langle o, t'\rangle$, $t' \in f$, and accordingly the TFSM $R_{(S,P)}$ moves from a current state r to the next state r' according to the clocked transition $\langle r, \langle i, [t, t]\rangle, \langle o, f\rangle, r'\rangle$. The procedure terminates when the TFSM $R_{(S,P)}$ reaches one of the deadlock states r_S

Algorithm 1 Deriving an r-distinguishing TFSM of two TFSMs

Input: Complete observable TFSMs $S = \langle S, I, O, \lambda_S, \hat{s}\rangle$ and $P = \langle P, I, O, \lambda_P, \hat{p}\rangle$
Output: A distinguishing TFSM $R_{(S,P)}$ if TFSMs S and P are r-distinguishable

1: $Q := S \cap P$;
2: derive the FSM abstraction $A(Q)$;
3: $R := \langle R, I, O, \lambda_R\rangle$, where initially λ_R is empty and R contains two deadlock states r_S and r_P;
4: $k := 1$;
5: $Q_k := Q$; *//Q is the set of states of TFSM Q which are pairs of states of S and P*
6: **while** ($\hat{q} \in Q_k$ **and** the set Q_k has k-undefined states) **do**
7: determine all states of the set Q_k which are k-undefined in $A(Q)$;
8: **for all** k-undefined states $q = \langle s, p\rangle$ of the set Q_k **do**
9: **if** (k == 1) **then**
10: determine an abstract input $\langle i, g\rangle$ such that it is undefined at state q;
11: **else**
12: determine an abstract input $\langle i, g\rangle$ such that for each transition $\langle q, \langle i, g\rangle, \langle o, f\rangle, q'\rangle \in \lambda_Q$, state q' is $(k-1)$-undefined;
13: **end if**
14: add state q into the set R;
15: **for all** abstract outputs $\langle o, f\rangle$ **do**
16: **if** there is a transition $\langle q, \langle i, g\rangle, o, f, q'\rangle \in \lambda_A$ **then** *//implies that* $k > 1$
17: add to λ_R the tuple $\langle (q, \langle i, [t, t]\rangle, \langle o, f\rangle, q'\rangle$, $t \in g$;
18: **else**
19: add to λ_R the tuple $\langle q, \langle i, [t, t]\rangle, \langle o, f\rangle, r_S\rangle$ if for each $t \in g$ the output o can be produced by S for time instances $t' \in f$;
20: add to λ_R the tuple $\langle q, \langle i, [t, t]\rangle, \langle o, f\rangle, r_P\rangle$ if for each $t \in g$ the output o can be produced by P for time instances $t' \in f$;
21: **end if**
22: **end for**
23: delete state q from the set Q_k;
24: **end for**
25: $k := k + 1$; $Q_k := Q_{k-1}$;
26: **end while**
27: **if** $\hat{q} \notin Q_k$ **then**
28: convert the tuple $R = \langle R, I, O, \lambda_R\rangle$ into a TFSM R by claiming state $\hat{q}$ as the initial state of the TFSM and augment R (if it is necessary) to an output-complete TFSM by adding transitions to deadlock states;
29: **return** the largest initially connected submachine of TFSM R as the TFSM $R_{(S,P)}$;
30: **else**
31: **return** TFSMs S and P are not r-distinguishable.
32: **end if**

or r_P. Correspondingly, if state r_S (r_P) of $R_{(S,P)}$ is reached then the TFSM under test is S (P).

Similar to [6], it can be shown that each trace of a TFSM $R_{(S,P)}$ obtained in the above algorithm is of order $|S| \cdot |P|$ where S and P are the sets of states of TFSMs S and P, respectively and only one trace of $R_{(S,P)}$ is used when performing the experiment. In this paper, as for other distinguishing experiments, the complexity of an adaptive experiment is measured using the height of the experiment, i.e., the length of a longest trace to a deadlock state in the (acyclic) TFSM $R_{(S,P)}$. As TFSM $R_{(S,P)}$ has at most $|S| \cdot |P|$ states, this length, and thus, the complexity of an adaptive experiment, is at most $|S| \cdot |P|$ and this upper bound is reachable as this upper bound is reachable for two untimed FSMs [22].

Example 1. Consider the running example and TFSMs S_1 and S_3 with the initial states 1 and 3, respectively. We add into R two deadlock states r_{S_1} and r_{S_3} with subscripts indicating the initial states of the machines. The intersection $Q = S_1 \cap S_3$ is shown in Figure 2. The FSM abstraction $A(Q)$ is constructed from Q by having the same states and splitting every transition of Q using the abstract inputs and outputs given above. A fragment of $A(Q)$ for states $\langle 1,3\rangle$ and $\langle 3,2\rangle$ under the input i_1 of the intersection Q is shown in Figure 3. In particular, Figure 3 includes the transitions at states $\langle 1,3\rangle$ and $\langle 3,2\rangle$ under i_1 of Q (in Figure 2) and their corresponding transitions in $A(Q)$ derived using the partitions Π_{i_1}, Π_{o_1} and Π_{o_2} given above. By applying Algorithm 1, initially, $k = 1$, the set $Q_1 = Q$ includes all

$A(Q)$	$\langle 1,3\rangle$	$\langle 3,2\rangle$
i_1	$(t=0), \langle 1,3\rangle/\langle o_1, 2<t<3\rangle$; $(0<t<2), \langle 1,3\rangle/\langle o_1, 2<t<3\rangle$ $(t=2), \langle 1,3\rangle/\langle o_1, 2<t<3\rangle$; $(2<t<3), \langle 3,2\rangle/\langle o_1, t=2\rangle$ $(2<t<3), \langle 3,2\rangle/\langle o_1, 0<t<2\rangle$; $(t=3), \langle 3,2\rangle/\langle o_1, t=0\rangle$ $(t=3), \langle 3,2\rangle/\langle o_1, 0<t<2\rangle$; $(t>3), \langle 3,1\rangle/\langle o_1, t=0\rangle$ $(t>3), \langle 3,1\rangle/\langle o_1, 0<t<2\rangle$; $(t>3), \langle 3,1\rangle/\langle o_1, t=2\rangle$ $(t>3), \langle 3,1\rangle/\langle o_1, 2<t<3\rangle$; $(t>3), \langle 3,1\rangle/\langle o_1, t=3\rangle$ $(t>3), \langle 3,1\rangle/\langle o_1, 3<t<5\rangle$; $(2<t<3), \langle 2,4\rangle/\langle 0_2, t=0\rangle$ $(2<t<3), \langle 2,4\rangle/\langle 0_2, 0<t<5\rangle$; $(t=3), \langle 2,4\rangle/\langle o_2, t=0\rangle$ $(t=3), \langle 2,4\rangle/\langle 0_2, 0<t<5\rangle$	$(t=0), \langle 3,1\rangle/\langle o_1, 2<t<3\rangle$; $(t=0), \langle 3,1\rangle/\langle o_1, t=3\rangle$ $(t=0), \langle 3,1\rangle/\langle o_1, 3<t<5\rangle$; $(0<t<1), \langle 3,1\rangle/\langle o_1, 2<t<3\rangle$ $(0<t<1), \langle 3,1\rangle/\langle o_1, t=3\rangle$; $(0<t<1), \langle 3,1\rangle/\langle o_1, 3<t<5\rangle$ $(t=2), \langle 3,1\rangle/\langle o_1, 2<t<3\rangle$; $(t=2), \langle 3,1\rangle/\langle o_1, t=3\rangle$ $(t=2), \langle 3,1\rangle/\langle o_1, 3<t<5\rangle$; $(2<t<3), \langle 2,2\rangle/\langle o_1, t=0\rangle$ $(2<t<3), \langle 2,2\rangle/\langle o_1, 0<t<2\rangle$; $(t=3), \langle 2,2\rangle/\langle o_1, t=0\rangle$ $(t=3), \langle 2,2\rangle/\langle o_1, 0<t<2\rangle$; $(t>3), \langle 1,3\rangle/\langle o_1, t=0\rangle$ $(t>3), \langle 1,3\rangle/\langle o_1, 0<t<2\rangle$; $(t>3), \langle 1,3\rangle/\langle o_1, t=2\rangle$ $(t>3), \langle 1,3\rangle/\langle o_1, 2<t<3\rangle$; $(t>3), \langle 1,3\rangle/\langle o_1, t=3\rangle$ $(t>3), \langle 1,3\rangle/\langle o_1, 3<t<5\rangle$

Figure 3: Fragment of the abstract FSM $A(Q)$

states of TFSM Q with the initial state $\langle 1,3\rangle$. States 3 and 2 of state $\langle 3,2\rangle$ in Q_1 are 1-r-distinguishable by abstract input $\langle i_2, 1\rangle$ and states 2 and 4 of state $\langle 2,4\rangle$ in Q_1 are 1-r-distinguishable by $\langle i_1, 2\rangle$. Thus, we add states $\langle 3,2\rangle$ and $\langle 2,4\rangle$ into the set R, that initially contains only deadlock states r_{S_1} and r_{S_3}, remove these states from Q_1, obtain Q_2 as $Q_1 \setminus \{\langle 3,2\rangle, \langle 2,4\rangle\}$, and add into (initially empty) λ_R the tuples

$$\langle\langle 3,2\rangle, \langle i_2, [1,1]\rangle, \langle o_1, [0,0]\rangle, r_{S_1}\rangle,$$
$$\langle\langle 3,2\rangle, \langle i_2, [1,1]\rangle, \langle o_1, (0,2)\rangle, r_{S_1}\rangle,$$
$$\langle\langle 3,2\rangle, \langle i_2, [1,1]\rangle, \langle o_1, [2,2]\rangle, r_{S_1}\rangle,$$
$$\langle\langle 3,2\rangle, \langle i_2, [1,1]\rangle, \langle o_1, (2,3)\rangle, r_{S_1}\rangle,$$
$$\langle\langle 3,2\rangle, \langle i_2, [1,1]\rangle, \langle o_1, [3,3]\rangle, r_{S_1}\rangle,$$
$$\langle\langle 3,2\rangle, \langle i_2, [1,1]\rangle, \langle o_1, (3,5)\rangle, r_{S_1}\rangle,$$

and add the tuples

$$\langle\langle 2,4\rangle, \langle i_2, [2,2]\rangle, \langle o_1, [0,0]\rangle, r_{S_1}\rangle,$$
$$\langle\langle 2,4\rangle, \langle i_2, [2,2]\rangle, \langle o_1, (0,2)\rangle, r_{S_1}\rangle,$$
$$\langle\langle 2,4\rangle, \langle i_2, [2,2]\rangle, \langle o_1, [2,2]\rangle, r_{S_1}\rangle,$$
$$\langle\langle 2,4\rangle, \langle i_2, [2,2]\rangle, \langle o_1, (2,3)\rangle, r_{S_1}\rangle,$$
$$\langle\langle 2,4\rangle, \langle i_2, [2,2]\rangle, \langle o_1, [3,3]\rangle, r_{S_1}\rangle,$$
$$\langle\langle 2,4\rangle, \langle i_2, [2,2]\rangle, \langle o_1, (3,5)\rangle, r_{S_1}\rangle,$$
$$\langle\langle 2,4\rangle, \langle i_2, [2,2]\rangle, \langle o_2, [0,0]\rangle, r_{S_3}\rangle,$$
$$\langle\langle 2,4\rangle, \langle i_2, [2,2]\rangle, \langle o_2, (0,5)\rangle, r_{S_3}\rangle.$$

Afterwards, in a second iteration of the loop, we observe that states 1 and 3 of state $\langle 1,3\rangle$ in Q_2 are 2-r-distinguishable. In fact, the abstract input $\langle i_1, 3\rangle$ when applied at state $\langle 1,3\rangle$ of $A(Q)$ reaches only states $\langle 3,2\rangle$ and $\langle 2,4\rangle$ which are both 1-undefined. Thus, we add state $\langle 1,3\rangle$ into R, add into λ_R the tuples $\langle\langle 1,3\rangle, \langle i_1, [3,3]\rangle, \langle o_1, [0,0]\rangle, \langle 2,4\rangle\rangle$, $\langle\langle 1,3\rangle, \langle i_1, [3,3]\rangle, \langle o_1, (0,2)\rangle, \langle 3,2\rangle\rangle$, and add the tuples, $\langle\langle 1,3\rangle, \langle i_1, [3,3]\rangle, \langle o_2, [0,0]\rangle, \langle 2,4\rangle\rangle$, $\langle\langle 1,3\rangle, \langle i_1, [3,3]\rangle, \langle o_2, (0,5)\rangle, \langle 3,2\rangle\rangle$. Afterwards by deleting $\langle 1,3\rangle$, which is the initial state of $A(Q)$, from Q_2 we stop. Convert the tuple R into TFSM $R_{(S_1,S_3)}$ with initial state $\langle 1,3\rangle$ and obtain a partial TFSM as shown in Figure 4.

$R_{(S_1,S_3)}$	$\langle 1,3\rangle$	$\langle 3,2\rangle$	$\langle 2,4\rangle$	r_{S_1}	r_{S_3}
$\langle i_1, [3,3]\rangle$	$\langle 3,2\rangle/\langle o_1, [0,0]\rangle$ $\langle 3,2\rangle/\langle o_1, 0 < t < 2\rangle$ $\langle 2,4\rangle/\langle o_2, [0,0]\rangle$ $\langle 2,4\rangle/\langle o_2, 0 < t < 5\rangle$				
$\langle i_1, [2,2]\rangle$			$r_{S_1}/\langle o_1, [0,0]\rangle$; $r_{S_1}/\langle o_1, 0 < t < 2\rangle$ $r_{S_1}/\langle o_1, [2,2]\rangle$; $r_{S_1}/\langle o_1, 2 < t < 3\rangle$ $r_{S_1}/\langle o_1, [3,3]\rangle$; $r_{S_1}/\langle o_1, 3 < t < 5\rangle$ $r_{S_3}/\langle o_2, [0,0]\rangle$; $r_{S_3}/\langle o_2, 0 < t < 5\rangle$		
$\langle i_2, [1]\rangle$		$r_{S_1}/\langle o_1, [0,0]\rangle$; $r_{S_1}/\langle o_1, 0 < t < 2\rangle$ $r_{S_1}/\langle o_1, [2,2]\rangle$; $r_{S_1}/\langle o_1, 2 < t < 3\rangle$ $r_{S_1}/\langle o_1, [3,3]\rangle$; $r_{S_1}/\langle o_1, 3 < t < 5\rangle$ $r_{S_3}/\langle o_2, [0,0]\rangle$; $r_{S_3}/\langle o_2, 0 < t < 5\rangle$			

Figure 4: A part of the TFSM $R_{(S_1,S_3)}$

3.3 Deriving a Separating Sequence

In order to derive a separating sequence for two given TFSMs S and P, in the following, we adapt the algorithm given in [19] to deal with the abstract FSM $A(Q)$ of $Q = S \cap P$. Correspondingly, a separating sequence (if exists) will be derived for TFSMs S and P with output delays. If a separating sequence over abstract inputs $\langle i, g\rangle$ is derived from $A(Q)$, then the sequence is replaced by a corresponding timed sequence, over timed inputs $\langle i, t\rangle$, $t \in g$, that is a separating sequence for TFSMs S and P.

Here we define the following notion used in Algorithm 2. Given state s of an FSM $S = \langle S, I, O, \lambda_S, \hat{s}\rangle$, state s' is an i-successor of state s if there exists is a

Algorithm 2 Deriving a Separating Sequence of Two TFSMs

Input: Complete observable TFSMs $S = \langle S, I, O, \lambda_S, \hat{s}\rangle$ and $P = \langle P, I, O, \lambda_P, \hat{p}\rangle$
Output: A (shortest) separating sequence of TFSMs $S = \langle S, I, O, \lambda_S, \hat{s}\rangle$ and $P = = \langle P, I, O, \lambda_P, \hat{p}\rangle$ (if such a sequence exists)

1: derive the intersection $Q = S \cap P$;
2: **if** Q is a complete TFSM **then**
3: the TFSMs $S = \langle S, I, O, \lambda_S, \hat{s}\rangle$ and $P = \langle P, I, O, \lambda_P, \hat{p}\rangle$ are non-separable;
4: **end** Algorithm 2;
5: **end if**
6: derive from $Q = S \cap P$ (with input and output partitions Π_i and Π_o), the abstract FSM $A(Q)$ with abstract inputs and outputs $\{\langle i, g\rangle : i \in I, g \in \Pi_i\}$ and $\{\langle o, f\rangle : o \in O, f \in \Pi_o\}$;
7: derive a truncated successor tree of the FSM $A(Q)$. The root of this tree, which is at the 0^{th} level, is the initial state $\langle \hat{s}, \hat{p}\rangle$ of $A(Q)$; the nodes of the tree are labeled with subsets of states of $A(Q)$. Given already derived j tree levels, $j \geqslant 0$, a non-leaf (intermediate) node of the j^{th} level labeled with a subset C of states of $A(Q)$ and a abstract input $\langle i, g\rangle$, there is an outgoing edge from this non-leaf node labeled with $\langle i, g\rangle$ to the node with the subset of the $\langle i, g\rangle$-successors of states of the subset C. A current node *Current*, at the k^{th} level, $k \geqslant 0$, labeled with the subset C of states, is claimed as a leaf node if one of the following conditions holds:
8: **Rule 1:** There exists an input $\langle i, g\rangle$ such that each state $\langle s, p\rangle$ of the set C has no $\langle i, g\rangle$-successors in $A(Q)$;
9: **Rule 2:** There exists a node at the j^{th} level, $j < k$, labeled with a subset R of states with the property $R \subseteq C$;
10: **if** none of the paths of the truncated tree derived at Step 7 is terminated using **Rule 1 then**
11: the TFSMs $S = \langle S, I, O, \lambda_S, \hat{s}\rangle$ and $P = \langle P, I, O, \lambda_P, \hat{p}\rangle$ are non-separable;
12: **end** Algorithm 2;
13: **end if**
14: **if** there is a leaf node, *Leaf*, labeled with the subset C of states such that for some (abstract) input $\langle i, g\rangle$, each state of the set C has no $\langle i, g\rangle$-successors **then**
15: select such a path with minimal length, append an input sequence that labels the path with input $\langle i, g\rangle$ and transform the obtained input sequence replacing each abstract input of the sequence $\langle i, h\rangle$ by a timed input $\langle i, t\rangle$, $t \in h$;
16: the obtained timed input sequence is a shortest separating sequence of TFSMs S and P;
17: **end if**

transition $\langle s, i, o, s'\rangle$ in λ_S. Generally, for a nondeterministic FSM, the set of i-successors of state s can have several states. Given a set of states $M \subseteq S$ of the

complete FSM S, and an input i, the set M' of states is an i-successor of the set M if M' is the union of the sets of i-successors over all states of the set M.

Similar to [19] it can be shown that Algorithm 2 returns a separating sequence α if and only if the TFSMs S and P are separable. The separating sequence α can be applied to a TFSM under experiment (S or P) and since the sets of output responses of TFSMs S and P do not intersect, after getting the output response to α the conclusion can be drawn which TFSM is under the experiment. In addition, it can be shown that the complexity (length of a separating sequence) is exponential w.r.t. to the number of states of TFSMs S and P as it happens for untimed FSMs [19]. The length of a separating sequence of two FSMs with n and m states is at most 2^{mn-1} [19] and this upper bound is reachable, and thus, it is reachable for TFSMs as well.

The above algorithm is based on deriving a successor tree using an (FSM) abstraction $A(Q)$ of the intersection $Q = S \cap P$. As $A(Q)$ can have more inputs than Q, we compare the above approach with another approach where a successor tree can be derived using Q instead [6]. In both approaches, in the worst case, each path p from the root node to a leaf node has to be traversed and a number o of elementary operations (**Rule 1** and **Rule 2**) have to be applied at each node of a path. Let l be the maximum length of a path, then the complexity of the algorithm equals the product $p \cdot l \cdot o$. The maximal length l is the same for the two approaches and l is of the order $O(2^{mn})$ for TFSMs S and P with m and n states, respectively [19]. Further, in both approaches, **Rule 1** and **Rule 2** of the above algorithm have to be checked at each node of the derived successor tree where a node is labeled with the set C of states of a corresponding TFSM Q or of the abstraction FSM $A(Q)$. Checking these rules using $Q = S \cap P$ is more complex since at each node for each input i and each subset Q_{kj} of states at the node we have to derive the set Π as the intersection of $\Pi\langle q, i\rangle$ over all states $q \in Q_{kj}$ while in the approach based on $A(Q)$, the intersection is calculated only once when deriving $A(Q)$. As the number of guards we need to intersect is proportional to the product of the finite upper bound of guards for input i and the number of states of the set Q_{kj}, in the approach based on $Q = S \cap P$, the number of calculations which have to be performed for deriving the intersection of guards at each node polynomially grows compared with the approach based on $A(Q)$. On the other hand, the number of inputs of $A(Q)$ can be larger than that of Q. If $\mathbf{B}$ is the maximum finite bound for a given input i over all states then for each i, the number of (abstract) inputs of $A(Q)$ can be $2 \cdot \mathbf{B}$ times bigger than that of Q, since in $A(Q)$ time domains for an i are derived based on the corresponding guards for all states of $A(Q)$. As the number p of paths of the successor tree exponentially depends on the number of inputs considered at each tree node, this implies that the complexity of the approach based on $A(Q)$ will exponentially grow compared to the approach based on Q, since p is of the order $O(|I|^l)$ where $|I|$ is the number of inputs of Q or $A(Q)$, respectively. This difference between the two approaches can be bypassed by considering for each input i only guards corresponding to a given state of Q when deriving the abstraction $A(Q)$, i.e., not taken into account guards under this input over other states of Q. In this case, it can well happen that $A(Q)$ is partially specified. The above algorithm can

be adapted to partial FSM $A(Q)$; however, this is not done in this paper in order to simplify the presentation of the algorithms and to avoid presenting more complex FSM related definitions that consider defined and undefined input sequences at states. If partially specified FSM $A(Q)$ is used, the number p will be the same for both approaches. Generally, the approach based on the partial FSM abstraction of the intersection performs less computations than the approach based on the intersection Q instead. However, the best way to assess any abstraction method is thorough experimental evaluation with large size specifications and this could be the topic of another paper. It is worth mentioning that though the length of a separating sequence can reach length 2^{mn-1} (for TFSMs S and P with m and n states) [19]; nevertheless, experiments with various size FSM specifications show that this length usually does not exceed mn [18].

As $A(Q)$ can have more inputs than Q, here we also compare the approach given in this paper (Algorithm 1) based on using $A(Q)$ with another approach [6] based on using Q instead for deriving an adaptive distinguishing sequence (represented as a distinguishing machine). For both approaches, in the worst-case, the maximum length l of a path from the initial state of the constructed FSM $R_{(S,P)}$ to the deadlock state r_S or r_P is the same and is of the order $O(mn)$ for TFSMs S and P with m and n states, respectively [5]. In addition, as both approaches are based on deriving a submachine of a $A(Q)$ or of Q, the number of paths p included as transitions in the tuples of λ_R in both approaches is the same, and p is of the order $O(2^{mn})$ [22]. Moreover, in the approach that is based on the intersection Q, in the worst case, for a given input, we have to consider all possible time domains $\langle i, g\rangle$, $g \in \Pi$, over all states $q \in Q_k$. As the number of guards we need to intersect when deriving the set Π is proportional to the product of the finite upper bound of guards for input i and the number of states of the set Q_k, the number of calculations which have to be performed at each step almost coincide in both approaches. However, unlike the algorithm based on Q, the algorithm based on using $A(Q)$ performs less computations at each node as the intersection of guards for each input and each set Q_k of states will be performed only once when deriving $A(Q)$. To the best of our knowledge, no experiments were conducted for deriving adaptive distinguishing sequences and it would be interesting to assess the length of adaptive distinguishing sequences in practice and to evaluate the performance of the above approaches with respect to large size FSM specifications.

4 Conclusion

In this paper, a method for distinguishing two complete possibly nondeterministic TFSMs is presented based on an FSM abstraction of the intersection of the two TFSMs. The abstraction is derived by appropriate partitioning the input and output time domains. It is shown how a traditional preset FSM-based method can be used for deriving a separating sequence for the given TFSMs using the FSM abstraction. In addition, using the FSM abstraction, we present an algorithm for deriving an r-distinguishing TFSM that represents a simple adaptive distinguish-

ing experiment for two given TFSMs. We compare the complexity of a proposed approach with that of another approach that is based directly on the intersection of two given TFSMs and show that in both approaches, similar to untimed FSMs, when distinguishing two TFSMs with m and n states, the length of a longest trace of a corresponding r-distinguishing machine is at most mn, while the length of a separating sequence is at most 2^{mn-1}, and these upper bounds are reachable [19,22].

As a future work, it would be interesting to investigate the possibility of adapting the presented work for distinguishing more than two machines as well as for a TFSM model with multiple clocks where the main challenge is the derivation of appropriate partitions of input and output time domains. In addition, it would be interesting to experiment and assess the performance of the proposed methods using large size specifications.

Acknowledgements

The authors would like to thank Dr. Zoltán Ésik and the anonymous reviewers for their helpful comments for improving the manuscript.

References

[1] Alur, Rajeev, Courcoubetis, Costas, and Yannakakis, Mihalis. Distinguishing tests for nondeterministic and probabilistic machines. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, STOC '95, pages 363–372, New York, NY, USA, 1995. ACM.

[2] Bochmann, Gregor V. and Petrenko, Alexandre. Protocol testing: review of methods and relevance for software testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '94, pages 109–124, New York, NY, USA, 1994. ACM.

[3] Dorofeeva, Rita, El-Fakih, Khaled, Maag, Stephane, Cavalli, Ana R., and Yevtushenko, Nina. Fsm-based conformance testing methods: A survey annotated with experimental evaluation. *Inf. Softw. Technol.*, 52(12):1286–1297, December 2010.

[4] Gill, Arthur. Sate-identification experiments in finite automata. *Information and Control*, 4(2-3):132–154, 1961.

[5] Gromov, M. L., Evtushenko, N. V., and Kolomeets, A. V. On the synthesis of adaptive tests for nondeterministic finite state machines. *Program. Comput. Softw.*, 34(6):322–329, 2008.

[6] Gromov, Maxim, El-Fakih, Khaled, Shabaldina, Natalia, and Yevtushenko, Nina. Distinguing non-deterministic timed finite state machines. In *Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and*

29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems, FMOODS '09/FORTE '09, pages 137–151, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] Hierons, Rob M. Testing from a nondeterministic finite state machine using adaptive state counting. *IEEE Trans. Comput.*, 53(10):1330–1342, October 2004.

[8] Hierons, Robert M., Merayo, Mercedes G., and Núñez, Manuel. Testing from a stochastic timed system with a fault model. *J. Log. Algebr. Program.*, 78(2):98–115, 2009.

[9] Kohavi, Zvi. *Switching and Finite Automata Theory.* McGraw-Hill, 1978.

[10] Krichen, Moez and Tripakis, Stavros. State identification problems for timed automata. In *Proceedings of the 17th IFIP TC6/WG 6.1 international conference on Testing of Communicating Systems*, TestCom'05, pages 175–191, Berlin, Heidelberg, 2005. Springer-Verlag.

[11] Lee, David and Yannakakis, Mihalis. Testing finite-state machines: State identification and verification. *IEEE Trans. Comput.*, 43(3):306–320, March 1994.

[12] Lee, David and Yannakakis, Mihalis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[13] Mathur, Aditya P. *Foundations of Software Testing.* Addison-Wesley Professional, 1st edition, 2008.

[14] Merayo, Mercedes G., Núñez, Manuel, and Rodríguez, Ismael. Extending efsms to specify and test timed systems with action durations and timeouts. In *Proceedings of the 26th IFIP WG 6.1 international conference on Formal Techniques for Networked and Distributed Systems*, FORTE'06, pages 372–387, Berlin, Heidelberg, 2006. Springer-Verlag.

[15] Merayo, Mercedes G., Núñez, Manuel, and Rodríguez, Ismael. Formal testing from timed finite state machines. *Computer Networks*, 52(2):432–460, 2008.

[16] Petrenko, Alexandre and Yevtushenko, Nina. Conformance tests as checking experiments for partial nondeterministic fsm. In *Proceedings of the 5th international conference on Formal Approaches to Software Testing*, FATES'05, pages 118–133, Berlin, Heidelberg, 2006. Springer-Verlag.

[17] Petrenko, Alexandre and Yevtushenko, Nina. Adaptive testing of deterministic implementations specified by nondeterministic fsms. In *Proceedings of the 23rd IFIP WG 6.1 international conference on Testing software and systems*, ICTSS'11, pages 162–178, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] Shabaldina, Natalia, El-Fakih, Khaled, and Yevtushenko, Nina. Testing nondeterministic finite state machines with respect to the separability relation. In *Proceedings of the 19th IFIP TC6/WG6.1 international conference, and 7th international conference on Testing of Software and Communicating Systems*, TestCom'07/FATES'07, pages 305–318, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] Spitsyna, Natalia, El-Fakih, Khaled, and Yevtushenko, Nina. Studying the separability relation between finite state machines. *Softw. Test. Verif. Reliab.*, 17(4):227–241, December 2007.

[20] Starke, Peter H. *Abstract Automata.* Elsevier, 1972.

[21] Tanenbaum, Andrew S. *Computer networks.* Prentice-Hall, 3 edition, 1996.

[22] Yevtushenko, Nina and Spitsyna, Natalia. On the upper of length of separating and r-distinguishing sequences for observable nondeterministic FSMs. In *Proceedings of Artificial intelligence systems and computer sciences*, pages 124–126, 2005. (in Russian).

[23] Yevtushenko, Nina, Vetrova, Maria, and Petrenko, Alexandre. *Analysis and synthesis of nondeterministic FSMs: operators and relations.* Tomsk State University publishing, 2006. (in Russian).

Received 9th May 2012

Acta Cybernetica **21** (2013) 223–234.

On Shuffle Ideals of General Algebras

Ville Piirainen*

Abstract

We extend a word language concept called shuffle ideal to general algebras. For this purpose, we introduce the relation $\mathcal{SH}$ and show that there exists a natural connection between this relation and the homeomorphic embedding order on trees. We establish connections between shuffle ideals, monotonically ordered algebras and automata, and piecewise testable tree languages.

1 Introduction and preliminaries

This work is a part of an ongoing study on piecewise testability and related matters for tree languages. Piecewise testable languages and their algebraic properties have been approached from various directions, and offer a wide field of interesting notions for study from the tree language viewpoint. In addition to the ingenious combinatorial approach of Simon [10], there have been a few approaches with a more algebraic flavour, and this work is inspired most importantly by the papers by Straubing and Thérien [12], and Henckell and Pin [5]. These works concern, of course, word languages, subsets of a free monoid X^*, and obviously are not directly generalizable for tree languages, subsets of a term algebra $\mathcal{T}_\Sigma(X)$. However, all these papers contain many algebraic insights that can be considered in the tree language setting. We are much indebted to the work on ordered monoids in these papers, as well as to the related work on varieties of ordered algebras by Bloom [2], and Petković and Salehi [6].

The shuffle operation is a natural operation to consider for the elements of a free monoid. Using this operation one obtains so called shuffle ideals, which are subsets of a free monoid closed under the shuffle operation. As noted for example in [9], by considering all boolean combinations of shuffle ideals on a given free monoid, one obtains exactly all piecewise testable languages over that monoid. In fact, the shuffle, the class of piecewise testable languages, the Green's $\mathcal{J}$-relation for semigroups and the class of monotonically ordered monoids are all concepts which are strongly connected to each other, and we shall use these connections to investigate the notion of shuffling for general algebras.

The shuffle operation cannot be directly defined for any given ΣX-trees, since even the product of two trees cannot be uniquely defined in a way that would suit

*University of Turku, E-mail: ville.piirainen@utu.fi

all applications. While the operation itself does not generalize directly, the shuffle ideals, as languages, have direct counterparts in the tree language setting, as we shall see.

After this first section of introduction and preliminaries, in the second section, we introduce the shuffle relation $\mathcal{SH}$ and the shuffle ideals, and investigate their basic properties. In the third section, we establish a connection between so-called monotonically ordered algebras and the $\mathcal{SH}$-relation. Finally, we discuss some connections between the relation $\mathcal{SH}$ and piecewise testable tree languages.

As a general reference on algebraic tree language theory, we recommend [11]. It contains most of the basic theory on which this paper is built, and also some discussion on the points one has to take into account when moving from word languages to tree languages. However, we recall here a few of the most important definitions and notions that we need in this paper, since some of them have various different versions in the literature.

We are mainly interested in trees and their languages, and we follow the theoretical framework of [11] which depends heavily on universal algebra. The tree recognizers, general algebras, have a finite number of named operations, from which all other operations of the algebra are composed. Moreover, the number of arguments of each operation is fixed. Hence, trees considered here are terms over suitable alphabets, in which each node of a tree labeled with a given symbol always has a fixed number of children. We use the following notation.

Definition 1.1. *A* ranked alphabet Σ *is a finite set of function symbols, and for all* $m \geq 0$, $\Sigma_m \subseteq \Sigma$ *denotes the subset of symbols of rank* m. *A* Σ-algebra $\mathcal{A} = (A, \Sigma)$ *consists of a non-empty set* A *equipped with operations* $f^{\mathcal{A}} : A^m \to A$, *for all* $m \geq 0$, $f \in \Sigma_m$.

For the rest of the paper, $\mathcal{A} = (A, \Sigma)$ is an arbitrary given Σ-algebra.

In the framework we use, the inner nodes and leafs of a tree have different labelings. In addition to ranked alphabets, we use leaf-alphabets, finite sets of symbols that are disjoint from the ranked alphabets. We identify trees with terms defined in the following definition.

Definition 1.2. *For a set* X, *called the* leaf alphabet, *the set of all* ΣX-terms $T_\Sigma(X)$ *is the smallest set such that* $X \cup \Sigma_0 \in T_\Sigma(X)$, *and for every* $m > 0$, $t_1, \ldots, t_m \in T_\Sigma(X)$ *and* $f \in \Sigma_m$, $f(t_1, \ldots, t_m) \in T_\Sigma(X)$.

For transforming a word concept into a tree concept we need a way to regard words as special trees. As usual, we regard words over an alphabet A as unary trees equipped with a single special leaf symbol ξ, and letters of the alphabet A are regarded as unary symbols of the ranked alphabet Σ. More precisely, let A be an alphabet, let $X = \{\xi\}$ and let $\Sigma = \Sigma_1 = A$. Let $\chi : A^* \to T_\Sigma(X)$ be the map such that $\varepsilon\chi = \xi$ and $(wa)\chi = a(w\chi)$ for any $a \in A$ and $w \in A^*$. Obviously, χ forms a bijective correspondence between A^* and $T_\Sigma(X)$.

For the purpose of generalizing the semigroup concept shuffle for Σ-algebras, we have chosen to follow the convention that the root of a ΣX-term corresponds

to the right end, and the leaf symbols to the left end of a word. This follows the usual tradition on how words and terms (trees) are read by their respective ordinary automata, from left to right and from leaf to root. This convention has the following consequences. The right translations of semigroups correspond to the algebraic translations of the term algebra $\mathcal{T}_\Sigma(X)$ of ΣX-trees, while the left translations correspond to the endomorphisms of the same term algebra. We use the translations in our effort to generalize the ideas of insertion and the shuffle ideal for trees in Section 3.

Definition 1.3. *A unary map $p : A \to A$ is an* elementary translation *of an algebra $\mathcal{A}$, if there exist $m > 0$, $f \in \Sigma_m$, $i = 1, \ldots, m$, and $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_m$ such that*

$$p(a) = f^{\mathcal{A}}(a_1, \ldots, a_{i-1}, a, a_{i+1}, \ldots, a_m),$$

for all $a \in A$. The set of all elementary translations of $\mathcal{A}$ is denoted $\mathrm{ETr}(\mathcal{A})$. *The set of* translations *of $\mathcal{A}$, denoted* $\mathrm{Tr}(\mathcal{A})$, *is the smallest set which includes the identity map and the elementary translations, and is closed under functional composition.*

The translations of a term algebra $\mathcal{T}_\Sigma(X)$ are induced by the ΣX-*contexts*, that is, the trees $p \in T_\Sigma(X \cup \{\xi\})$, where the symbol ξ appears exactly once. To simplify notation, a context $p \in T_\Sigma(X \cup \{\xi\})$ and the map $\hat{p} : T_\Sigma(X) \to T_\Sigma(X)$, $t \mapsto p(t)$ it induces are identified.

The concept of an ideal is common in algebra, and we introduce here a certain type of an ideal. We note that since we consider here general algebras with no additional requirements, the ideals presented here might differ from ideals defined for different purposes. The theory investigated here is closely related to that of ordered algebras, and as a reference concerning notation and points of view, we offer [6]. From this paper we adopt the following definition.

Definition 1.4. *An* ideal *of an algebra $\mathcal{A}$ is a non-empty set $I \subseteq A$ such that for any $p \in \mathrm{Tr}(\mathcal{A})$, and $a \in I$, $p(a) \in I$. The ideal generated by an element a is denoted $I(a)$.*

In essence, this definition states that if we choose any element from the ideal, any $n-1$ elements of the algebra ($n > 0$), and apply to them any n-ary function of the algebra, the resulting element is still in this ideal. Hence, the notion resembles that of a semigroup ideal, though not that of a Dedekind ideal. Namely, in ring theory there is such a distinction between the two operations that cannot be required in any given arbitrary Σ-algebra in a meaningful way.

In our effort to generalize the idea of shuffling for general algebras, and for non-linear trees, we have taken as a starting point the following definition from [9].

Definition 1.5. *For an alphabet X, a* shuffle ideal *of the free monoid X^* is a non-empty set $I \subseteq X^*$ such that for any words $u \in I$ and $v \in X^*$, their shuffle is included in the set I.*

For any word $u \in X^*$, if $u = x_1 \cdots x_n$ where $x_1, \ldots, x_n \in X$, then the shuffle ideal generated by u is the language $X^* x_1 X^* \cdots X^* x_n X^*$.

We connect the shuffle ideal to the homeomorphic embedding relation used in term rewriting theory. When words are interpreted as unary trees, it turns out that these notions are very naturally related to one another (see Example 2.2).

Definition 1.6. *The* homeomorphic embedding relation $\leq_{emb}$ *on* $T_\Sigma(X)$ *is defined as follows. For any* $s, t \in T_\Sigma(X)$, $s \leq_{emb} t$ *if and only if,*

(1) $t \in X \cup \Sigma$ *and* $s = t$, *or*

(2) $t = f(t_1, \ldots, t_m), s = f(s_1, \ldots, s_m)$ *and* $s_i \leq_{emb} t_i$ *for* $i = 1, \ldots, m$, *or*

(3) $t = f(t_1, \ldots, t_m)$ *and* $s \leq_{emb} t_i$ *for some* $i = 1, \ldots, m$.

If $s \leq_{emb} t$ $(s, t \in T_\Sigma(X))$, then essentially this means that all the nodes of the term s are embedded in the structure of t, in such a way that they retain their rank (arity) and relative position. For example, if $X = \{x, y\}$, and $\Sigma = \{g/1, f/2, h/2\}$, then

$$x \leq_{emb} f(x, y) \leq_{emb} f(g(x), h(y, x)) \leq_{emb} h(f(g(x), h(g(y), x)), h(x, y)).$$

2 Shuffle ideal

What we call a shuffle ideal borrows ideas from the shuffle operation and ideal defined for word languages (see [9]) as well as the embedding relation from rewriting theory (see [1]). These notions share a common idea: starting from a single element of a language, using suitable insertions, obtain the elements which contain the original element embedded in their structure. We begin by defining a relation that specifies the types of insertions in which we are interested here.

Definition 2.1. *Let* $\Rightarrow_{\mathcal{SH}}$ *be the relation on* A *such that for any* $a, b \in A$

$$a \Rightarrow_{\mathcal{SH}} b,$$

if and only if there exist an element $c \in A$ *and translations* $q, r \in \mathrm{Tr}(\mathcal{A})$ *such that* $a = q(c)$ *and* $b = q(r(c))$.

In essence, we decompose the element a into a product of an element c and a translation q, and then insert an another translation r into the middle of the product.

In the next example we show concretely how such insertions work in a term algebra $\mathcal{T}_\Sigma(X)$. The original term, which is embedded in the derived terms, is printed in boldface.

Example 2.1. Let $\Sigma = \{f/2, g/1\}$ and $X = \{x, y\}$. Then, for example

$$\mathbf{f(x, y)} \Rightarrow_{\mathcal{SH}} \mathbf{f}(f(y, \mathbf{x}), \mathbf{y}) \Rightarrow_{\mathcal{SH}} \mathbf{f}(f(y, \mathbf{x}), g(\mathbf{y})) \Rightarrow_{\mathcal{SH}} \mathbf{f}(f(f(y, y), \mathbf{x}), g(\mathbf{y})).$$

Consider for example the second step of the derivation. We can write $f(f(y,x),y) = f(f(y,x),\xi)(y)$, and by applying the context $g(\xi)$ we obtain $f(f(y,x),\xi)(g(\xi)(y)) = f(f(y,x),g(y))$.

In the following example we show how derivations can be made in the free monoid generated by the alphabet $\{a,b\}$. We denote by e the empty word, and by $u\xi v \in \mathrm{Tr}(X^*)$, for any $u,v \in X^*$, the (two-sided) translation such that $u\xi v(w) = uwv$.

Example 2.2. Let $X = \{a,b\}$, and let $w,w',w'' \in X^*$. We have for example the following derivation.

$$ab \Rightarrow_{\mathcal{SH}} aw'b \Rightarrow_{\mathcal{SH}} waw'bw''.$$

In the first step we can write that $ab = a\xi b(e)$, and further apply the translation $w'\xi e$ to obtain $a\xi b(w'\xi e(e)) = aw'b$. In the second step, we write first $aw'b = \xi(aw'b)$, and by using the translation $w\xi w''$ we obtain $\xi(w\xi w''(aw'b)) = \xi(waw'bw'') = waw'bw''$. In general, it is easy to see, that $ab \Rightarrow^*_{\mathcal{SH}} w$ if and only if $w \in X^*aX^*bX^*$.

The following lemmas are direct consequences of the Definition 2.1.

Lemma 2.1. *For all $a \in A$ and $p \in \mathrm{Tr}(\mathcal{A})$, $a \Rightarrow_{\mathcal{SH}} p(a)$.*

Proof. Let $a \in A$. Then, $a = \mathrm{id}(a)$, and $\mathrm{id}(p(a)) = p(a)$, for any $p \in \mathrm{Tr}(\mathcal{A})$. □

Lemma 2.2. *If $a \Rightarrow_{\mathcal{SH}} b$, then $p(a) \Rightarrow_{\mathcal{SH}} p(b)$, for any $p \in \mathrm{Tr}(\mathcal{A})$ and $a,b \in A$.*

Proof. If $a = q(c)$, and $b = q(r(c))$, for some $c \in A$ and $r,q \in \mathrm{Tr}(\mathcal{A})$, then $p(a) = p(q(c))$, and $p(b) = p(q(r(c)))$, for any $p \in \mathrm{Tr}(\mathcal{A})$, which proves the claim. □

As usual, we denote

$$\Rightarrow^*_{\mathcal{SH}} = \bigcup_{n\geq 0} \Rightarrow^n_{\mathcal{SH}}.$$

Definition 2.2. *We call a non-empty subset $I \subseteq A$ a* shuffle ideal *of $\mathcal{A} = (A,\Sigma)$, if for all $a,b \in A$,*

(SI) $a \in I$ and $a \Rightarrow_{\mathcal{SH}} b$ imply $b \in I$.

The following lemma is easy to prove.

Lemma 2.3. *The intersection of a set of shuffle ideals is either empty or a shuffle ideal.*

By the previous lemma, for a given element $a \in A$, we can define the *shuffle ideal generated by a* as the intersection of the shuffle ideals containing a. We denote this by $SH(a)$.

Lemma 2.4. *For any $a \in A$, $SH(a) = \{b \in A \mid a \Rightarrow^*_{\mathcal{SH}} b\}$.*

The following lemma is a direct consequence of Lemma 2.1.

Lemma 2.5. *For all $a \in A$ and $p \in \mathrm{Tr}(\mathcal{A})$, $SH(p(a)) \subseteq SH(a)$.*

Note that a shuffle ideal is always an ideal. The shuffle ideal generated by an element contains the ideal generated by the same element, but in general these sets are not the same, as demonstrated by the following example.

Example 2.3. Let $\mathcal{A} = (\{1,2,3\}, \{f/1, g/1\})$ be the algebra described in Figure 1, originally presented in [7]. A direct calculation shows that $I(3) = \{3\}$ but $SH(3) = \{2,3\}$.

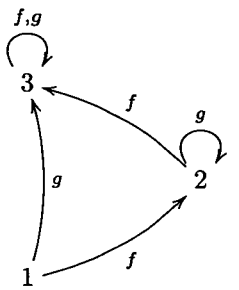

Figure 1: The algebra $\mathcal{A}$

Note that when interpreted for the free monoid X^*, the shuffle ideal generated by a word $w \in X^*$ corresponds exactly to the original notion. Indeed, if X is an alphabet, $w = x_1 \cdots x_n \in X^*$ and

$$u = u_1 x_1 u_2 \cdots u_n x_n u_{n+1} \in X^* x_1 X^* \cdots X^* x_n X^*,$$

then

$$x_n(\cdots x_1(\xi)\cdots) \Rightarrow^*_{\mathcal{SH}} u_{n+1}(x_n(u_n(\cdots(u_2(x_1(u_1(\xi))))\cdots))),$$

by Lemma 2.4. The converse is analogous.

Lemma 2.4 also gives us a naive algorithm to calculate the shuffle ideals $SH(a)$ of a finite algebra. The algorithm works in two parts. First we calculate $a \Rightarrow_{\mathcal{SH}}$ for each element $a \in A$, and then the equivalence closure $a \Rightarrow^*_{\mathcal{SH}}$.

1. Compute the table of translations for the algebra.
2. For each element $a \in A$ find all possible decompositions $a = p(b)$ $(p \in \mathrm{Tr}(\mathcal{A}), b \in A)$ from the table of translations.
3. For each decomposition $a = p(b)$, form all elements $p(r(b)))$, where $r \in \mathrm{Tr}(\mathcal{A})$. These elements form the sets $a \Rightarrow_{\mathcal{SH}}$.
4. Compute the reflexive transitive closure $\Rightarrow^*_{\mathcal{SH}}$ of the relation $\Rightarrow_{\mathcal{SH}}$.

Since the algorithm follows exactly the steps of the definitions of the shuffle ideal and the shuffle relation, it is obvious that this algorithm produces exactly the desired sets $SH(a)$ for all $a \in A$.

The complexity of the algorithm depends heavily on the structure of the algebra and its translation monoid $\mathrm{Tr}(\mathcal{A})$. In most of any meaningful examples Σ is fixed, so we measure complexity based only on $|A|$. It is worth mentioning though, that by choosing a suitable ranked alphabet Σ, one can easily devise exotic algebras such that the complexity of computing the elementary translations of the algebra exceeds any given bound which is dependent only on the size $|A|$ of the algebra, and hence the following analysis is not applicable universally. However, even in such exotic cases the number of different elementary translations has an upper bound which depends only on the size of $|A|$. Hence, we assume that we are given elementary translations induced by the algebra as the input for the algorithm.

If $|A| = n$, then the size of the translation monoid may equal n^n (the full transformation monoid on A), and its calculation that starts from the elementary translations may have a complexity of as high as $\mathcal{O}(n^{3n+1})$ depending on the size and structure of $\mathrm{ETr}(\mathcal{A})$. The size of table of translations may in the worst case equal n^{n+1}. Hence, the number of calculations generated by the third step of the algorithm may equal n^{2n+1}. The transitive closure can be calculated in $\mathcal{O}(n^3)$ time.

Next we show a concrete example of how the algorithm works.

Example 2.4. Let $\Sigma = \{f/1, g/1\}$, $A = \{1, 2, 3, 4, 5\}$, and let the operations be defined as in Figure 2.

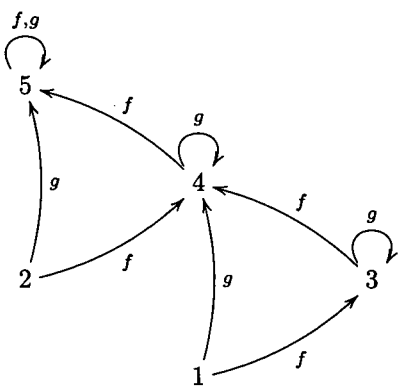

Figure 2: The algebra $\mathcal{A}$.

A direct calculation gives the table of translations for the algebra shown in Table 1. Note that for simplicity we have identified unary function symbols with the translations they define, and denoted fg the operation such that $(fg)(a) = f(g(a))$ for all $a \in A$.

Table 1: Table of translations for $\mathcal{A}$.

$\mathrm{Tr}(\mathcal{A})$	1	2	3	4	5
id	1	2	3	4	5
f	3	4	4	5	5
g	4	5	3	4	5
ff	4	5	5	5	5
fg	5	5	4	5	5
fff	5	5	5	5	5

Consider for example $SH(5)$. We have that $5 = g(2)$, which implies that $g(f(2)) = 4 \in SH(5)$, and continuing similarly $g(f(1)) = 3 \in SH(5)$. By performing the steps of our algorithm for all such decompositions we obtain the sets

$$\begin{aligned} SH(1) &= \{1,3,4,5\} \\ SH(2) &= \{2,3,4,5\} \\ SH(3) = SH(4) &= SH(5) = \{3,4,5\} \end{aligned}$$

We can form a quasi-order on a given algebra based on the inclusion of the ideals $SH(a)$. We denote this relation by $\leq_{\mathcal{SH}}$, and we define it so that for all $a, b \in A$,

$$a \leq_{\mathcal{SH}} b \iff SH(a) \supseteq SH(b).$$

In fact, $\leq_{\mathcal{SH}} = \Rightarrow^*_{\mathcal{SH}}$.

In the spirit of Green's relations, we define $\mathcal{SH} \subseteq A^2$ as the relation such that

$$a \; \mathcal{SH} \; b \iff SH(a) = SH(b).$$

By Lemma 2.2 it is a congruence. We say that $\mathcal{A}$ is *$\mathcal{SH}$-trivial* if $a \; \mathcal{SH} \; b$ implies $a = b$. It is clear, that the algebra is $\mathcal{SH}$-trivial, if and only if $\leq_{\mathcal{SH}}$ is an order. In the next section we investigate the properties of this order further.

As we saw in Example 2.4, the shuffle ideals $SH(a)$ of a given finite algebra can be calculated using the algorithm presented earlier in this section. We can then calculate the quasi-order $\leq_{\mathcal{SH}}$, and also determine whether the algebra is $\mathcal{SH}$-trivial or not.

3 Monotonically ordered algebras

In this section we investigate algebras which are equipped with a certain type of an order, namely a monotone order (see [3]). We show that algebras equipped with an admissible monotone order are bijectively connected to $\mathcal{SH}$-triviality.

Definition 3.1. *An algebra $\mathcal{A}$ is* monotone, *if there exists an order $\leq$ on A such that for all $n \geq 1$, $f \in \Sigma_n$ and $a_1, \ldots, a_n \in A$,*

(M) $a_1, \ldots, a_n \leq f^{\mathcal{A}}(a_1, \ldots, a_n)$.

Note that the condition (M) can be replaced with an equivalent condition: $a \leq p(a)$ for all $a \in A$ and for all $p \in \mathrm{ETr}(\mathcal{A})$.

Let us recall that a relation on a set is called a *pre-order* if it is reflexive and transitive.

Definition 3.2. *Let θ be a pre-order on A. It is* admissible, *if $a_1\ \theta\ b_1, \ldots, a_n\ \theta\ b_n$ imply $f^{\mathcal{A}}(a_1, \ldots, a_n)\ \theta\ f^{\mathcal{A}}(b_1, \ldots, b_n)$ for all $n \geq 0$, $a_1, \ldots, a_n, b_1, \ldots, b_n \in A$ and $f \in \Sigma_n$.*

Equivalently, a pre-order θ is admissible, if for all $a, b \in A$ and for all $p \in \mathrm{ETr}(\mathcal{A})$, $a\ \theta\ b$ implies $p(a)\ \theta\ p(b)$. An *ordered algebra* $(\mathcal{A}, \leq)$ consists of an algebra, and an admissible order $\leq$ on A.

An ordered algebra $(\mathcal{A}, \leq)$ is *monotone* if (M) is satisfied for the given order $\leq$. Following the definition presented in [7] we call an algebra $\mathcal{A}$ *monotonically ordered* if there exists an ordered algebra $(\mathcal{A}, \leq)$ which is monotone. Note that in [7] we used the term monotonously ordered.

Before our main result we prove a useful lemma.

Lemma 3.1. *If $(\mathcal{A}, \leq)$ is monotone, then $a \Rightarrow_{\mathcal{SH}} b$ implies $a \leq b$ for all $a, b \in A$.*

Proof. Let $a, b \in A$ be such that $a \Rightarrow_{\mathcal{SH}} b$. There exist $q, r \in \mathrm{Tr}(\mathcal{A})$ and $c \in A$ such that $a = q(c)$ and $b = q(r(c))$. Now, by the properties of the monotone order on A, we have that $c \leq r(c)$, and hence $a = q(c) \leq q(r(c)) = b$. □

Theorem 3.1. *An algebra $\mathcal{A}$ is monotonically ordered if and only if it is $\mathcal{SH}$-trivial.*

Proof. Assume that $\mathcal{A}$ is $\mathcal{SH}$-trivial. Then, $\leq_{\mathcal{SH}}$ is a partial order on A. Also, $a \leq_{\mathcal{SH}} p(a)$, since $SH(p(a)) \subseteq SH(a)$ by Lemma 2.5.

For proving that the order is admissible, let $a, b \in A$ be such that $a \leq_{\mathcal{SH}} b$. Now, $b \in SH(a)$, and hence $a \Rightarrow^*_{\mathcal{SH}} b$ by Lemma 2.4, which means that for some $n \geq 0$, $a \Rightarrow^n_{\mathcal{SH}} b$. By Lemma 2.2, it follows that $p(a) \Rightarrow^*_{\mathcal{SH}} p(b)$, which implies that $p(b) \in SH(p(a))$, and therefore $p(a) \leq_{\mathcal{SH}} p(b)$.

For the other direction, let $(\mathcal{A}, \leq)$ be monotone. Assume that $SH(a) = SH(b)$ for some $a, b \in A$. Then, $a \Rightarrow^*_{\mathcal{SH}} b$. Now, by Lemma 3.1 we get directly that $a \leq b$. By a symmetric argument also $b \leq a$, which implies $a = b$, which proves that $\mathcal{A}$ is $\mathcal{SH}$-trivial. □

In the next proposition we show that the order $\leq_{\mathcal{SH}}$ is the least admissible and monotone order on a given monotonically ordered algebra. Before that, we give a simple example which shows that such an order on an algebra need not be unique.

Example 3.1. Let $\Sigma = \{f/1\}$ and $A = \{a, b\}$. Define the algebra $\mathcal{A}$ so that $f^{\mathcal{A}}(a) = a$ and $f^{\mathcal{A}}(b) = b$. Now, $\leq_{\mathcal{SH}} = \Delta_A$, but the relation $\{(a, a), (a, b), (b, b)\}$ is also a monotone and admissible ordering for $\mathcal{A}$.

Proposition 3.1. *If an ordered algebra $(\mathcal{A}, \leq)$ is monotone, then $\leq_{\mathcal{SH}} \subseteq \leq$.*

Proof. If $a \leq_{\mathcal{SH}} b$, for some $a, b \in A$, then $a \Rightarrow^*_{\mathcal{SH}} b$, and Lemma 3.1 implies directly that $a \leq b$. □

As we shall see, in the term algebra $\mathcal{T}_\Sigma(X)$, the relation $\Rightarrow^*_{\mathcal{SH}}$ equals the homeomorphic embedding relation of terms. Thus, $\Rightarrow^*_{\mathcal{SH}}$ can be regarded as a generalization of the embedding relation for general algebras. Before the proposition, we note an obvious lemma.

Lemma 3.2. *For any leaf alphabet X and ranked alphabet Σ, the algebra $\mathcal{T}_\Sigma(X)$ is monotonically ordered by $\leq_{emb}$.*

Proposition 3.2. *For any X and Σ, and $s, t \in T_\Sigma(X)$,*

$$s \leq_{emb} t \text{ if and only if } s \Rightarrow^*_{\mathcal{SH}} t$$

Proof. It follows immediately from the previous lemma, and Lemma 3.1, that $\Rightarrow^*_{\mathcal{SH}} \subseteq \leq_{emb}$.

For the other direction, we proceed by structural induction following the definition of the relation $\leq_{emb}$. Note that by the previous lemma, $\mathcal{T}_\Sigma(X)$ is monotonically ordered, or equivalently $\mathcal{SH}$-trivial (Theorem 3.1), and $\Rightarrow^*_{\mathcal{SH}}$ is an admissible, monotone order. Assume that $s \leq_{emb} t$.

1. If $s = t$, then $s \Rightarrow_{\mathcal{SH}} t$.
2. Assume that $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$, where $s_i \leq_{emb} t_i$ for $i = 1, \ldots, n$, and assume that the claim holds for s_i and t_i for all $i = 1, \ldots, n$. Then, $s_i \Rightarrow^*_{\mathcal{SH}} t_i$ for $i = 1, \ldots, n$, and by the $\mathcal{SH}$-triviality of $\mathcal{T}_\Sigma(X)$, $f(s_1, \ldots, s_n) \Rightarrow^*_{\mathcal{SH}} f(t_1, \ldots, t_n)$.
3. Assume that $t = f(t_1, \ldots, t_n)$ and $s \leq_{emb} t_i$ for some $i = 1, \ldots, n$, and assume that the claim holds for s and t_i. Then, $s \leq_{emb} t_i$ implies $s \Rightarrow^*_{\mathcal{SH}} t_i \Rightarrow_{\mathcal{SH}} t$.

□

We conclude the section by considering some variety properties of monotonically ordered algebras. The class of $\mathcal{SH}$-trivial algebras (i.e. that of monotonically ordered algebras) is closed under forming direct products and subalgebras, but not homomorphic images [7]. Hence, the class is not a variety. However, in the following we show that the class of monotone ordered algebras is closed under order-preserving homomorphisms, which makes it a variety of ordered algebras [2].

In [2] a pre-order on an ordered algebra is said to be *admissible*, if it is an admissible relation, and contains the ordering of the algebra. If $\precsim$ is an admissible pre-order on $\mathcal{A}$, then $\sim = \precsim \cap \succsim$ is a congruence on $\mathcal{A}$, and $\mathcal{A}/\sim$ is ordered by the relation $\preceq$ defined so that for all $a, b \in A$, $a/\sim \preceq b/\sim$ if and only if $a \precsim b$ (see [2], p. 201).

Proposition 3.3. *The class of monotone ordered algebras is closed under order-preserving homomorphisms, i.e. homomorphisms of ordered algebras.*

Proof. Let $(\mathcal{A}, \leq)$ be a monotone ordered algebra. By Proposition 1.3 in [2], it is sufficient to look at the quotient algebras with respect to the admissible pre-orders on $(\mathcal{A}, \leq)$. Hence, assume that $\precsim$ is an admissible pre-order, and consider the order $\preceq$ on $\mathcal{A}/\sim$ derived from $\precsim$, where $\sim = \precsim \cap \succsim$.

Now, let $n \geq 0$, $a_1, \ldots, a_n \in A$, and $f \in \Sigma_n$. For every $i = 1, \ldots, n$, it follows from $a_i \leq f^{\mathcal{A}}(a_1, \ldots, a_n)$ that $a_i/\sim \preceq f^{\mathcal{A}}(a_1, \ldots, a_n)/\sim = f^{\mathcal{A}/\sim}(a_1/\sim, \ldots, a_n/\sim)$. □

Theorem 2.6 in [2] states that every variety of ordered algebras is defined by a set of inequalities. In the case of monotone orders such a set is immediately given by the definition.

Example 3.2. If $\Sigma = \{f/2\}$, then the class of monotone ordered Σ-algeras is defined by the set $\{x \leq f(x,y),\ y \leq f(x,y)\}$.

The class of languages corresponding to the class of finite monotonically ordered algebras can be characterized as follows. The k-piecewise testable tree languages for some fixed Σ and X were defined in [7] as the unions of π^k-classes, for a certain finite congruence π^k. It was also proved that the algebra $\mathcal{T}_\Sigma(X)/\pi^k$ is monotonically ordered. Hence, each piecewise testable tree language can be recognized by a finite monotonically ordered algebra, and it was shown also in [7], that all languages recognized by finite monotonically ordered algebras are piecewise testable.

It is clear that the languages recognized by finite monotone ordered algebras in the sense of [6] are included in the variety of tree languages corresponding to the variety of finite algebras generated by the finite monotonically ordered algebras, which are exactly the piecewise testable tree languages. Hence, all languages recognized by finite monotone ordered algebras are piecewise testable. However, for example the language $\{x\} \subseteq T_\Sigma(X)$, where $X = \{x\}$ and $\Sigma = \{f/1\}$, cannot be recognized by a monotone ordered algebra in the sense of [6], even if the language is most certainly piecewise testable.

A shuffle ideal of a term algebra is clearly a piecewise testable tree language. Namely, $SH(t)$ contains exactly all the terms which have t as a piecewise subtree. In fact, this implies directly that each piecewise testable tree language can be obtained as a boolean combination of suitable shuffle ideals. This generalizes the result that a piecewise testable word language is a boolean combination of shuffle ideals.

Further remarks

We presented here a natural generalization of the shuffle ideal, and we established connections between the shuffle relation, the homeomorphic embedding relation and monotonically ordered algebras. Monotonically ordered algebras and the embedding relation were very useful in our earlier work on piecewise testability for trees [7], and hence it is not surprising, that the shuffle ideals investigated here have a similar connection to piecewise testability as in the word case.

Our definition of the shuffle ideal suggests also a definition for the shuffle operation, which would be suitable for terms of term algebras and elements of general

algebras. Such a product would be defined not between two elements, but rather between a translation and an element. Each translation can be decomposed (not in a unique way in general) into a product of elementary translations, and each element of an algebra can also be decomposed into a product of elementary translations and a generator of the algebra. By merging these sequences in a similar manner as shuffling two words, one obtains elements which form a set that could be seen as the shuffle of these objects.

References

[1] Avenhaus, J. *Reduktionssysteme.* Springer-Verlag, Berlin, 1995.

[2] Bloom, S. Varieties of ordered algebras. *Journal of Computer and System Sciences* 13:200–212, 1976.

[3] Gécseg, F., and Imreh, B. On monotone automata and monotone languages. *Journal of Automata, Languages and Combinatorics*, 7(1):71–82, 2002.

[4] Grätzer, G. *Universal Algebra.* Van Nostrand Company, 1968.

[5] Henckell, K., and Pin, J.-E. Ordered monoids and $\mathcal{J}$-trivial monoids. In Birget, J.-C. et al, editors, *Algorithmic Problems in Groups and Semigroups*, pages 121–137, Birkhäuser, Boston, 2000.

[6] Petković, T., and Salehi, S. Positive varieties of tree languages. *Theoretical Computer Science*, 347(1-2):1–35, 2005.

[7] Piirainen, V. Piecewise testable tree languages. TUCS Technical Reports 634, Turku Centre for Computer Science, Turku, Finland, 2004.

[8] Piirainen, V. Monotone algebras, $\mathcal{R}$-trivial monoids and a variety of tree languages. *Bulletin of the EATCS*, 84:189–194, 2004.

[9] Pin, J.-E. Syntactic semigroups. In Rozenberg, G., and Salomaa, A., editors, *Handbook of Formal Languages, Vol. 1.: Word, language, grammar*, pages 679–746, Springer-Verlag, New York, 1997.

[10] Simon, I. Piecewise testable events. In *Automata Theory and Formal Languages, (Proc. 2nd GI conf.), Lecture Notes in Computer Science* 33:214–222, Springer-Verlag, Berlin, 1975.

[11] Steinby, M. Algebraic classifications of regular tree languages. In Kudryavtsev, V., and Rosenberg, I., editors, *Structural Theory of Automata, Semigroups and Universal Algebra*, pages 381–432, Springer, 2005.

[12] Straubing, H., and Thérien, D. Partially Ordered Finite Monoids and a Theorem of I. Simon. *Journal of Algebra*, 119:393–399, 1988.

Received 11th May 2012

Acta Cybernetica 21 (2013) 235–265.

Simulated Annealing for Aiding Genetic Algorithm in Software Architecture Synthesis*

Outi Sievi-Korte† Erkki Mäkinen‡ and Timo Poranen‡

Abstract

The dream of software engineers is to be able to automatically produce software systems based on their requirements. Automatic synthesis of software architecture has already been shown to be feasible with genetic algorithms. Genetic algorithms, however, easily become very slow if the size of the problem and complexity of mutations increase as GAs handle a large population with much data. Also, for purely scientific interest it is worthwhile to investigate how other search algorithms handle the problem of software architecture synthesis. The present paper studies the possibilities of using simulated annealing for synthesizing software architecture. For this purpose we have two goals: 1) to study whether a simpler search algorithm can handle synthesis and 2) if a seeded algorithm can provide quality results faster than a simple genetic algorithm. We start from functional requirements which form a base architecture and consider three quality attributes, modifiability, efficiency and complexity. Synthesis is performed by adding design patterns and architecture styles to the base architecture. The algorithm thus produces a software architecture which fulfills the functional requirements and also corresponds to the quality requirements. It is concluded that simulated annealing as such does not produce good architectures, but it is useful for speeding up the evolution process by quickly fine-tuning a seed solution achieved with a genetic algorithm. The main contribution is thus a new seeded algorithm for software architecture design.

Keywords: search-based software engineering, simulated annealing, software design, genetic algorithm, software architecture

1 Introduction

The ultimate goal of software engineering is to be able to automatically produce software systems based on their requirements. In Model Driven Architecture

*This work was funded by the Academy of Finland (project Darwin).

†Tampere University of Technology - corresponding author, E-mail: outi.sievi-korte@tut.fi, Address: Department of Software Systems, Korkeakoulunkatu 1, P.O.Box 553, 33101 Tampere, Finland

‡University of Tampere, E-mail: {erkki.makinen, timo.t.poranen}@uta.fi

(MDA), class level designs of the software can already be transformed straightforwardly into code [10]. However, a human is still required to interpret the given quality requirements and build the class level design, or architecture, based on which code can be written. This process is time consuming and requires expertise, as systems become increasingly large and complex and the quality requirements are often conflicting. Errors in design phase are unfortunately common, and have a large impact on the functionality of the system. Our goal is to automate the process of turning requirements into software architecture, where quality requirements are not only met but also optimized to suit the preferences of the client.

Architectural design largely means the application of known standard solutions in a combination that optimizes the quality properties (like modifiability and efficiency) of the software system. These standard solutions are well documented as architectural styles [39] and design patterns [11]. We argue that software architecture comes in parts: the functional requirements, the quality requirements and the actual architectural design solutions. The functional and quality requirements can only be elicited manually, but combining them to design solutions and, thus, producing a complete architecture, which is more than the sum of its parts, can be done automatically. Hence, we see the formation of software architecture as a series of transformations beginning with a very crude outline of a system with only the basic functionalities and ending with a highly sophisticated design. So far, this has been accomplished by humans. Thus, as we view that software architecture requires combining different entities (design solutions and requirements) and the automatic process is synthetic when compared to man made architectures, we refer to our approach as architecture synthesis.

Seeing software architecture as a combination of design solutions makes it an optimization problem — what is the best way of combining the solutions, with respect to quality requirements? Search-based software engineering (SBSE) studies the application of meta-heuristic algorithms to such software engineering problems [9]. In this field, genetic algorithms (GAs) [22] have been shown to be a feasible method for producing software architectures from functional requirements [29, 33, 34]. However, experiments with asexual reproduction [30] suggest that the crossover operator which is an essential part of GAs might not be critical for producing good architectures, supporting the idea of using a simpler search method. Additionally, the GA easily becomes very slow if the system is large, or if the search leans heavily towards certain mutations (due to preferences of the architect). These heavy mutations combined with a large system meant that the GA, which has to handle an entire population of solutions simultaneously, had to deal with a massive amount of data. It is, thus, natural to ask if other (lighter) search methods are capable of producing equally good architectures alone or in co-operation with genetic algorithms. The purpose of the present paper is to study the possibilities of simulated annealing (SA) in the process of searching good architectures when functional requirements are given.

While GA is already shown to produce reasonable software architectures, it is of great interest to study whether SA is capable to do the same, as it explores the search space in a completely different way than GA. An affirmative answer

would, of course, give us a new competitive practical method for producing software architectures. Contrary to GAs, SA is a local search method which intensively uses the concept of neighborhood, i.e., the set of possible solutions that are near to the current solution. The neighborhood is defined via transformations that change an element of the search space (here, software architecture) to another. In our application the transformations mean implementing a design pattern or an architectural style. Contrary to, for example, hill climbing algorithms, SA does allow also temporarily exploring worse solutions than what have been found so far. Due to the nature of the fitness landscape (many small peaks and large dips which lead to high peaks), this is essential in eventually finding a good architecture. Our decision to study SA first is also backed up by the promising studies in related fields where SA has been used for software refactoring [23]. Results from our studies conducted with SA will give us further information on what is required from the synthesis, and we may then possibly study other algorithms, such as particle swarm optimization and ant colony optimization.

It is known that seeding GAs enable them to produce better results faster [17, 36]. Our hypothesis is that a SA algorithm could also be used to quickly produce a seed. An initial population can be generated based on this seed. A significantly smaller number of generations would then suffice to find good solutions with the GA.

As with our GA approach, we begin with the functional requirements of a given system. The actual architecture is achieved by the SA algorithm, which gradually transforms the system by adding (and removing) design patterns and applying architecture styles. The resulting architecture is evaluated from three (contradicting) viewpoints: modifiability, efficiency and complexity. As the SA is implemented as close to the GA as possible, our set of research questions thus becomes: How good are the architectures produced by SA (compared to GA)? What kind of fitness values does SA achieve (compared to GA)? How fast is SA (compared to GA)? And finally, how well does a seeded algorithm perform in terms of both quality and speed (compared to GA)?

This paper proceeds as follows. In Section 2 we sketch current research in the field of search algorithms in software design that is relevant for the present paper. In Section 3 we cover the basics of implementing a SA algorithm and give the algorithmic presentation for our GA, to be used in the experiments. In Section 4 we introduce our method by defining the input for the SA algorithm, the transformations and the evaluation function. In Section 5 we present the results from our experiments, as we examine different parameters for the SA and combining SA with our GA implementation. In Section 6 we discuss the findings and in Section 7 we give a conclusion of our results.

2 Related Work

SBSE considers software related topics as combinatorial search problems. Traditionally, testing has been the clearly most studied area inside SBSE [13]. Other

well studied areas include software clustering and refactoring [9, 13, 26]. Using meta-heuristic algorithms in the area of software design, and in particular at software architecture design, is quite a novel idea. Only a few studies have been published where the algorithm actually attempts to design something new, rather than re-designing an existing software system. Approaches dealing with higher level structural units, such as patterns, have also recently gained more interest. We will briefly discuss the studies with the closest relation to our approach. As our method in part combines two algorithms, and the result can be viewed as a seeded algorithm (either SA provides a seed for the GA or vice versa), we will also briefly discuss approaches using seeding.

Amoui et al. [2] use the GA approach to improve the reusability of software by applying architecture design patterns to a UML model. Their goal is to find the best sequence of transformations, i.e., pattern implementations. Used patterns come from the collection presented by Gamma et al. [11]. From the software design perspective, the transformed designs of the best chromosomes are evolved so that abstract packages become more abstract and concrete packages, in turn, become more concrete. This approach uses one quality factor (reusability) only, while we use three quality factors, and also a more refined starting point than what is used in our approach.

Bowman et al. [7] study the use of a multi-objective genetic algorithm (MOGA) in solving the class responsibility assignment problem. The objective is to optimize the class structure of a system through the placement of methods and attributes within given constraints. So far, they do not demonstrate assigning methods and attributes "from scratch" (based on, e.g., use cases), but try to find out whether the presented MOGA can fix the structure if it has been modified. Thus, their approach currently works for refactoring only, and is not able to do forward design, which is our aim.

Simons et al. [44] study using evolutionary, multi-objective search and software agents to aid the software architect in class design. One individual (solution) is thus the design containing all methods and attributes (and their class distribution). Coupling and cohesion are used to calculate fitness. Simons et al. suggest that a global multi-objective search is unnecessary, and the search should be narrowed towards the "most useful and interesting candidate designs". They attempt to achieve this by isolating discrete zones from the search space, and then using a local search within these zones. Local search is conducted using a single-objective genetic algorithm, which only considers coupling in the fitness calculations. The designer then obtains the results of these local searches. Simons and Parmee [43] have further enhanced their studies with elegance metrics, which should conform to the desire for symmetry that human designers have.

Räihä et al. [29] have taken the design of software architecture a step further than Simons and Parmee [40, 41] by starting the design from a responsibility dependency graph. The dependency graph can also be achieved from use cases, but the architecture is developed further than the class distribution of actions and data. A GA is used for the automation of design. Mutations are implemented as adding or removing an architectural design pattern [11] or an interface or splitting or join-

ing class(es). Implemented design patterns are Façade and Strategy, as well as the message dispatcher architecture style [39].

Räihä et al. [34] have also applied GAs in model transformations that can be understood as pattern-based refinements. In MDA, such transformations can be exploited for deriving a Platform Independent Model from a Computationally Independent Model. The approach uses design patterns as the basis of mutations and exploits various quality metrics for deriving a fitness function. They give a genetic representation of models and propose transformations for them. The results suggest that GAs provide a feasible vehicle for model transformations, leading to convergent and reasonably fast transformation process. Räihä et al. [31] have also later on added scenarios, which are common in real world architecture evaluations, to evaluate the fitness of their synthesized architectures. Our work differs from the work of Räihä et al. [31] by using simulated annealing in addition to GA.

Jensen and Cheng [15] present an approach based on genetic programming (GP) for generating refactoring strategies that introduce design patterns. They have implemented a tool, RE-MODEL, which takes as input a UML class diagram representing the system under design. The system is refactored by applying mini-transformations. The encoding is made in tree form (suitable for GP), where each node is a transformation. A sequence of mini-transformations can produce a design pattern; a subset of the patterns specified by Gamma et al. [11] is used to identify desirable mini-transformation sequences. Mutations are applied by simply changing one node (transformation), and crossover is applied as exchanging subtrees. The QMOOD [4] metrics suite is used for fitness calculations. In addition to the QMOOD metrics, the authors also give a penalty based on the number of used mini-transformations and reward the existence of (any) design patterns. The output consists of a refactored software design as well as the set of steps to transform the original design into the refactored design. This way the refactoring can be done either automatically or manually; this decision is left for the software engineer. This approach is close to those of Räihä et al. [29] and the approach used here, the difference being that Jensen and Cheng have clearly a refactoring point of view, while we attempt upstream synthesis, thus expecting less from the architect and relying more on the algorithm, which makes our problem setting far more complex. Our fitness metrics are also different, as we only reward patterns that clearly improve the design — the simple existence of a pattern is not a reason for reward itself.

A higher level approach is studied by Aleti et al. [1], who use AADL models as a basis, and attempt to optimize the architecture with respect to Data Transfer Reliability and Communication Overhead. They use a GA and a Pareto optimal fitness function in their ArcheOptrix tool, but they concentrate on the optimal deployment of software components to a given hardware platform rather than how the components are actually constructed and how they communicate with one another. Research has also been made on identifying concept boundaries and thus automating software comprehension [12] and re-packaging software [5], which can be seen as finding working subsets of an existing architecture. These approaches are, however, already pushing the boundaries of the concept "software architecture

design". As for different aspects on GAs, the role of crossover operations in genetic synthesis of software architectures is studied by Räihä et al. [30, 32].

SA has been used in the field of search-based software engineering for software refactoring [23, 24, 25] and quality prediction [6]. O'Keeffe and Ó Cinnéide [23, 24, 25] work on the class level and use SA to refactor the class hierarchy and move methods in order to increase the quality of software. Their goal is to minimize unused, duplicated and rejected methods and unused classes, and to maximize abstract classes. The algorithm operates with pure source code, and the outcome is given as refactored code as well as a design improvement report. This approach is the closest to the one presented here, but it operates on a lower level and backwards (re-engineering), while our approach operates on a higher level architecture and goes forwards in the design process. Similar studies (class level refactoring) have also been made by Seng et al. [37, 38] who use GA as their search algorithm and Harman and Tratt [14], who use hill climbing. In the area of quality prediction, Bouktif et al. [6] attempt to reuse and adapt quality predictive models, each of which is viewed as a set of expertise parts. The search then aims to find the best subset of expertise parts, which forms a model with an optimal predictive accuracy.

In UML software design SA has been used in the context of dynamic parameter control in interactive local search by Simons and Parmee [42]. The level of design is quite similar, as it also deals with classes, methods and attributes. In this study the approach using simulated annealing was shown to be inferior to other method used in parameter control, while dynamic parameter control in general proved to be an efficient way for improving the results. Our approach differs significantly from that of Simons and Parmee, as we use simulated annealing itself in a different way (as the actual search algorithm itself, as opposed to controlling the parameters). We also have a very different mutation setting and problem domain. We have sixteen mutations, while there were only a couple in the presented study, thus the setting for dynamically controlling all the probabilities is much more complex, though we acknowledge the idea worth pursuing (initial experiments with a similar idea have been done in our previous work [34]). All in all, the studies using SA are few, and none use this approach for such a high-level design problem as designing software architecture from requirements

Our approach of combining SA and GA can be seen as a seeded algorithm, as one algorithm provides a developed seed for the other. Julstrom [17] has used the idea of seeding the initial population of a GA with advanced individuals in the rectilinear Steiner problem. The seeded algorithm produced more consistent results and was significantly faster than the algorithm with a randomly created initial population. Ramsey and Greffenstett [36] have studied case-based initialization of GAs in learning systems. In their study, the population of the GA is dynamically initialized with achieved (good) results, which aids in (intentionally) biasing the search towards a certain area, and quickly answering to a changing environment.

3 Simulated Annealing

Simulated annealing is a widely used optimization method for hard combinatorial problems. Principles behind the method were originally proposed by Metropolis et al. [20] and later Kirkpatrick et al. [18] generalized the idea for combinatorial optimization.

Algorithm 1 simulatedAnnealing

1: **Input:** Responsibility dependency graph G, base architecture M, initial temperature t_0, frozen temperature t_f, cooling ratio α, and temperature constant r
2: **Output:** UML class diagram D
3: $initialSolution \leftarrow \text{encode}(G, M)$
4: $initialQuality \leftarrow \text{evaluate}(initialSolution)$
5: $S_1 \leftarrow initialSolution$
6: $Q_1 \leftarrow initialQuality$
7: $t \leftarrow t_0$
8: **while** $t > t_f$ **do**
9: $\quad r_i \leftarrow 0$
10: $\quad$ **while** $r_i < r$ **do**
11: $\quad\quad S_i \leftarrow \text{transform}(S_1)$
12: $\quad\quad Q_i \leftarrow \text{evaluate}(S_i)$
13: $\quad\quad$ **if** $Q_i > Q_1$ **then**
14: $\quad\quad\quad S_1 \leftarrow S_i$
15: $\quad\quad\quad Q_1 \leftarrow Q_i$
16: $\quad\quad$ **else**
17: $\quad\quad\quad \delta \leftarrow Q_1 - Q_i$
18: $\quad\quad\quad p \leftarrow \text{UniformProbability}$
19: $\quad\quad\quad$ **if** $p < e^{\frac{-\delta}{t}}$ **then**
20: $\quad\quad\quad\quad S_1 \leftarrow S_i$
21: $\quad\quad\quad\quad Q_1 \leftarrow Q_i$
22: $\quad\quad\quad$ **end if**
23: $\quad\quad$ **end if**
24: $\quad\quad r_i \leftarrow r_i + 1$
25: $\quad$ **end while**
26: $\quad t \leftarrow (1 - \alpha) \times t$
27: **end while**
28: $D \leftarrow \text{generateUML}(S_1)$
29: **return** D

The SA algorithm starts from an initial solution which is enhanced during the annealing process by searching and selecting other solutions from the neighborhood of the current solution. There are several parameters that guide the annealing. The search begins with initial temperature t_0 and ends when temperature t is decreased to the frozen temperature t_f, where $0 \leq t_f \leq t_0$. The temperature gives the

Algorithm 2 geneticAlgorithm

```
1: Input: formalization of solution, initialSolution
2: population ← createPopulation(initialSolution)
3: while NOT terminationCondition do
4:   for all chromosome in population do
5:     p ← randomProbability
6:     if p > mutationProbability then
7:       mutate(chromosome)
8:     end if
9:   end for
10:  for all chromosome in population do
11:    cp ← randomProbability
12:    if cp > crossoverProbability then
13:      addToParents(chromosome)
14:    end if
15:  end for
16:  for all chromosome in parents do
17:    father ← chromosome
18:    mother ← selectNextChromosome(parents)
19:    offspring ← crossover(father, mother)
20:    addToPopulation( offspring)
21:    removeFromParents (father, mother)
22:  end for
23:  for all chromosome in population do
24:    calculatefitness(chromosome)
25:  end for
26:  selectNextPopulation()
27: end while
```

probability of choosing solutions that are worse than the current solution. The result of a transformation that worsens the current solution by d, is accepted to be the new current solution if a randomly generated real i is less than or equal to a limit which depends on the current temperature t. If a transformation improves the current solution, it is accepted directly without a test.

An important parameter of SA is the cooling schedule, i.e., how the temperature is decreased. We use the geometric cooling schedule, in which a constant r is used to determine when the temperature is decreased, and the next temperature is obtained simply by multiplying the current temperature by cooling ratio a ($0 < a < 1$). This is the most frequently used schedule [45]. It was chosen because of its simplicity, and because of the fact that all the classical cooling schedules can be tuned so that they give the same practical temperatures [45].

The SA has been successfully applied for numerous combinatorial optimization problems, for an instructive introduction to the use of SA as a tool for experimental algorithmics, see [3, 16]. In order to determine good parameters for a problem,

experimental analysis is often needed. There are also adaptive techniques for detecting the parameters [19]. The SA implementation used in our tests is shown in Algorithm 1. The encoding, transformation and evaluation procedures are discussed in more detail in Section 4. Notice, that our SA only operates with a single solution at a time, and the solution is built by transformations (i.e., moving towards better neighbors).

In Section 5 we compare the present SA and our previous GA implementation [29]. We assume the reader has knowledge of the basic principles of GA, as given by, e.g., Michalewicz [21]. The GA implementation used is given in Algorithm 2. Mutation is executed in the same way as a transformation for simulated annealing, and more details will be given in Section 4. Crossover is a single-point random crossover and selecting the next population is made with a rank-based roulette wheel selection. For more details on how crossover and selection is implemented in our approach, we refer to [27].

The result of GA is the best solution found during the search process. Thus, in that sense, both SA and GA are single solution algorithms and their comparison is straightforward. In order to be able to fairly compare the implementations, the solutions produced by the two methods should be evaluated by the same quality functions and the initial solutions should be of the same quality. Hence, we use the same method for producing the initial solutions for SA as we have done with GA in [29, 31, 33, 34]. The initial solution is achieved by encoding functional requirements and thus building a base architecture. The base class structure is derived from the base architecture, and the base architecture is achieved by randomly applying a transformation. The same approach for creating several solutions for an initial population is used in our GA implementation [29, 31, 33, 34], and thus the initial quality is the same for both SA and GA, as they both use the same evaluation function.

4 Method

We begin by creating use cases to define the basic functional requirements. Use cases are an intuitive starting point in most software projects, and little domain knowledge is required to define them. Thus, use cases are a natural way to begin eliciting the functional requirements of a system. Use cases can, in turn, be refined into sequence diagrams. The refining process requires some effort from the architect but still quite little domain knowledge and is still fairly intuitive, as the architect simply needs to think how different use cases proceed on operation level. From sequence diagrams it is simple to elicit classes (the participants/owners' of lifelines) and operations (calls in the diagram). This results in a base architecture, giving a structural view of the functional requirements of the system at hand but not dealing with the quality requirements. The base architecture is encoded to a form that can be processed by the search algorithm in question. The algorithm produces software architecture for the given quality requirements by implementing selected architecture styles and design patterns, and produces a UML class diagram as the

result.

4.1 Requirements

We will use two example systems: the control system for a computerized home, called hereafter ehome, and a robot war game simulator, called robo. We will demonstrate building input for the search algorithm in the case of ehome; the process is similar in the case of robo.

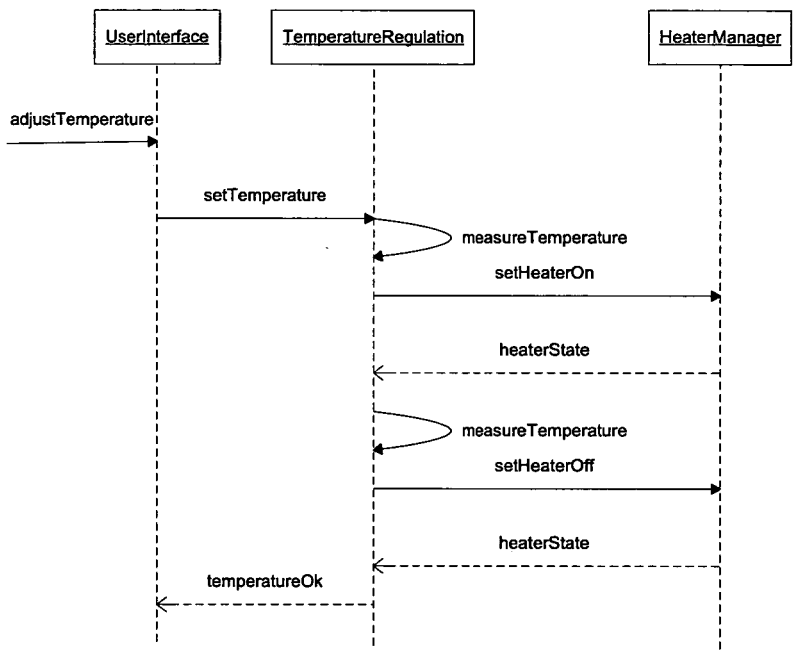

Figure 1: Adjust temperature use case refined

Specifying requirements begins with giving use cases. Use cases for the ehome system are assumed to consist of, e.g., logging in, changing the room temperature, changing the unit of temperature, making coffee, moving drapes, and playing music. Here, we will take as an example the adjust room temperature use case. The user simply places a command that the temperature should be adjusted (for the sake of simplicity, we can here consider elevation), and ehome adjusts the temperature by turning on the heater.

The sequence diagram for the temperature adjustment use case is given in Figure 1. The process begins with a call from the user to set the temperature to a new level. The system then calls the temperature regulation component, which measures the current temperature, and then sets the heater on. After the correct temperature is reached, the heater is turned off.

While sequence diagrams already give a good understanding of how the different operations depend on each other, a structural view still needs to be obtained, as patterns cannot be inserted into sequences of calls. Fortunately, sequence diagrams can easily be turned into class diagrams. At this point, the class diagram would not consist of anything but the classes, their methods and attributes, and connections between classes, as defined in the sequence diagram. We have chosen to use sequence diagrams as the basis, as they can be straightforwardly build based on use case diagrams, and use case diagrams are the most intuitive way to start formulating the requirements.

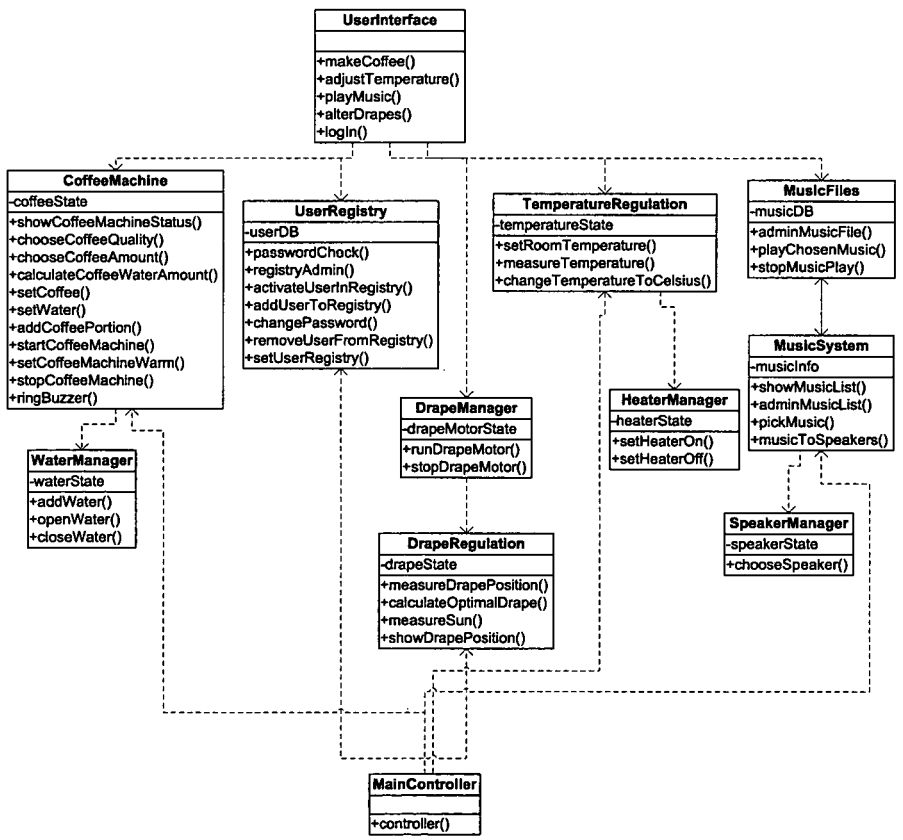

Figure 2: Base architecture for ehome

The base architecture in Figure 2 for the ehome system can be straightforwardly derived from the sequence diagrams. We depict architecture as a class diagram, as we consider the architecture to be the classes or components of a system the interfaces and other communication mechanisms between them. Thus, a class view is natural for our purposes.

The messages in the sequence diagram become the operations and the objects/components become the classes. Also, if the need for a data source is detected or the object clearly has a state, they will become attributes in the classes. For example, in Figure 1 both the `Temperature Regulation` and `Heater Manager` have states, such as on or off for the `Heater Manager`. The base architecture only contains use relationships, as no more detail is given for the algorithm at this point. The base architecture represents the basic functional decomposition of the system. A base architecture for robo (which can be achieved by performing the same steps as did with ehome) is given in Figure 3.

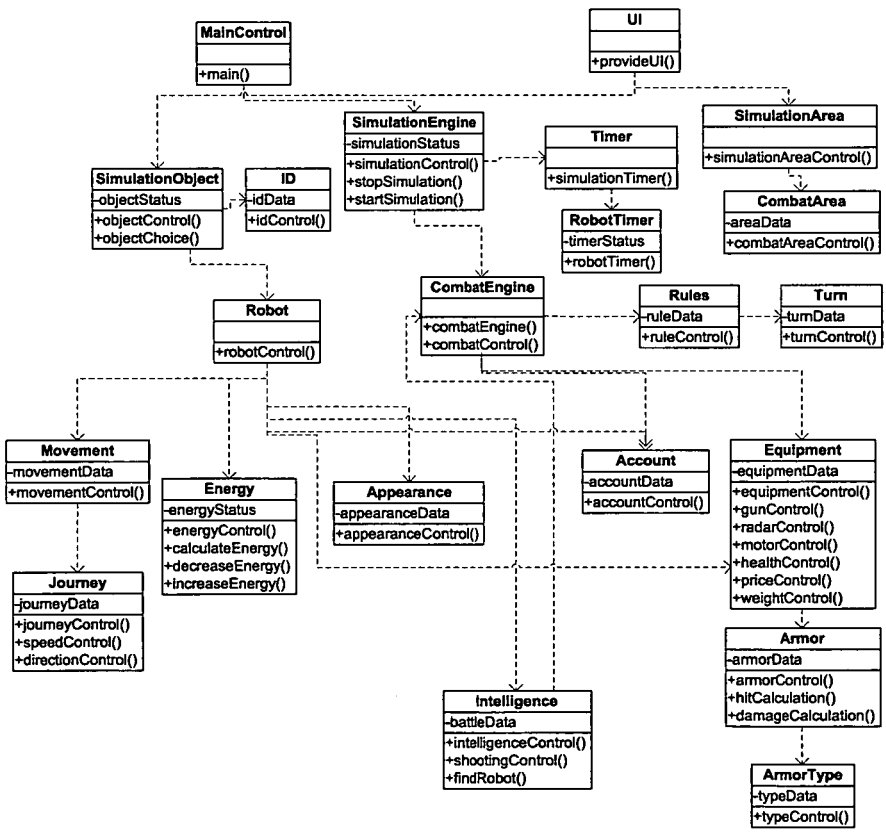

Figure 3: Base architecture for robo

After the operations are derived from the use cases, some properties of the operations can be estimated to support the synthesis, regarding the amount of data an operation needs, frequency of calls, and sensitiveness for variation. For example, it is likely that the coffee machine status can be shown in several different ways, and thus it is more sensitive to variation (called hereafter the variability of an operation) than ringing the buzzer when the coffee is done. Measuring the position of drapes requires more information than running the drape motor (which can be interpreted as the required parameter size), and playing music quite likely has a higher usage frequency than changing the password for the system. Relative values for the chosen properties can similarly be estimated for all operations. Here we have used the scale of Low (1), Medium (3) and High (5). This optional information, together with operation call dependencies, is included in the information subjected to encoding.

4.2 Encoding

Ultimately, there are two kinds of data regarding each operation o_i. Firstly, there is the basic information given as input. This contains the operations $O_i = \{o_{i1}, o_{i2}, \ldots, o_{ik}\}$ depending on o_i, its name n_i, type d_i ("f" as in functional for methods, "d" as in data for attributes), frequency f_i, parameter size p_i and variability v_i. Secondly, there is the information regarding o_i's place in the architecture: the class(es) $C_i = \{C_{i1}, C_{i2}, \ldots, C_{iv}\}$ it belongs to, the interface I_i it implements, the dispatcher D_i it uses, the operations $OD_i \subseteq (O_i)$ that call it through the dispatcher, the design patterns $P_i = \{P_{i1}, P_{i2}, \ldots, P_{im}\}$ it is a part of, and the pre-determined base architecture class MC_i. The dispatcher is given a separate field as opposed to other patterns for efficiency reasons.

The base architecture is encoded as a vector $V < ov_1, ov_2, \ldots, ov_n >$ of vectors $ov_1, ov_2, \ldots, ov_n$ for the algorithm. Each vector ov_k, in turn, contains all data for a single operation. Thus, n is the number of operations of a system, and the collection of these operation defining vectors depicts the entire system when collected into one vector V. Figure 4 depicts an operation vector ov_i. The same encoding works for both SA and GA. For GA, the chromosome is the vector V, and each vector ov_i is a supergene, which contains the fields described above.

O_i	n_i	d_i	f_i	p_i	v_i	C_i	I_i	D_i	OD_i	MC_i	P_i

Figure 4: Operation vector ov_i

We will give an example from the ehome system of how the given data structure works. In the base architecture phase, if the `TemperatureRegulation` class is given #ID 2 (and the interface #ID 2), for operation `measureTemperature` (#id 9) the ov_9 would have the following values: O_9 = {#idSetRoomTemperature}, n_9 = measureTemperature, d_9 = f, $p_9 = 3$, $f_9 = 1$, $v_9 = 3$, $C_9 = 2$, $I_9 = 0$, $D_9 = 0$, $OD_9 = \emptyset$, $MC_9 = 2$, $P_9 = \emptyset$. The interface has value 0, as `measureTemperature` is only required by `setRoomTemperature`, which is in the same class, and thus

does need an interface to access this operation. The fields for message dispatcher and pattern have $\emptyset$ values, as no architectural solutions are included in the base architecture. As the operation is here located in the original base architecture class, the values for C and MC are the same. Note, that the encoding is indeed operation-centered. Thus, modifications to the architecture are considered from the viewpoint of how a particular operation can be accessed, and not how two classes communicate with each other. In practice, the base architecture is encoded into a text file, which is given as input for the algorithm, with each operation in its own line.

4.3 Transformations

An architecture is transformed (i.e., one of its neighbors is found) by implementing architecture styles and design patterns to a given solution. The patterns we have chosen include very high-level architectural styles [39] (message dispatcher and client-server), medium-level design patterns [11] (`Façade` and `Mediator`), and low-level design patterns [11] (`Strategy`, `Adapter` and `Template Method`). This selection of patterns and styles allows us to see how well the algorithm handles different types of changes. High-level patterns have a larger impact, as they usually affect large parts of the architecture, while lower level patterns only affect small parts. The transformations are implemented in pairs of introducing a pattern or removing a pattern. This ensures a wider traverse through the search space, as while implementing a pattern might improve the quality of architecture at one point, it might become redundant over the course of development. The dispatcher architecture style makes a small exception to this rule: the actual dispatcher must first be introduced to the system, after which the responsibilities can communicate through it. The transformations are the following, and each of them has a certain probability with which it is selected:

- introduce/remove message dispatcher
- communicate/remove communication through dispatcher
- introduce/remove server
- introduce/remove `Façade`
- introduce/remove `Mediator`
- introduce/remove `Strategy`
- introduce/remove `Adapter`
- introduce/remove `Template Method`.

The legality of applying a pattern is always checked before transformations by giving pre-conditions. For example, the structure of the `Template Method` demands that depending operations are in the same class. In addition, a corrective

function is added to check that the solution conforms to certain architectural laws, and that no anomalies are brought to the architecture. These laws demand uniform calls between two classes (e.g., through an interface or a dispatcher but not both), and state some basic rules regarding architectures (e.g., no operation can implement more than one interface). The corrective function, for example, discards interfaces that are not used by any class, and adds dispatcher connections between operations in two classes, if such a connection already exists between some operations in those classes. For example, if the "add `Strategy`" transformation is chosen, it is checked that the operation o_i is called by some other operation in the same class c and that it is not a part of another pattern already (pattern field is empty). Then, a `Strategy` pattern instance sp_i is created. It contains information of the new class(es) sc_i where the different versions of the operation are placed, and the common interface si_i they implement. It also contains information of all the classes and operations that are dependent on o_i, and thus use the `Strategy` interface. Then, the value in the class field in the vector ov_i (representing o_i) would be changed from c to sc_i, the interface field would be given value si_i and the pattern field the value sp_i. Adding other patterns is done similarly. Removing a pattern is done in reverse: the operation placed in a pattern class would be returned to its original base architecture class, and the pattern found in the supergenes pattern field would be deleted, as well as any classes and interfaces related to it.

4.4 Quality Function

In the case of software architecture design, selecting an appropriate evaluation function is particularly difficult, as there is no clear value to measure in the solutions. In real world, evaluation of software architecture is almost always done manually by human designers, and metric calculations are only used as guidelines. Also, two architects rarely agree on a unique quality for certain architecture, as evaluation is bound to be subjective, and different values and backgrounds will influence the outcome of any evaluation process. However, for a search algorithm to be able to evaluate the architecture, a purely numerical quality value must be calculated.

In a fully automated approach, no human interception is allowed, and the evaluation function needs to be based on metrics. The selection of metrics may be as arguable as the evaluations of two architects on a single software architecture. The rationale behind the selected metrics in this approach is that they have been widely used and recognized to accurately measure some quality aspects of software architecture. Hence, the metrics are chosen so that they measure quality aspects that can be seen as most agreed upon in the real world, and singular values can be seen as accurate as possible. However, the combination of metrics and multiple optimization is another problem entirely. For many metrics, it may be arguable what quality attribute they measure, and may be seen as measurements for several different quality attributes. Many of these quality attributes, however, are controversial. A perfect example is the selected quality attribute pair: modifiability and efficiency. The problem of multiple optimization is a direct result of the contradictive aims of the two quality attributes: when attempting to optimize one, the

quality will decrease in view of the other. In our GA approach we have implemented Pareto optimality [33] to conquer this problem. However, when evaluating the applicability of simulated annealing, we found it more practical to use a single weighted fitness, as we wanted to maintain SA as "pure" as possible (local and efficient), even though there are multi-objective versions of SA as well (e.g., [46]).

The chosen quality function is based on well-known software metrics [8]. These metrics, especially coupling and cohesion, have been used as a starting point for the quality function, and have been further developed and grouped to achieve clear sub-functions for modifiability and efficiency, both of which are measured with a positive and negative metric. The biggest modifications to the basic metrics include taking into account the positive effect of interfaces and the dispatcher and client-server architecture styles in terms of modifiability, as well as the negative effect of the dispatcher and server in terms of efficiency. Choosing and grouping the metrics this way makes sure that all architectural decisions are always considered from all viewpoints. Adding a pattern always adds a class or an interface (or both), and is thus considered by complexity. As the calls to an operation are also affected, the change is always also considered positive or negative by both modifiability and efficiency.

Dividing the evaluation function into sub-functions also answers the demands of the real world. Hardly any architecture can be optimized from all quality viewpoints, but some viewpoints are ranked higher than others, depending on the demands regarding the architecture. By separating efficiency and modifiability, which are especially difficult to optimize simultaneously, we can assign a bigger weight to the more desired quality aspect, if we want to. When w_i is the weight for the respective sub-function sf_i, the evaluation function $f_c(x)$ (which should be maximized) for solution x can be expressed as

$$f_c(x) = w_1 \times sf_1 - w_2 \times sf_2 + w_3 \times sf_3 - w_4 \times sf_4 - w_5 \times sf_5. \quad (1)$$

Here, sf_1 measures positive modifiability, sf_2 negative modifiability, sf_3 positive efficiency, sf_4 negative efficiency and finally sf_5 measures complexity. All the sub-functions are normalized so that they have the same range. The sub-functions are defined as follows ($|X|$ denotes the cardinality of X):

$$sf_1 = |\textit{calls to interfaces}| \times \sum_{k=0}^{i}(v_k) + |\textit{calls through dispatcher}|) \times \sum_{k=0}^{d}(v_k),$$

$$sf_2 = |\textit{direct calls between operations in different classes}| \times \sum_{k=0}^{c}(v_k))$$
$$+ |\textit{calls between operations within same class}| \times \sum_{k=0}^{s}(v_k) \times 2,$$

$$sf_3 = |\textit{operations dependent of each other within same class}| \times \sum_{k=0}^{w}(p_k) + |\textit{used operations in same class}| \times \sum_{k=0}^{u}(p_k) + |\textit{depending operations in same class}| \times \sum_{k=0}^{e}(p_k),$$

$$sf_4 = \sum |\textit{ClassInstabilities}| + (2 \times |\textit{dispatcherCalls}| + |\textit{serverCalls}|) \times \sum_{k=0}^{ds}(f_k) + |\textit{calls between operations in different classes}|, \textit{and}$$

$$sf_5 = |\textit{classes}| + |\textit{interfaces}|.$$

In sf_1, i is the number of operations called through an interface, d is the number of operations called through dispatcher, and v is the variability value of an operation (as in Fig. 4). The variability values v of those operations that are involved in interface or dispatcher calls, respectively, are summed. In sf_2, c is the number of calls from a different class to an operation with no interface and with variability value v_k, sc is the number calls from within the same class to an operation with variability value v_k. The calculation is similar to that in sf_1, as variability values of operations are summed if said operations are called based on given criteria. Calls within class are given a constant multiplier 2, as it is considered that a call within class bonds two operations and thus has double the negative effect on modifiability. The w, u and e in sf_3 are the numbers of the types of calls as specified in sf_3, similarly as in sf_1 and sf_2. In sf_3, however, the parameter size values p are summed instead of variability values. It should also be noted, that in sf_1, most patterns also contain an interface. In sf_3, "used operations in same class" means a set of operations in class C, which are all used by the same operation from class D. Similarly, "depending operations in same class" mean a set of operations in class K, which all use the same operations in class L. In sf_4, ds is the number of calls through dispatcher or server where the called operation's frequency value is f_k. The multiplier 2 for calls relayed by the message dispatcher is given as there are always two calls when the message dispatcher is used - one from the calling class to the dispatcher and one from the dispatcher to the receiving class.

5 Experiments

In this section we present the results from the preliminary experiments done with our approach. Tests were made using the ehome and robo example systems (introduced before). The selected two systems are very different in nature and structure,

which would lead to very different architectures. Choosing these two different systems shows that the algorithm is not confined to any particular system, but can be generally used for any type of system. Most of the parameters used in our tests originate from the previous tests reported in [29, 31, 34], and give promising results with the GA approach. The implementation was made with Java 1.5. The tests were run on a DELL laptop with 2,95 GB of RAM and 2,26 GHz processor, running with Windows XP.

All tests were made with the constant r set to 20, and frozen (final) temperature t_f set to 1. The weights for all sub-functions of the quality evaluation function were set to the same, i.e., all weights w_i were set to 1, as we did not want to favor any particular quality attribute over another, but aimed for balanced designs. Also, by setting the weights to 1 we do not need to consider the effect of the weights in fitness curves.

The GA used in the combination experiments is based on our previous implementations [29, 31, 34]. The GA uses the same encoding, transformations (mutations) and quality function as defined here for the SA. As stated in Section 3, the crossover operator is a single-point random crossover and selection is made with a rank-based roulette wheel method. As this paper concentrates on simulated annealing, the particularities of the GA implementation are not discussed further here; details can be found in [29, 31, 34, 27].

5.1 Using SA First

The standard tests were made with 7500 as starting temperature and 0.05 as cooling ratio. A longer annealing was also experimented with by setting the starting temperature to 10 000 (cooling ratio 0.05), and a faster annealing was tested by setting the cooling ratio to 0.15 (starting temperature 7500). A lower starting temperature had also been tested previously with no obvious benefits [35]. The values were selected based on trial-and-error experiments. However, the results were unsatisfactory for both systems, and there were no significant differences between the results achieved with different SA parameters. The trend of the quality curve for the SA was descending, and the end quality value was worse than the initial value (the initial value is the same as where the GA starts in the curves given in the following section). The high temperature tests for both systems took approximately 10 seconds per run and the fast annealing tests less than 5 seconds per run, standard test runtime is reported in the following. We then tried to build a base solution with a short and fast annealing (starting temperature 2500 and cooling ratio 0.15), and then continue the search with a genetic algorithm, which ran for 250 generations and had a population of 100 (combination SAGA). This approach did not produce much better results: the SA curves were quite similar than with longer and slower runs, and while the quality curve for the GA portion did increase for a short while, it began to quickly descend drastically. Also in this case the end quality value was worse than the value for the initial solution. Again, the runtime for the SAGA seeded algorithm is reported in the following when compared to other approached. All experiments were run for 20 times.

5.2 Using a Combination of GA and SA

As using the SA alone or for producing a seed did not produce good results, we tried using GA for creating a good base solution (again, with 250 generations and a population of 100), and then applying SA (starting temperature 2500, cooling ratio 0.15) for further tuning the solution (combination GASA). The experiments were run for 20 times and presented fitness curves are the average curves of the 20 runs. We have chosen to show average curves, as we are, after all, interested in how the algorithms behave in general, not individual runs. In this case, the results were much better. The GA does a good basic work, and the SA is able to further improve the solution very quickly.

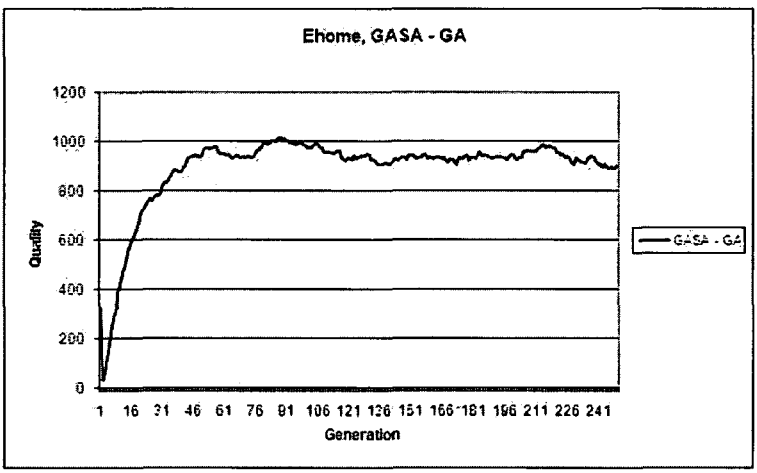

Figure 5: GA portion of GASA quality curve for ehome

Figure 5 presents the GA portion and Figure 6 the SA portion of the GASA quality curve for ehome. Figures 7 and 8 present the respective curves for robo. The GA curves represent the average of the elite (top 10% of the population) (given as an average over the 20 runs), and the SA curves are naturally simply the average of fitness values of the 20 runs. Note that the SA algorithm starts where the GA ends: the difference in the GA end value and SA start value is due to the fact that quality values are not recorded until one round of transformations has already been completed and because the GA curve is the average of elite, while SA handles only one solution.

As can be seen in Figure 5, the GA begins with a short plummet, after which the quality (fitness) begins to develop steadily. We expect the plummet to be an effect of using the message dispatcher very early on. When the message dispatcher is used sparingly (as is the case after only a few mutations), its penalty is greater than its reward. After about 100 generations the fitness appears to stabilize, i.e., the curve is not increasing, and it does not seem likely to further develop. In Figure 6, the SA begins to develop the solution from where the GA left off, and the curve

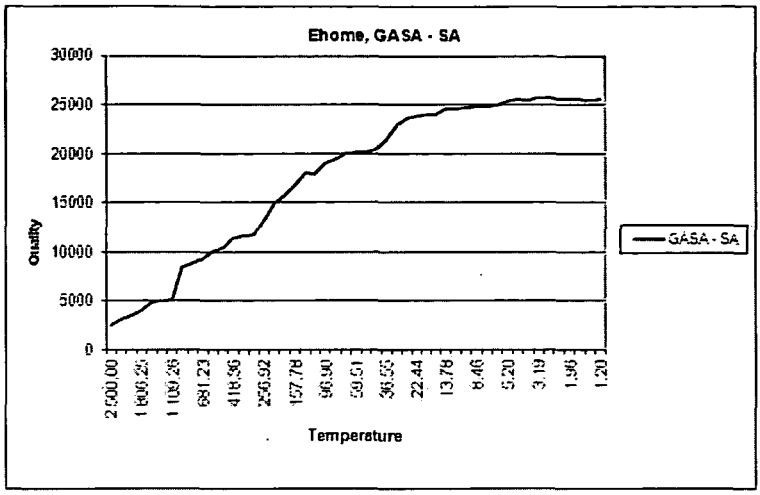

Figure 6: SA portion of GASA quality curve for ehome

develops rapidly until quite near the end of the SA process.

In Figure 7, depicting the GA portion for the robo system, the GA first plummets similarly as in the curve for ehome, but after it starts ascending, the development seems more rapid and steady than for the ehome, and it appears as if the quality could still increase after the GA finishes. The SA portion of the GASA curve for robo, in Figure 8, appears quite similar to the GA curve at first, but looking at the actual quality values reveals that the SA develops much more quickly than the GA. In the end the curve has reached a plateau, giving reason to believe that some optimum has been found.

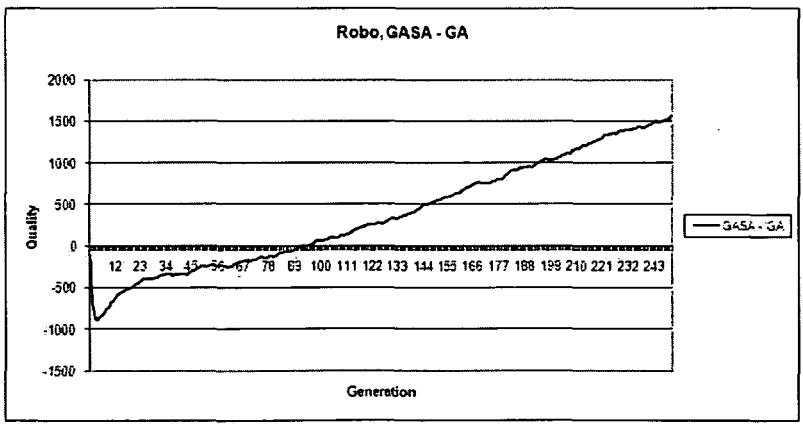

Figure 7: GA portion of GASA quality curve for robo

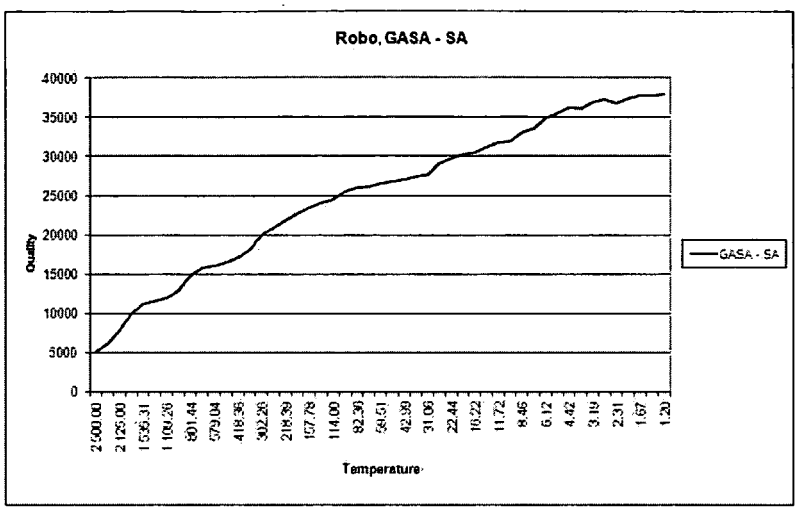

Figure 8: SA portion of GASA quality curve for robo

We have calculated the average fitnesses and standard deviations of GASA runs in Table 1. The average of the (averages of) elite is naturally the value where GA fitness curves end (Figures 5 and 7). The average of best (seed) is the average of the absolute best individuals provided by GA, which are then given as a seed for SA for further development. For SA we only have one value, as the algorithm only handles one individual at a time. From Table 1 we can see that the deviation especially in the case of GA is quite large, and the algorithm is not as stable as could be hoped. However, the deviation within the solutions after the SA (i.e., the final solutions from the seeded algorithm) is much smaller. There was no clear correlation between the elite fitness after GA and the fitness value after SA.

System		GA		SA
		Elite	Best (seed)	
Ehome	Average	898.7	2695.5	25536.8
	St. deviation	390.7	984.8	1233.6
Robo	Average	1558.6	4456.5	37921.8
	St. deviation	776.6	2127.3	7559.9

Table 1: Statistical markers

Finally, we have compared the runtimes of GA and SA and their combinations to random search (RS) and hill climbing (HC). The runtimes have been collected in Table 2. RS was run for 3500 iterations (same amount of iterations as SA with standard parameters) and HC was allowed 150 attemps of finding a neighbor after each ascent. All algorithms were run 20 times. The average fitness value

achieved with RS was -1915 for ehome and -4266 for robo. For HC, in 50% of the cases the algorithm only managed to ascend once, after which the algorithm terminated as 150 attempts at finding a better neighbor failed. In the rest of the cases HC was able to ascend 6-12 times. Due to such inconsistent results, average fitness values do not provide good information. However, HC is clearly not a suitable algorithm for this problem. As can be seen in Table 2, all local searches are significantly faster than GA or combinations with GA. It should be noticed, though, that the GA performs at least 25000 fitness evaluations (100 in a population times 250 generations and additionally the evaluations of offspring), while the SA only performs 3500 fitness evaluations (with the selected parameters). Also, crossover is a very time consuming operation for the GA. As a conclusion, GA and GASA are clearly the slowest algorithms, but produced just as clearly the best results.

	GA	SA	GASA	SAGA	HC	RS
Ehome	~45s	~5s	~100s	~50s	~6s	~4s
Robo	~35s	~6s	~40s	~40s	~2s	~4s

Table 2: Runtimes for different algorithms

6 Discussion

In Section 5 we discussed the quality curves of the experiments made with the SA algorithm. Naturally, the UML graphs given as output should also be examined to get a wholesome idea of whether the results with extreme quality values are actually good. In addition to discussing the class diagrams related to the test graphs presented in Section 5 (the GASA tests), we will also discuss the UML graphs achieved when SA was used primarily. The example solutions are given in a simplified format high-level where the design solutions are emphasized, rather than giving the actual class diagrams given by the algorithm, as they would be too space consuming and difficult to interpret. As the format is free form, we have not included class relations, but simple use relations only. There are no methods or attributes present in the solutions that were not there in the base architecture.

6.1 Proposed Architectures with GASA

Using the GASA approach produced very similar solutions for both ehome and robo systems. The solutions were built around the message dispatcher, as nearly all communication between classes (in different base architecture classes) was handled through it. The dispatcher makes the system highly modifiable, as classes do not need to know any details of other classes; they merely send and receive messages

through the dispatcher. The architecture is also easy to understand quickly, as the message dispatcher creates a logical center for the system and separates different model classes. However, the message dispatcher creates huge loss in efficiency, as the increased message traffic greatly affects the performance of the system. Thus, it should be used as the primary method for communication or not be used at all, as in the case where it is only partially used the cost in efficiency is bigger than the gain in modifiability.

In addition to the message dispatcher, all solutions achieved with the GASA approach had several instances of the `Adapter` pattern. The `Adapter` pattern is easy to apply, as it has very loose preconditions, but it is more costly in terms of efficiency than other patterns. There were usually also several instances of the `Template Method` pattern, which, in turn, is very low cost in terms of both efficiency (it does not increase the number of calls) and complexity (only one class, no interface). In some cases, however, the algorithm had preferred the `Strategy` pattern, and there would be many instances of `Strategy`, while only a few `Template Method` instances.

An example solution for ehome achieved with GASA is presented in Figure 9. As can be seen, nearly all connections are handled via the message dispatcher, as only calls from the `Main` component to `Music System` and `Coffee Machine`, and from `Music System` to `Music Files` are handled directly between the components. The example also shows that the `Template Method` is used very much to create low-level modifiability. The ehome is particularly suitable for a message dispatcher architecture style, and achieving a high level of message-based communication between components is desirable, as the message dispatcher is then used to its full potential and enables independency between components. The `Adapters` for `Water Control` and `Speaker Manager` are also particularly well placed, as these components are intuitively such that they could be replaced with new ones (in an ehome we may want to change the water faucet or upgrade to better speakers without changing the underlying kitchen or music systems), and thus the interface might change. The `Template Methods` for `Coffee Machine`, `Temperature Regulation` and `Music System` are also well chosen, as the specialized operations are such that alternative versions are easily conjured. Other `Adapters`, `Template Methods` and `Strategies` are acceptable, but a human designer would probably not apply them.

A similar example solution for robo (also achieved with GASA) is presented in Figure 10. As can be seen, the message dispatcher is used here even more intensely than in the case of ehome, as only connections between `CombatEngine` and `Rules` and some connections involving the `SimulationObject` are not using the message dispatcher, even though the amount of components is larger than in the case of ehome. However, while using the message dispatcher in these proportions is desirable if it is chosen as the primary architecture style, if we consider the type of system the robo is (a framework), in real life a message dispatcher would probably not be the best option. All the components are actually tightly linked, and the design should concentrate more on extendibility and the actual functionality of the system. Also, as robo is a gaming application, using the mes-

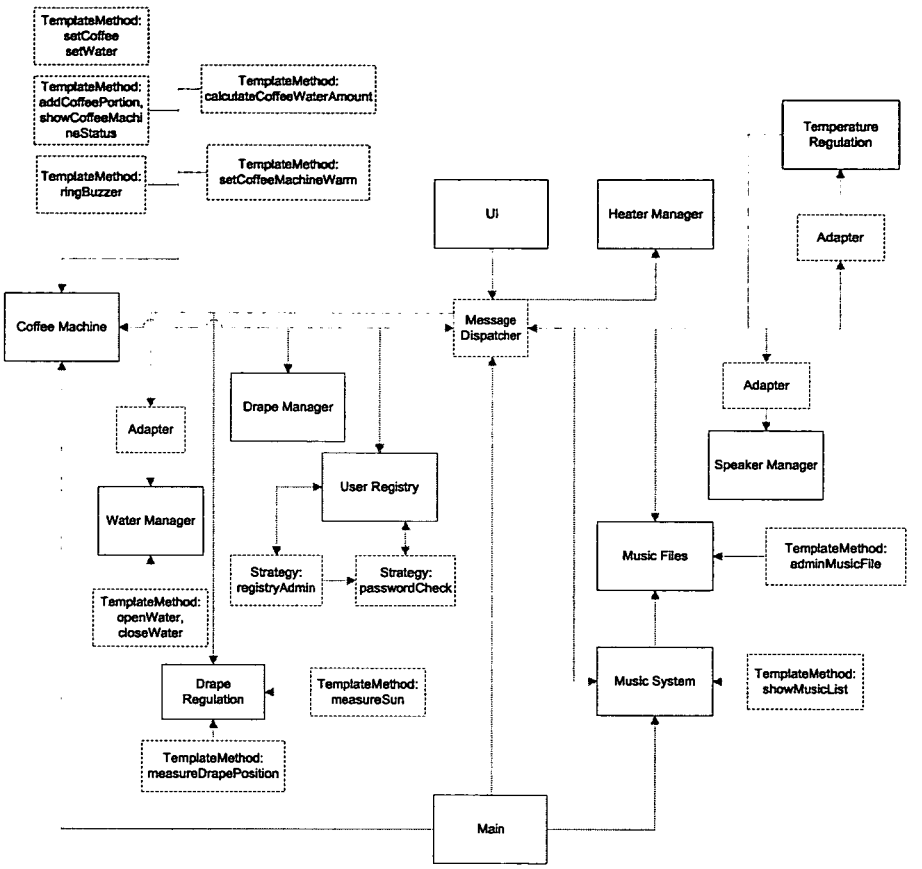

Figure 9: Example architecture for ehome, with GASA algorithm combination

sage dispatcher in this extent would probably lead to significant disadvantage in terms of efficiency, which is particularly undesirable when the system needs to respond quickly. The SA (or GA), however, does not have such high-level knowledge of the type of system it is dealing with and bases the design simply on the quality values, which are achieved from general structural decisions only. For robo, there are also several `Adapter`, `TemplateMethod` and `Strategy` patterns, and the usage of these different patterns is more balanced than in the case of ehome, where the `Template Method` was the dominating pattern. In the proposed solution for robo, the `Template Method` and `Strategy` patterns are all intelligently used, as they consider operations and classes where the need for specialization is easily

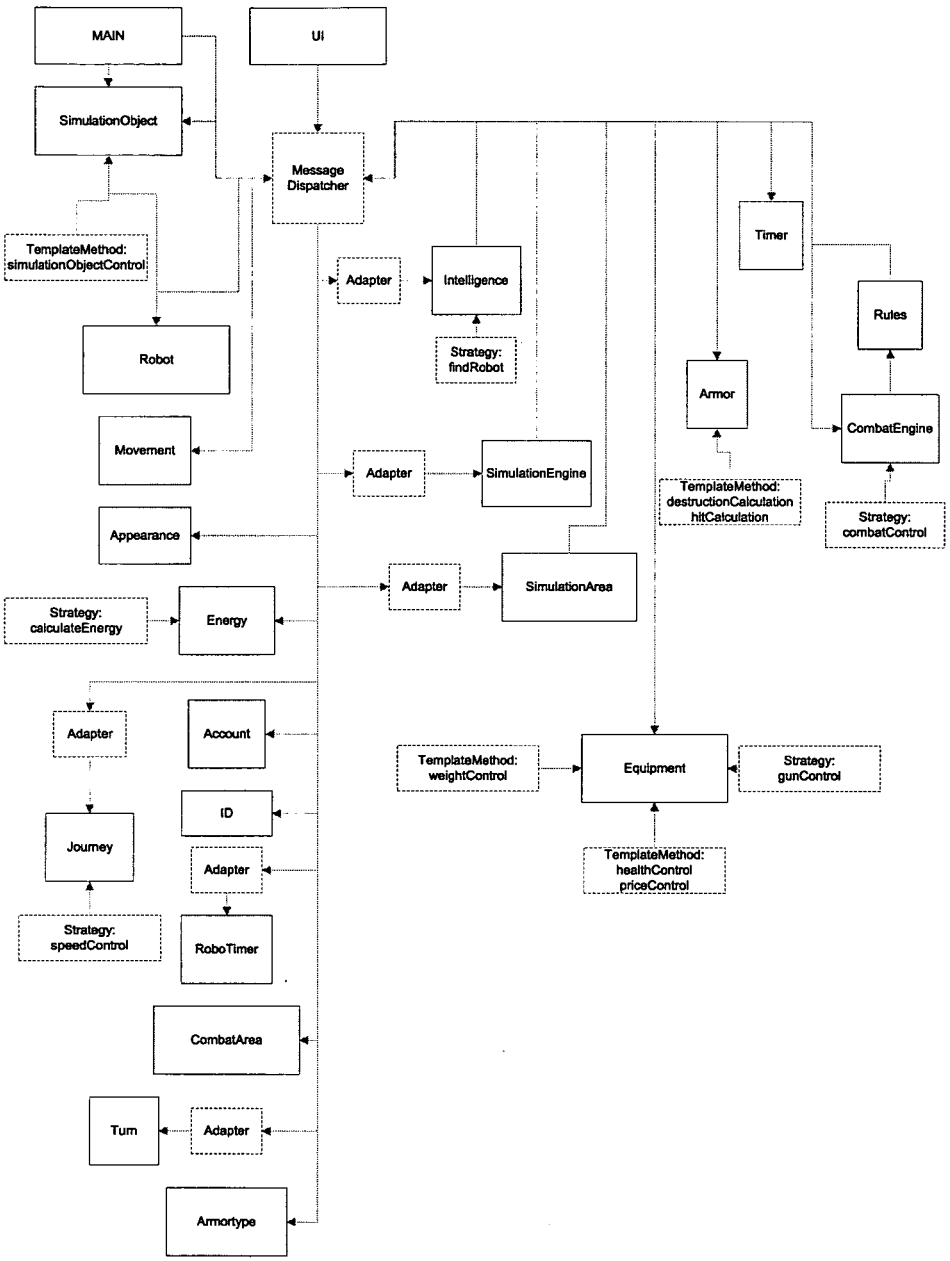

Figure 10: Example architecture for robo, with GASA algorithm combination

seen. The `Adapters`, however, are not used particularly well. To summarize, using the message dispatcher gives a clear focal point in the solutions, and the full potential of the message dispatcher is used. It should also be pointed out that solutions achieved after only running the GA (i.e., the seeds for the SA) often had the message dispatcher, but its usage was mostly quite minimal, as only a couple of components were communicating via the dispatcher. Thus, the SA algorithm has a significant influence in achieving a much better level of usage in the final solution. In addition, low-level design patterns are used to further fine-tune the solution at class-level.

6.2 Proposed Architectures Based on SA

As mentioned, we also performed tests with only SA and by combining SA to GA by using the SA produced solution as a seed for the GA. The produced solutions were very similar for all cases of the SA (high temperature, standard, and fast annealing) and the SAGA approach.

In these cases, the message dispatcher architecture style did not appear in any of the solutions for either system. As for the patterns, the Adapter pattern was clearly the most popular in all the solutions for both systems. For the robo system, there were very few instances of other patterns; only a couple of `Template Method` or `Strategy` patterns could be found in the solutions. The solutions for robo seemed quite difficult to understand at a glance; the structure depends greatly on the base architecture, and as all classes are "by default" given an interface, the minimum amount of classes/interfaces is 44 for the robo system. When the patterns are added (even if only a few) the architecture easily becomes quite complex. The solutions for ehome were significantly easier to understand, as the amount of classes/interfaces that appear by default is roughly half the amount of classes for robo system. Curiously enough, there seemed to also be slightly more appearances of the `Strategy` and `Template Method` patterns in the ehome solutions than in robo, but the ehome solutions still seemed more understandable.

It appears that the SA by itself is incapable of introducing solutions that produce delayed reward, such as the message dispatcher architecture style. Also, even if the GA is able to introduce such solutions after being given the seed from the SA, it will take exceptionally long before the reward will overcome the cost, as the SA has already developed the solutions a great deal, and the GA may have to reverse the design process (i.e., apply the remove-transformations) in order to apply needed changes. The results of merely SA based systems are, thus, unsatisfactory.

7 Conclusions and Future Work

We have presented an approach that uses SA in software architecture synthesis. A base architecture is given as input and architecture styles and design patterns are used as transformations when searching for a better solution in the neighborhood. The solution is evaluated with regard to modifiability, efficiency and complexity.

The experimental results achieved with this approach show that SA on its own is not able to produce good quality solutions in terms of quality values or the resulting UML class diagrams. Attempts of improving the SA based solution with GA were also unsuccessful in increasing the quality values. However, when combining GA and SA so that the SA fine-tunes a basic solution achieved with the GA, both the quality values and the class diagrams are very good. Moreover, as SA is significantly faster than the GA, the result was obtained much quicker than would have been possible by using only GA. Thus, it is concluded that while SA is not sophisticated enough to be able to introduce complex alterations that require several transformations and produce delayed reward, it is able to quickly improve solutions where the base for such alteration has already been made.

It should be noted though, that SA seems to act very "single-mindedly". When SA was used on its own, no solutions contained the message dispatcher architecture style. When SA was used after the GA, all the solutions used the message dispatcher architecture style very heavily, whether it was actually desired or not. Thus, it appears that the mechanism in SA that should prevent it from being stuck to a local optimum is not sufficient to divert the search in the case of software architecture synthesis.

When compared to the manual process, any of the presented algorithms (GA, SA, GASA or SAGA) performs significantly faster than a human designer. A human designer would need several hours to peform the design task, while our algorithms manage in mere minutes. In terms of quality, the GA and GASA come quite close to results from a human designer. Previous studies have shown that GA is at a level of a college student [28], and GASA manages to produce better quality and faster results. Thus, in relation to the ultimate goal of automating software engineering, this paper brings us closer to that goal by providing a more efficient way of automating software architecture design while also producing better quality results than what have been previously achieved with GA alone.

In our future work we will concentrate on practical issues, and improve our basic implementation so that patterns (which are currently hardcoded), could be added at will. This will significantly increase the search space, but will also make the need for an algorithm to handle a large amount of patterns even greater. Moreover, the larger the system is and the more computation is required, the more there will also be need for a way to quicken the evolutionary process. Thus, we will also be doing experiments on very large systems to further see how much the seeded algorithm can outperform the GA in terms of time.

8 Acknowledgements

The authors would like to thank professor Kai Koskimies for helpful discussions and the anonymous referees for their valuable comments.

References

[1] Aleti, A., Björnander, S., Grunske, L., and Meedeniya, I. Archeopterix: an extendable tool for architecture optimization of AADL models. In *Proceedings of the ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pages 61–71, 2009.

[2] Amoui, M., Mirarab, S., Ansari, S., and Lucas, C. A genetic algorithm approach to design evolution using design pattern transformation. *International Journal of Information Technology and Intelligent Computing*, 1:235–245, 2007.

[3] Aragon, C. R., Johnson, D. S., McGeoch, L. A., and Schevon, C. Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, 1991.

[4] Bansiya, J. and Davis, C. G. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[5] Bodhuin, T., Di Penta, M., and Troiano, L. A search-based approach for dynamically re-packaging downloadable applications. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON07)*, pages 27–41, 2007.

[6] Bouktif, S., Sahraoui, H., and Antoniol, G. Simulated annealing for improving software quality prediction. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2006)*, pages 1893–1900. ACM, 2006.

[7] Bowman, M., Briand, L. C., and Labiche, Y. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. *IEEE Transaction on Software Engineering*, 36(6):817–837, 2010.

[8] Chidamber, S. R. and Kemerer, C. F. A metrics suite for object oriented design. *IEEE Transaction on Software Engineering*, 20(6):476–492, 1994.

[9] Clarke, J., Dolado, J. J., Harman, M., Hierons, R. M., Jones, B., Lumkin, M., Mitchell, B., Mancoridis, S., Rees, K., Roper, M., and Shepperd, M. Reformulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, 2003.

[10] Frankel, D. S. *Model Driven Architecture - Applying MDA to Enterprise Computing.* Wiley Publishing, Inc., 2003.

[11] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[12] Gold, N., Harman, M., Li, Z., and Mahdavi, K. A search based approach to overlapping concept boundaries. In *Proceedings of the 22nd International Conference on Software Maintenance (ICSM 06)*, pages 310–319. IEEE, 2006.

[13] Harman, M., Mansouri, S. A., and Zhang, Y. Search based software engineering: a comprehensive review of trends, techniques and applications. Technical report TR-09-03, King's College, London, United Kingdom, 2009.

[14] Harman, M. and Tratt, L. Pareto optimal search based refactoring at the design level. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, pages 1106–1113. ACM, 2007.

[15] Jensen, A. C. and Cheng, B. H. C. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2010)*, pages 1341–1348. ACM, 2010.

[16] Johnson, D. S., Aragon, C. R., McGeoch, L. A., and Schevon, C. Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Operations Research*, 37(6):865–892, 1989.

[17] Julstrom, B. A. Seeding the population: improved performance in a genetic algorithm for the rectilinear Steiner problem. In *Proceedings of the ACM Symposium on Applied Computing (SAC94)*, pages 222–226. ACM, 1994.

[18] Kirkpatrick, S., Gelatt, C., and Vecchi, M. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[19] Laarhoven van, P. J. M. and Aarts, E. *Simulated Annealing: Theory and Applications.* Kluwer, 1987.

[20] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., and Teller, E. Equation of state calculation by fast computing machines. *Journal of Chemical Physics*, 21:32–40, 1953.

[21] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolutionary Programs.* Springer-Verlag, 1992.

[22] Mitchell, M. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[23] O'Keeffe, M. and Ó Cinnéide, M. Towards automated design improvements through combinatorial optimization. In *Workshop on Directions in Software Engineering Environments (WoDiSEE2004), W2S Workshop - 26th International Conference on Software Engineering*, pages 75–82. IEEE, 2004.

[24] O'Keeffe, M. and Ó Cinnéide, M. Search-based software maintenance. In *Proceedings of Conference on Software Maintenance and Re-engineering (CSMR'06)*, pages 249–260. IEEE, 2006.

[25] O'Keeffe, M. and Ó Cinnéide, M. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, 2008.

[26] Räihä, O. A survey on search-based software design. *Computer Science Review*, 4(4):203–249, 2010.

[27] Räihä, O. *Genetic Algorithms in Software Architecture Synthesis.* PhD thesis, University of Tampere, 2011.

[28] Räihä, O., Hadaytullah, Koskimies, K., and Mäkinen, E. Synthesizing architecture from requirements: A genetic approach,. In *Relating Software Requirements and Architectures*, pages 307–331. Springer, 2011.

[29] Räihä, O., Koskimies, K., and Mäkinen, E. Genetic synthesis of software architecture. In *Proceedings of the 7th International Conference on Simulated Evolution and Learning (SEAL08)*, pages 565–574. Springer, 2008.

[30] Räihä, O., Koskimies, K., and Mäkinen, E. Empirical study on the effect of crossover in genetic software architecture synthesis. In *Proceedings of the World Congress on Nature and Biologically Inspired Computing (NaBIC09)*, pages 619–625. IEEE, 2009.

[31] Räihä, O., Koskimies, K., and Mäkinen, E. Scenario-based genetic synthesis of software architecture. In *Proceedings of the 4th International Conference on Software Engineering Advances (ICSEA09)*, pages 437–445. IEEE, 2009.

[32] Räihä, O., Koskimies, K., and Mäkinen, E. Complementary crossover for genetic software architecture synthesis. In *Proceedings of the 10th International Conference on Intelligent Systems Design and Applications (ISDA10)*, pages 359–366. IEEE, 2010.

[33] Räihä, O., Koskimies, K., and Mäkinen, E. Generating software architecture spectrum with multi-objective genetic algorithms. In *Proceedings of the Third World Congress on Nature and Biologically Inspired Computing (NaBIC11)*, pages 29–36. IEEE, 2011.

[34] Räihä, O., Koskimies, K., Mäkinen, E., and Systä, T. Pattern-based genetic model refinements in MDA. *Nordic Journal of Computing*, 14(4):322–339, 2008.

[35] Räihä, O., Mäkinen, E., and Poranen, T. Using simulated annealing for producing software architectures. Technical Report D-2009-2, University of Tampere, Tampere, Finland, 2009.

[36] Ramsey, C. L. and Grefenstett, J. J. Case-based initialization of genetic algorithms. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 84–91. Morgan Kaufmann Publishers, 1993.

[37] Seng, O., Bauyer, M., Biehl, M., and Pache, G. Search-based improvement of subsystem decomposition. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005)*, pages 1045–1051. ACM, 2005.

[38] Seng, O., Stammel, J., and Burkhart, D. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2006)*, pages 1909–1926. ACM, 2006.

[39] Shaw, M. and Garlan, D. *Software Architecture - Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[40] Simons, C. L. and Parmee, I. C. A cross-disciplinary technology transfer for search-based evolutionary computing: from engineering design to software engineering design. *Engineering Optimization*, 39(5):631–648, 2007.

[41] Simons, C. L. and Parmee, I. C. Single and multi-objective genetic operators in object-oriented conceptual software design. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, pages 1957–1958. ACM, 2007.

[42] Simons, C. L. and Parmee, I. C. Dynamic parameter control of interactive local search in UML software design. In *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, pages 3397–3404. IEEE Press, 2010.

[43] Simons, C. L. and Parmee, I. C. Elegant object-oriented software design via interactive, evolutionary computation. *IEEE Transactions on Systems, Man and Cybernetics, Part C Applications and Reviews*, 42(6):1797–1805, 2012.

[44] Simons, C. L., Parmee, I. C., and Gwynllyw, R. Interactive, evolutionary search in upstream object-oriented class design. *IEEE Transactions on Software Engineering*, 36(6):798816, 2010.

[45] Trikia, E., Colletteb, Y., and Siarry, P. A theoretical study on the behavior of simulated annealing leading to a new cooling schedule. *European Journal of Operational Research*, 166:77–92, 2005.

[46] Varadharajan, T. K. and Rajendran, C. A multi-objective simulated-annealing algorithm for scheduling in flowshops to minimize the makespan and total flowtime of jobs. *European Journal of Operational Research*, 167:772–795, 2005.

Received 20th August 2012

Acta Cybernetica **21** (2013) 267–271.

Realizing Small Tournaments Through Few Permutations*

Christian Eggermont[†], Cor Hurkens[†] and Gerhard J. Woeginger[†]

Abstract

Every tournament on 7 vertices is the majority relation of a 3-permutation profile, and there exist tournaments on 8 vertices that do not have this property. Furthermore every tournament on 8 or 9 vertices is the majority relation of a 5-permutation profile.

Keywords: voting systems; digraph realization; extremal combinatorics

1 Introduction

A *tournament* $T = (V, A)$ is a directed graph on a vertex set V whose arc set A contains exactly one arc between any pair of distinct vertices. A finite family of (not necessarily distinct) permutations of V forms a *realization* of the tournament, if for every arc $uv \in A$ vertex u precedes vertex v in more than half of the permutations. A realization of the tournament by k permutations is called a *k-permutation profile*. McGarvey [2] proved that every tournament has a realization by a finite number of permutations. Subsequent results by Stearns [6] and Erdős & Moser [1] yield that every tournament on n vertices can be realized by $O(n/\log n)$ permutations, and that some tournaments on n vertices cannot be realized by fewer than $\Omega(n/\log n)$ permutations. We define the *McGarvey number* $\mathrm{McG}(T)$ of a tournament T as the size of the smallest possible permutation family that realizes the tournament; note that $\mathrm{McG}(T)$ always is an odd integer.

Shepardson & Tovey [5] analyzed several combinatorial questions on the so-called *predictability number* of tournaments, a parameter closely related to realizations of tournaments. Page 502 of [5] formulates the conjecture that every 7-vertex tournament T has $\mathrm{McG}(T) \leq 3$. In this technical note we confirm this conjecture, and we also discuss a number of related questions. Our results confirm this conjecture:

*This work was supported bythe Netherlands Organisation for Scientific Research (NWO), grant 639.033.403; by DIAMANT (an NWO mathematics cluster); by the Future and Emerging Technologies unit of the European Community (IST priority), under contract no. FP6-021235-2 (project ARRIVAL).

[†]Department of Mathematics and Computer Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, Netherlands, E-mail: {c.e.j.eggermont,wscor,gwoegi}@win.tue.nl

- Every 7-vertex tournament T satisfies $\mathrm{McG}(T) \leq 3$.
- Every 8-vertex and every 9-vertex tournament T satisfies $\mathrm{McG}(T) \leq 5$.
- There exist 96 non-isomorphic 8-vertex tournaments T with $\mathrm{McG}(T) = 5$.
- There exist 17.674 non-isomorphic 9-vertex tournaments with $\mathrm{McG}(T) = 5$.

All our results have been derived with the help of computer programs, and in particular with the help of the software packages AIMMS and CPLEX.

2 Mathematical model and computational results

We express every permutation of the vertex set $V = \{1, 2, \ldots, n\}$ by a a transitive tournament, which can be considered as a permutation (total order) of V. It is well-known (see for instance Moon [4]) that a tournament without directed triangles is transitive. We use n^2 integer variables $x_{uv} \in \{0, 1\}$ with $u, v = 1, \ldots, n$ to encode the arcs of the tournament, and we impose the following two families of linear inequalities.

$$x_{uv} + x_{vu} = 1 \qquad \text{for all } u, v \in V$$

$$x_{uv} + x_{vw} + x_{wu} \leq 2 \qquad \text{for all } u, v, w \in V$$

The first constraint family enforces that for every two vertices u, v there is either an arc uv or an arc vu but not both. The second constraint family forbids the occurrence of directed triangles (and thus makes the tournament transitive).

In order to decide whether a given tournament $T = (V, A)$ can be realized by three permutations, we introduce three such sets of integer variables x_{uv}, x'_{uv}, x''_{uv} together with the corresponding families of constraints. Furthermore we add the constraints

$$x_{uv} + x'_{uv} + x''_{uv} \geq 2 \qquad \text{for all } uv \in A.$$

These constraints ensure that vertex u precedes vertex v in more than half of the three permutations. McKay [3] gives a list that enumerates all 456 non-isomorphic tournaments on seven vertices. We worked through the tournaments on this list one by one, and for each of them the software package AIMMS managed to find a feasible solution to the corresponding linear integer program. We also worked through the list of 6,880 non-isomorphic tournaments on eight vertices and through the list of 191,536 non-isomorphic tournaments on nine vertices; for all these tournaments AIMMS found a realization by five permutations.

Theorem 1. *Every tournament on $n \leq 7$ vertices has a realization by three permutations. Every tournament on $n \leq 9$ vertices has a realization by five permutations.* □

0000110.011000.00010.0001.100.10.1	1001010.110010.11000.0101.001.00.0
0000110.101000.10010.0001.100.10.1	1001100.110010.10010.1010.001.00.1
0001010.011000.00100.0001.110.10.1	1001100.111000.01010.0010.101.10.1
0001100.101000.10010.0001.100.10.1	1010000.000111.01000.1100.101.11.0
0001100.101000.10100.0010.110.11.1	1010000.001101.00010.1100.101.11.0
0010001.010100.00010.0110.000.10.1	1010000.100011.00101.1100.010.01.0
0010001.110000.01010.0110.100.10.1	1010000.100011.10100.1110.100.11.1
0010010.101000.00110.0001.100.10.1	1010000.100101.00011.1100.010.01.0
0010010.101000.10100.1001.110.10.1	1010000.100101.01001.1100.110.11.0
0010100.101000.00011.0001.100.10.0	1010000.100101.10010.1110.100.11.1
0010100.101000.10010.1001.110.10.1	1010000.101001.00101.1100.110.11.0
0011000.000011.00100.1010.101.10.1	1010000.101001.00110.1100.101.11.0
0011000.000110.10000.0111.100.11.1	1010000.101100.10001.1110.110.11.1
0011000.000110.10000.1101.011.10.1	1010000.111000.01100.1110.111.11.1
0011000.000110.10000.1101.110.11.1	1010001.110001.00110.0011.000.00.0
0011000.010010.00100.1110.011.01.1	1010001.110001.00110.1010.010.00.1
0011000.010010.00100.1110.101.11.1	1010001.110001.00110.1010.100.10.1
0011000.100100.01010.0001.110.10.1	1010001.110010.00011.1100.100.01.0
0011000.100100.10100.0011.010.10.1	1010001.110010.00101.0110.000.01.0
0011000.101000.10100.0011.110.10.1	1010001.110010.10100.1011.001.00.0
0011100.110000.00110.1010.001.10.1	1010001.110010.10100.1011.010.00.1
0011100.110000.01010.0110.001.10.1	1010001.110100.00011.1010.001.00.0
0100010.001100.10010.0001.100.10.1	1010001.110100.00011.1010.100.10.1
0100010.011000.10010.0101.100.10.1	1010001.110100.01010.0011.010.00.1
0100010.101000.10101.0001.100.10.0	1010001.110100.01010.0011.100.01.0
0100100.010000.01010.0110.100.11.1	1010010.110001.00011.1100.100.01.0
0100100.010010.00110.0001.000.10.1	1010010.110001.00101.0110.000.01.0
0100100.101000.10011.0001.100.10.0	1010100.101010.01100.0001.001.10.0
0101000.110000.10110.0011.100.10.1	1010100.110001.00011.1010.010.00.1
0101100.010100.01010.0010.001.10.1	1010100.110001.01010.0011.001.00.0
0101100.011000.01010.0010.101.10.1	1010100.111000.01010.0110.001.10.1
0110000.001100.10010.1001.110.10.1	1010100.111000.01010.1010.101.10.1
0110000.101000.10110.0011.100.10.1	1100000.010110.11000.0111.100.11.1
0110010.100001.10101.1001.100.10.0	1100000.011010.00110.0001.100.10.1
0110100.100001.10011.1001.100.10.0	1100000.101010.10110.1000.110.01.1
0110100.101000.11010.0101.001.10.0	1100000.110010.10110.1100.010.01.1
0111000.100100.11010.1010.110.01.1	1100000.110100.10110.0011.000.10.1
1000010.110100.11000.0101.010.10.1	1100000.110100.11010.1001.110.10.1
1000010.111000.00110.0001.100.10.1	1100000.111000.10101.0011.100.10.0
1000100.010110.01000.0010.100.11.1	1100100.110010.11010.1010.001.00.1
1000100.110010.11000.1100.110.11.1	1101000.011100.00110.0010.001.10.1
1000100.110100.10010.1001.010.10.1	1101000.101010.11010.0001.101.00.0
1001000.100100.00011.1000.110.10.1	1101000.110010.11100.0101.001.10.0
1001000.100110.10010.0100.100.11.1	1110000.100110.11001.0011.100.01.0
1001000.101100.00110.0000.110.11.1	1110000.101010.10101.0011.100.01.0
1001000.110010.10001.1100.100.10.1	1110000.101010.11010.0101.001.00.0
1001000.110100.01100.0010.110.11.1	1110000.110001.10011.1110.100.01.0
1001000.110100.11000.0011.110.10.1	1110000.111000.11100.1110.011.10.1

Table 1: The 96 non-isomorphic 8-vertex tournaments T with $\mathrm{McG}(T) = 5$.

	0	1	2	3	4	5	6	7
0	-	*	1	1	0	0	*	*
1	*	-	*	1	1	0	0	*
2	0	*	-	*	1	1	0	*
3	0	0	*	-	1	1	1	0
4	1	0	0	0	-	*	1	1
5	1	1	0	0	*	-	1	0
6	*	1	1	0	0	0	-	1
7	*	*	*	1	0	1	0	-

Table 2: The adjacency matrix of the directed graph G_8.

In our computational experiments, we detected that 96 of the 6.880 non-isomorphic 8-vertex tournaments cannot be realized by three permutations. These tournaments are listed in Table 1. Each tournament is represented as the upper triangle of the adjacency matrix in row order, and consecutive rows are always separated by dots (this is the representation used in McKay's list [3]).

We also took a closer look at these 96 exceptional tournaments, and tried to understand their common properties. We used CPLEX to analyze their structure, and to identify minimal infeasible subsystems of the underlying linear integer programs. It turned out that all 96 tournaments contain the directed subgraph G_8 whose adjacency matrix is depicted in Table 2. The arcs marked by '*' are unspecified, and their orientation can be set arbitrarily in the tournaments. (Note: Since there are eight vertex pairs with unspecified arcs, this would yield 256 corresponding 8-vertex tournaments; however symmetries and isomorphisms reduce this number to 96.)

Observation 2. *If a tournament T contains the graph G_8 as a subgraph on eight vertices, then T has no realization by three permutations.* □

We stress that the copyright on this graph G_8 belongs to Shepardson & Tovey [5] who established that any tournament containing a subgraph G_8 has a predictability number of at most 13/20.

Finally, our programs detected that 17,674 out of 191,536 non-isomorphic 9-vertex tournaments cannot be realized by three permutations.

3 Conclusions

The computational approach described in this note is strong enough to handle all tournaments with $n \leq 9$ vertices. For $n = 10$ vertices the running times would still be manageable, but we did not spend much time on McKay's list [3] with 9,733,056 non-isomorphic tournaments on ten vertices: we do not expect any surprises from them, and we firmly believe that all of them will be realizable by five permutations.

It might be interesting to determine the smallest tournament that has no realization by five permutations. We randomly explored a (tiny) fraction of the set of 20-vertex tournaments, but we did not succeed in finding anything (for $n > 20$ the computation times become prohibitively large). The counting argument of Stearns [6] yields the existence of a 41-vertex tournament T_{41} with $\mathrm{McG}(T_{41}) \geq 7$. However, for small tournaments the asymptotic bounds implied by [6] seem to be rather loose: The same counting argument only yields the existence of a 19-vertex tournament T_{19} with $\mathrm{McG}(T_{19}) \geq 5$, whereas we know that there exist 8-vertex tournaments with that property.

References

[1] P. Erdős and L. Moser. On the representation of directed graphs as unions of orderings. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences, Series A 9*, 125–132, 1964.

[2] D.C. McGarvey. A theorem on the construction of voting paradoxes. *Econometrica 21*, 608–610, 1953.

[3] B. McKay. Combinatorial data — Catalogue of non-isomorphic tournaments up to 10 vertices. 2008. `http://cs.anu.edu.au/~bdm/data/digraphs.html`

[4] J.W. Moon. Topics on tournaments. *Holt, Rinehart, and Winston, New York*, 1968.

[5] D. Shepardson and C.A. Tovey. Smallest tournaments not realizable by $\frac{2}{3}$-majority voting. *Social Choice and Welfare 33*, 495–503, 2009.

[6] R. Stearns. The voting problem. *American Mathematical Monthly 66*, 761–763, 1959.

Received 18th January 2013

Acta Cybernetica **21** (2013) 273–285.

On Closedness Conditions, Strong Separation, and Convex Duality

Miklós Ujvári*

Abstract

In the paper, we describe various applications of closedness and duality theorems from previous works of the author. First, the strong separability of a polyhedron and a linear image of a convex set is characterized. Then, it is shown how stability conditions (known from the generalized Fenchel-Rockafellar duality theory) can be reformulated as closedness conditions. Finally, we present a generalized Lagrangian duality theorem for Lagrangian programs described with cone-convex/cone-polyhedral mappings.

Keywords: regularity condition, strong separation, convex duality

1 Introduction

Closedness conditions require the closedness of convex sets of the form

$$(AC_1) + C_2 := \{Ax + y : x \in C_1,\, y \in C_2\}$$

or

$$C_1 + A^{-1}(C_2) := \{x + v : x \in C_1,\, Av \in C_2\},$$

where A is an m by n real matrix, C_1 and C_2 are convex sets in $\mathcal{R}^n$ and $\mathcal{R}^m$, respectively. These conditions play an important role in the theory of duality in convex programming, see [7] and [8]. In this paper our aim is to describe further applications.

We begin this paper with stating the main results of [7] and [8]. First we fix some notation.

Let us denote by $\operatorname{rec} C$ and $\operatorname{bar} C$ the *recession cone* and the *barrier cone* of a convex set C in $\mathcal{R}^d$, respectively, that is let

$$\begin{aligned} \operatorname{rec} C &:= \left\{v \in \mathcal{R}^d : x + \lambda v \in C \ (x \in C,\, \lambda \geq 0)\right\}, \\ \operatorname{bar} C &:= \left\{w \in \mathcal{R}^d : \inf\{w^T x : x \in C\} > -\infty\right\}. \end{aligned}$$

Then $\operatorname{rec} C$ and $\operatorname{bar} C$ are convex cones.

*H-2600 Vác, Szent János utca 1., Hungary. E-mail: ujvarim@cs.elte.hu

Let us denote by $\operatorname{ri} C$ (resp. $\operatorname{cl} C$) the *relative interior* (resp. *closure*) of the convex set C in $\mathcal{R}^d$. The relative interior of a convex set C is convex, and is nonempty if the convex set C is nonempty. (See [4] for the definition and properties of the relative interior.)

The main result of [7] and [8] is the following closedness theorem. See [8] for an extension of Theorem 1.1 with statements concerning the recession cones. See [3], [7] for further closedness theorems.

Theorem 1.1. *Let A be an m by n real matrix. Let C_1 be a closed convex set in $\mathcal{R}^n$, and let P_2 be a polyhedron in $\mathcal{R}^m$. Then between the statements*

a) $(A^T \operatorname{bar} P_2) \cap \operatorname{ri}(\operatorname{bar} C_1) \neq \emptyset$,

b) $A^{-1}(-\operatorname{rec} P_2) \cap (\operatorname{rec} C_1) \subseteq -\operatorname{rec} C_1$,

c) $(AC_1) + P_2$ *is closed,*

d) $C_1 + A^{-1}(P_2)$ *is closed,*

hold the following logical relations: a) is equivalent to b); c) is equivalent to d); a) or b) implies c) and d).

In [7] two applications of Theorem 1.1 are mentioned. These duality theorems are stated in Theorems 1.2 and 1.3.

We will use the terminology and notations of [5] here. Let $f : \mathcal{R}^n \to \mathcal{R} \cup \{+\infty\}$ be a convex function, and let $g : \mathcal{R}^m \to \mathcal{R} \cup \{-\infty\}$ be a concave function. Let $A \in \mathcal{R}^{m \times n}$ be a matrix, and let $a \in \mathcal{R}^n$, $b \in \mathcal{R}^m$ be vectors. We will consider the following pair of programs from [5]:

$$\begin{aligned} (P): &\quad \text{Find} \quad \inf\{f(x) - g(Ax - b) + a^T x : x \in \mathcal{R}^n\}, \\ (D): &\quad \text{Find} \quad \sup\{g^c(y) - f^c(A^T y - a) + b^T y : y \in \mathcal{R}^m\}. \end{aligned}$$

Here f^c and g^c denote the *convex conjugate function* of f and the *concave conjugate function* of g, respectively, that is let

$$f^c(w) := \sup\left\{w^T x - f(x) : x \in \mathcal{R}^n\right\}, \quad g^c(y) := \inf\left\{y^T z - g(z) : z \in \mathcal{R}^m\right\}.$$

Let $[f]$ and $[g]$ denote the *epigraph* of f and the *hypograph* of g, respectively, that is let

$$[f] := \{(x, \mu) \in \mathcal{R}^{n+1} : f(x) \leq \mu\}, \quad [g] := \{(z, \nu) \in \mathcal{R}^{m+1} : g(z) \geq \nu\}.$$

The function f is *closed* whenever its epigraph $[f]$ is closed, and f is a *polyhedral convex* function when its epigraph $[f]$ is a polyhedron. Let $F(f)$ and $F(g)$ denote the domain of finiteness of the functions f and g, respectively, that is let

$$F(f) := \{x \in \mathcal{R}^n : f(x) < +\infty\}, \quad F(g) := \{z \in \mathcal{R}^m : g(z) > -\infty\}.$$

The points of the set

$$\mathbf{P} := F(f) \cap \{x : Ax - b \in F(g)\}$$

are called the *feasible solutions* of program (P). We denote by v_P the *optimal value* of program (P), that is let

$$v_P := \inf \left\{ f(x) - g(Ax - b) + a^T x : x \in \mathbf{P} \right\}.$$

For the program (D) the set $\mathbf{D}$ and the value v_D can be defined similarly.

With this notation the main duality results of [7] can be stated as follows.

Theorem 1.2. *Let f be a convex function on $\mathcal{R}^n$, and let $-g$ be a polyhedral convex function on $\mathcal{R}^m$. Then between the statements*

a) the function f is closed, and there exists a strictly feasible solution of the program (D), that is a point $y_0 \in \mathcal{R}^m$ such that $y_0 \in F(g^c)$ and $A^T y_0 - a \in \operatorname{ri} F(f^c)$,

b) it holds that $\mathbf{P} \cup \mathbf{D} \neq \emptyset$, and the primal closedness assumption is satisfied, that is the set

$$C_P := \begin{pmatrix} A & 0 \\ a^T & 1 \end{pmatrix} [f] + (-[g])$$

is closed,

c) the optimal values of programs (P) and (D) are equal, and the primal optimal value v_P is attained if it is finite,

hold the following logical relations: a) implies b); b) implies c).

The next theorem is a counterpart of Theorem 1.2, as for closed convex functions f and $-g$ the equations $f^{cc} = f$ and $g^{cc} = g$ hold, so Theorem 1.2 can be dualized.

Theorem 1.3. *Let f be a closed convex function on $\mathcal{R}^n$, and let $-g$ be a polyhedral convex function on $\mathcal{R}^m$. Then between the statements*

a) there exists a strictly feasible solution of the program (P), that is a point $x_0 \in \mathcal{R}^n$ such that $x_0 \in \operatorname{ri} F(f)$ and $Ax_0 - b \in F(g)$,

b) it holds that $\mathbf{P} \cup \mathbf{D} \neq \emptyset$, and the dual closedness assumption is satisfied, that is the set

$$C_D := \begin{pmatrix} A^T & 0 \\ b^T & 1 \end{pmatrix} [g^c] + (-[f^c])$$

is closed,

c) the optimal values of programs (P) and (D) are equal, and the dual optimal value v_D is attained if it is finite,

hold the following logical relations: a) implies b); b) implies c).

In the paper, we describe various applications of these closedness and duality theorems: Theorems 1.1, 1.2, and 1.3 will be applied in Sections 2, 3, and 4, respectively. In Section 2 an analogue of Theorem 1.1 is proved, where the property closedness is replaced by strong separability. In Section 3 we reformulate stability conditions (known from the generalized Fenchel-Rockafellar duality theory, see [5]) as closedness conditions. Generalized Lagrangian duality (for programs with cone-convex constraints) is the topic of several papers, see for example [9], [2], and [1]. Our approach is different: in Section 4 we study Lagrangian programs described with cone-convex/cone-polyhedral mappings.

2 Strong separation

In this section we will prove an analogue of Theorem 1.1 for strong separation, where the property "closed" is replaced with the property "the origin is not an element of the closure".

Two nonempty convex sets C_1 and C_2 in $\mathcal{R}^n$ are called *strongly separable* if there exists a vector $a_1 \in \mathcal{R}^n$ such that

$$\sup\{a_1^T x_1 : x_1 \in C_1\} < \inf\{a_1^T x_2 : x_2 \in C_2\}.$$

It is well-known (see [4], Theorem 11.4) that the sets C_1 and C_2 are strongly separable if and only if $0 \notin \operatorname{cl}(C_2 + (-C_1))$. (Note that the sets C_1 and C_2 are disjoint if and only if $0 \notin C_2 + (-C_1)$.) This fact implies the following lemma (see Corollaries 11.4.2 and 19.3.3 in [4]).

Lemma 2.1. *Let C_1 be a convex set in $\mathcal{R}^n$, and let P_1, P_2 be polyhedrons in $\mathcal{R}^n$. Then, the following statements hold:*

a) If $0 \notin \operatorname{cl} C_1$ then the sets $\{0\}$ and C_1 are strongly separable.

b) If $P_1 \cap P_2 = \emptyset$ then the sets P_1 and P_2 are strongly separable.

The next theorem is an immediate consequence of Theorem 1.1.

Theorem 2.1. *Let A be an m by n real matrix. Let C_1 be a convex set in $\mathcal{R}^n$, and let P_2 be a polyhedron in $\mathcal{R}^m$. Then between the statements*

a) $0 \notin (AC_1) + P_2$ (that is the sets AC_1 and $-P_2$ are disjoint),

b) $0 \notin C_1 + A^{-1}(P_2)$ (that is the sets $-C_1$ and $A^{-1}(P_2)$ are disjoint),

c) $0 \notin \operatorname{cl}((AC_1) + P_2)$ (that is the sets AC_1 and $-P_2$ are strongly separable),

d) $0 \notin \operatorname{cl}(C_1 + A^{-1}(P_2))$ (that is the sets $-C_1$ and $A^{-1}(P_2)$ are strongly separable),

hold the following logical relations: a) is equivalent to b); a) is equivalent to c) if the set $(AC_1) + P_2$ is closed; b) is equivalent to d) if the set $C_1 + A^{-1}(P_2)$ is closed.

Specially, all the four statements are equivalent if from Theorem 1.1 statement a), b), c) or d) holds. □

The statements c) and d) in Theorem 2.1 are equivalent in the general case as well, as the following theorem shows.

Theorem 2.2. *Let A be an m by n real matrix. Let C_1 and C_2 be convex sets in $\mathcal{R}^n$ and $\mathcal{R}^m$, respectively. Then,*

a) if $0 \notin \operatorname{cl}((AC_1) + C_2)$ then $0 \notin \operatorname{cl}(C_1 + A^{-1}(C_2))$ (in other words the strong separability of the sets AC_1 and $-C_2$ implies the strong separability of the sets $-C_1$ and $A^{-1}(C_2)$),

b) the statement a) can be reversed if $C_2 \subseteq A(\mathcal{R}^n)$,

c) the statement a) can be reversed if the set C_2 is a polyhedron.

Proof. a) The proof is indirect: We will show that $0 \in \mathrm{cl}\,(C_1 + A^{-1}(C_2))$ implies $0 \in \mathrm{cl}\,((AC_1) + C_2)$. Let $x_i \in C_1$, $v_i \in A^{-1}(C_2)$ for $i = 1, 2, \ldots$, and suppose that $x_i + v_i \to 0$ $(i \to \infty)$. Then $A(x_i + v_i) \to 0$ $(i \to \infty)$ also holds. As $Av_i \in C_2$ for $i = 1, 2, \ldots$ by definition, we can see that $0 \in \mathrm{cl}\,((AC_1) + C_2)$; the statement a) is proved.

b) Let us suppose now that the set C_2 is a subset of the image space of the matrix A. We will show that then $0 \notin \mathrm{cl}\,(C_1 + A^{-1}(C_2))$ implies $0 \notin \mathrm{cl}\,((AC_1) + C_2)$. By Lemma 2.1, the origin can be strongly separated from the convex set $C_1 + A^{-1}(C_2)$, that is there exists a vector $a_1 \in \mathcal{R}^n$ such that

$$0 < \inf\{a_1^T x : x \in C_1 + A^{-1}(C_2)\}. \tag{1}$$

As the recession cone of the set $A^{-1}(C_2)$ contains the null space of the matrix A, the inequality (1) implies that the vector a_1 is an element of the image space $A^T(\mathcal{R}^m)$: there exists a vector $z \in \mathcal{R}^m$ such that $a_1 = A^T z$.

Suppose indirectly, that $0 \in \mathrm{cl}\,((AC_1) + C_2)$. Then there exist points $x_i \in C_1$, $y_i \in C_2$ $(i = 1, 2, \ldots)$ such that

$$Ax_i + y_i \to 0 \ (i \to \infty).$$

By assumption, the set C_2 is a subset of the image space of the matrix A, so for some vectors $v_i \in \mathcal{R}^n$ (actually, $v_i \in A^{-1}(C_2)$), the equalities $y_i = Av_i$ $(i = 1, 2, \ldots)$ hold. But then

$$a_1^T(x_i + v_i) = z^T(Ax_i + y_i) \to 0 \ (i \to \infty),$$

contradicting (1). Hence, $0 \notin \mathrm{cl}\,((AC_1) + C_2)$; statement b) is proved as well.

c) Let us suppose that the set C_2 is a polyhedron. We will show that then the strong separability of the sets $-C_1$ and $A^{-1}(C_2)$ implies the strong separability of the sets AC_1 and $-C_2$. Notice that

$$A^{-1}(C_2) = A^{-1}(C_2 \cap A(\mathcal{R}^n)).$$

Here the set $C_2 \cap A(\mathcal{R}^n)$ is a subset of the image space of the matrix A, so by the statement b) the strong separability of the sets $-C_1$ and $A^{-1}(C_2)$ implies the strong separability of the sets AC_1 and $-C_2 \cap A(\mathcal{R}^n)$. Hence, there exist a vector $b_2 \in \mathcal{R}^m$ and a constant $\delta \in \mathcal{R}$ such that the set AC_1 is a subset of the closed halfspace $H^+ := \{y : b_2^T y \leq \delta\}$, and the polyhedrons $H^+ \cap A(\mathcal{R}^n)$ and $-C_2$ are disjoint. By Lemma 2.1, two disjoint polyhedrons are strongly separable, so the strong separability of the sets AC_1 and $-C_2$ follows, which finishes the proof of the theorem. □

Finally, we remark that the statement a) in Theorem 2.2 can not be reversed generally, even if the sets C_1 and C_2 are supposed to be closed and convex: there exist closed convex sets C_1 and C_2 such that

$$0 \in \mathrm{cl}\,((AC_1) + C_2),\ 0 \notin \mathrm{cl}\,(C_1 + A^{-1}(C_2))$$

for some linear mapping A.

In fact, let

$$A : (\lambda, \mu) \mapsto \lambda \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} + \mu \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \ (\lambda, \mu \in \mathcal{R});$$

$$C_1 := \mathcal{R} \times \{0\} \subseteq \mathcal{R}^2;\ C_2 := \mathrm{PSD}_2 - \begin{pmatrix} 1 & 1/2 \\ 1/2 & 0 \end{pmatrix},$$

where PSD_2 denotes the closed convex cone of the 2 by 2 real symmetric positive semidefinite matrices, that is (see [6]),

$$\mathrm{PSD}_2 = \left\{ \begin{pmatrix} \alpha & \beta \\ \beta & \gamma \end{pmatrix} \in \mathcal{R}^{2\times 2} : \alpha, \gamma, \alpha\gamma - \beta^2 \geq 0 \right\}.$$

Then,

$$\begin{pmatrix} 1+i+1/i & 1/2+i \\ 1/2+i & i \end{pmatrix} - i \cdot \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \to \begin{pmatrix} 1 & 1/2 \\ 1/2 & 0 \end{pmatrix} \ (i \to \infty)$$

shows that

$$\begin{pmatrix} 1 & 1/2 \\ 1/2 & 0 \end{pmatrix} \in \mathrm{cl}\,(\mathrm{PSD}_2 + AC_1).$$

Hence, $0 \in \mathrm{cl}\,((AC_1) + C_2)$.

On the other hand, it can be easily verified that

$$A^{-1}(C_2) = \{(\lambda, \mu) : \lambda \geq -1/2,\ \mu = -1/2\},$$

thus indeed $0 \notin \mathrm{cl}\,(C_1 + A^{-1}(C_2))$; the sets C_1 and C_2 meet the requirements.

3 Stable points

In this section, after describing a geometric and an equivalent algebraic definition of stable points, we reformulate the stability condition as a closedness condition.

The following lemma, concerning the programs (P) and (D), will be used.

Lemma 3.1. *Let us suppose that* $\mathbf{D} \neq \emptyset$. *Then the primal closedness assumption is satisfied (that is the set* C_P *is closed) if and only if for every vector* $b \in \mathcal{R}^m$ *the optimal values of programs* (P) *and* (D) *are equal, and the primal optimal value* v_P *is attained if it is finite.*

Proof. As the definition of the set C_P does not depend on the vector b, so the "only if" part of the lemma is a consequence of Theorem 1.2.

On the other hand, with minor modification of the proof of Theorem 4.1 in [7], it can be shown that:

$$(b, \delta) \in C_P \Leftrightarrow \exists x \in \mathcal{R}^n : f(x) - g(Ax - b) + a^T x \leq \delta;$$

and, in case of $\mathbf{P} \cup \mathbf{D} \neq \emptyset$,

$$(b, \delta) \notin \operatorname{cl} C_P \Leftrightarrow \exists y \in \mathcal{R}^m : g^c(y) - f^c(A^T y - a) + b^T y > \delta.$$

Hence, to prove the "if" part of the lemma, it is enough to verify that for every vector $b \in \mathcal{R}^m$ and for every constant $\delta \in \mathcal{R}$,

$$\exists x \in \mathcal{R}^n : f(x) - g(Ax - b) + a^T x \leq \delta \tag{2}$$

or

$$\exists y \in \mathcal{R}^m : g^c(y) - f^c(A^T y - a) + b^T y > \delta \tag{3}$$

holds. For a given vector $b \in \mathcal{R}^m$ two cases are possible:

Case 1: $\mathbf{P} = \emptyset$. Then $v_P = v_D = \infty$, and (3) holds for every $\delta \in \mathcal{R}$.

Case 2: $\mathbf{P} \neq \emptyset$. Then by assumption $v_P = v_D$ with primal attainment, so (2) holds for $\delta \geq v_P$, and (3) holds for $\delta < v_P$.

This way we have proved the "if" part of the lemma as well. □

The following stability conditions appear in the generalized Fenchel-Rockafellar duality theory concerning programs (P) and (D), see [5]. First, we recall the geometric definition of stability.

Let C be a convex set in $\mathcal{R}^d$, and let $e \in \operatorname{rec} C$. A point $x_0 \in C$ is called a *stable point* of the set C if for every affine set M in $\mathcal{R}^d$ satisfying

$$M \cap (\{x_0\} + \mathcal{R}e) \neq \emptyset \text{ and } M \cap (C + \mathcal{R}_{++}e) = \emptyset, \tag{4}$$

there exists a hyperplane H in $\mathcal{R}^d$ such that

$$M \subseteq H \text{ and } H \cap (C + \mathcal{R}_{++}e) = \emptyset. \tag{5}$$

(Here let $\mathcal{R}_{++}e := \{\lambda e : 0 < \lambda \in \mathcal{R}\}$, and let $\mathcal{R}e := \{\mu e : \mu \in \mathcal{R}\}$. It can be easily seen that (4) implies $e \notin \operatorname{rec} M$, and that (5) implies $e \notin \operatorname{rec} H$.)

For example, let us define the convex sets

$$\begin{aligned} C_1 &:= \{(x_1, x_2) \in \mathcal{R}^2 : x_1 \geq 0,\ x_2 \geq x_1^2\}, \\ C_2 &:= \{(x_1, x_2) \in \mathcal{R}^2 : x_1 \geq 0,\ x_2 \geq -\sqrt{x_1}\}. \end{aligned}$$

Then, the origin $x_0 = (0, 0)$ (with $e = (0, 1)$) is a stable point of the set C_1 but is not a stable point of the set C_2.

For a convex function h defined on $\mathcal{R}^n$ the point $u_0 \in F(h)$ is called a *stable point* of the function h, if (u_0, μ_0) is a stable point of the epigraph $[h]$ (with $e_1 := (0, 1) \in \operatorname{rec}[h]$) for some $\mu_0 \in \mathcal{R}$. In this case the function h is called *u_0-stable*. For example, it is proved in [5], that for every $u_0 \in \operatorname{ri} F(h)$, the function h is u_0-stable.

The next lemma, describing an algebraic characterization of u_0-stability, can also be found in [5], see Lemma 5.5.8.

Lemma 3.2. *Let $u_0 \in F(h)$. A convex function h on $\mathcal{R}^n$ is u_0-stable if and only if for every $n \times m$-matrix B and for every vector $w \in \mathcal{R}^n$ with $u_0 = By_0 - w$ for some $y_0 \in \mathcal{R}^m$, the relation*

$$\hat{h}^c(v) = \min\{h^c(x) + w^T x : B^T x = v\} \tag{6}$$

holds for all $v \in \mathcal{R}^m$. Here $\hat{h}(y) := h(By - w)$.

Now, we can derive, as an immediate consequence of Lemmas 3.1 and 3.2,

Theorem 3.1. *Let $u_0 \in F(h)$. A closed convex function h on $\mathcal{R}^n$ is u_0-stable if and only if for every $n \times m$-matrix B and for every vector $w \in \mathcal{R}^n$ with $u_0 = By_0 - w$ for some $y_0 \in \mathcal{R}^m$, the set*

$$\begin{pmatrix} B^T & 0 \\ w^T & 1 \end{pmatrix} [h^c] \tag{7}$$

is closed.

Proof. Apply Lemma 3.1 to the programs

$$\begin{aligned} (P_0): &\quad \text{Find} \quad \inf\{f_0(x) - g_0(A_0 x - b_0) + a_0^T x : x \in \mathcal{R}^n\}, \\ (D_0): &\quad \text{Find} \quad \sup\{g_0^c(y) - f_0^c(A_0^T y - a_0) + b_0^T y : y \in \mathcal{R}^m\}, \end{aligned}$$

where

$$f_0 := h^c,\ g_0(z) := \begin{cases} 0, & \text{if } z = 0, \\ -\infty & \text{otherwise} \end{cases} \quad (z \in \mathcal{R}^m),$$

$$A_0 := B^T,\ b_0 := v,\ a_0 := w.$$

We obtain that the set in (7) is closed if and only if for all $b_0 \in \mathcal{R}^m$ the optimal values of programs (P_0) and (D_0) are equal, and the primal optimal value v_{P_0} is attained if it is finite. This means that the set in (7) is closed if and only if (6) holds for all $v \in \mathcal{R}^m$. (Note that $\hat{h}^c(v)$ is the optimal value of the dual program (D_0), while the minimum on the right hand side of the equation in (6) is the optimal value of the program (P_0).) Then, Lemma 3.2 gives the statement. □

Specially, let p be a polyhedral convex function on $\mathcal{R}^n$. Then the conjugate function p^c is also a polyhedral convex function. In other words, the epigraph $[p^c]$ and its linear images are polyhedrons. Hence, by Theorem 3.1, for any vector $u_0 \in F(p)$, the function p is u_0-stable. For another proof of this fact, see [5], Theorem 5.5.9.

As special polyhedral convex functions, *partially linear* functions $-g_M$ are u_0-stable for every $u_0 \in F(g_M)$. Here $g_M : \mathcal{R}^n \to \mathcal{R} \cup \{-\infty\}$ is defined as follows:

$$g_M(u) := \begin{cases} \mu, & \text{if } (u, \mu) \in M, \\ -\infty & \text{otherwise,} \end{cases}$$

where $M \subseteq \mathcal{R}^{n+1}$ is an affine set.

The following proposition describes a characterization of stable points in terms of duality, see Theorems 5.3.12 and 5.3.13 in [5].

Proposition 3.1. *Let f be a convex function on $\mathcal{R}^n$, and let u_0 be a point of $F(f)$. Then, f is u_0-stable if and only if*

$$\inf_x (f(x) - g_M(x)) = \max_y (g_M^c(y) - f^c(y))$$

holds for every partially linear convex function $-g_M$ with $u_0 \in F(g_M)$.

We conclude this section with a general duality theorem (Theorem 5.7.5 in [5]) which is based on the notion of stable points. As we will see in the following section, Theorem 3.2 and Theorem 1.3 have a common special case: a duality theorem for generalized Lagrangian programs (Theorem 4.1).

We call program (P) *stably consistent* if there are feasible points x_f and x_g of program (P) such that the function f is x_f-stable and g is z_g-stable, where $z_g := Ax_g - b$. Stable consistency is similarly defined for program (D).

Theorem 3.2. *(Rockafellar) Assume that f is a convex function on $\mathcal{R}^n$ and $-g$ is a convex function on $\mathcal{R}^m$. Then, the following statements hold:*

a) If program (P) is stably consistent (in particular, if it has a strictly feasible solution), then $v_P = v_D$, and the dual optimal value v_D is attained if it is finite.

b) Assume that $f, -g$ are both closed functions. If program (D) is stably consistent (in particular, if it has a strictly feasible solution), then $v_D = v_P$, and the primal optimal value v_P is attained if it is finite.

4 Lagrangian duality

In this section a strong duality theorem concerning generalized Lagrangian programs will be derived from a strengthened version of Theorem 1.3.

Let us begin with describing a well-known property of convex functions, see [4], Theorem 7.5 and Corollary 7.5.1.

Lemma 4.1. *Let f be a convex function on $\mathcal{R}^n$. Then, its closure $\operatorname{cl} f = (f^c)^c$ satisfies*

$$(\operatorname{cl} f)(y) = \lim_{\lambda \to 1} f((1-\lambda)x + \lambda y) \tag{8}$$

for every $x \in \operatorname{ri} F(f)$, $y \in \mathcal{R}^n$. Furthermore, if f is a polyhedral convex function, then $\operatorname{cl} f = f$ and formula (8) holds for every $x \in F(f)$, $y \in \mathcal{R}^n$.

The following lemma shows that the implication "a)⇒c)" in Theorem 1.3 can also be proved without the assumption that the function f is closed.

Lemma 4.2. *Let f be a convex function on $\mathcal{R}^n$, and let $-g$ be a polyhedral convex function on $\mathcal{R}^m$. Let us suppose that the program (P) has a strictly feasible solution: a point $x_0 \in \mathcal{R}^n$ such that $x_0 \in \operatorname{ri} F(f)$ and $Ax_0 - b \in F(g)$. Then, the optimal values of programs (P) and (D) are equal. Furthermore, the dual optimal value v_D is attained if it is finite.*

Proof. Let us denote by $(\overline{P})$ the program, which we obtain by replacing the functions f and g with their closures $\operatorname{cl} f$ and $\operatorname{cl} g = g$, that is let

$$(\overline{P}): \quad \text{Find} \quad \inf\{\operatorname{cl} f)(x) - g(Ax - b) + a^T x : x \in \mathcal{R}^n\}.$$

Then the dual of program $(\overline{P})$ is program (D). The point x_0 is also a strictly feasible solution of program $(\overline{P})$, so by Theorem 1.3 the optimal values of programs $(\overline{P})$ and (D) are equal, and the optimal value of program (D) is attained if it is finite.

We will show that the optimal values of programs (P) and $(\overline{P})$ are equal. It is obvious, that $v_{\overline{P}} \leq v_P$, as $\operatorname{cl} f \leq f$. On the other hand, for a given $\mu > v_{\overline{P}}$, let x_1 be a feasible solution of program $(\overline{P})$ with corresponding value

$$\mu_1 := (\operatorname{cl} f)(x_1) - g(Ax_1 - b) + a^T x_1 < \mu.$$

Then, for $0 \leq \lambda < 1$ the point $x_\lambda := \lambda x_1 + (1 - \lambda)x_0$ is a strictly feasible solution of program (P). Moreover, by Lemma 4.1,

$$f(x_\lambda) \to (\operatorname{cl} f)(x_1),\ g(Ax_\lambda - b) \to g(Ax_1 - b)\ (0 \leq \lambda < 1,\ \lambda \to 1).$$

Consequently, we have for all $\mu > v_{\overline{P}}$,

$$v_{\overline{P}} \leq v_P \leq f(x_\lambda) - g(Ax_\lambda - b) + a^T x_\lambda \to \mu_1 < \mu\ (0 \leq \lambda < 1,\ \lambda \to 1).$$

Thus $v_P = v_{\overline{P}}$, which proves the statement. □

Now, we describe the definition of the generalized Lagrangian programs.

Let $C \subseteq \mathcal{R}^n$ be a convex set, and let $P \subseteq \mathcal{R}^n$ be a polyhedron. Let $K \subseteq \mathcal{R}^m$ be a convex cone, and let $R \subseteq \mathcal{R}^l$ be a polyhedral cone. Let $\tilde{f} : C \to \mathcal{R}$ be a convex function, and let $\tilde{p} : P \to \mathcal{R}$ be a polyhedral convex function. Let $\tilde{g} : C \to \mathcal{R}^m$ be a K-convex mapping, and let $\tilde{h} : P \to \mathcal{R}^l$ be an R-polyhedral mapping. (A mapping $\tilde{g} : C \to \mathcal{R}^m$ is *K-convex*, if the epigraph

$$[\tilde{g}]_K := \{(x, y) \in \mathcal{R}^n \times \mathcal{R}^m : x \in C,\ \tilde{g}(x) \leq_K y\}$$

is convex. A mapping $\tilde{h} : P \to \mathcal{R}^l$ is *R-polyhedral*, if the epigraph $[\tilde{h}]_R$ is a polyhedron. For example, every affine mapping is R-polyhedral. Here $x \leq_K y$ denotes that $y - x \in K$. Note that if $K \subseteq \mathcal{R}^m$ is a closed convex cone, and pointed also – that is, $K \cap -K = \{0\}$ holds –, then $x \leq_K y$ is the *cone-generated partial order* on $\mathcal{R}^m$. However, in what follows we do not assume closedness and pointedness of the convex cone K.)

Let us consider the following program pair:

$$(LP): \quad \text{Find} \quad \inf\{\tilde{f}(x) + \tilde{p}(x) : \tilde{g}(x) \leq_K 0,\ \tilde{h}(x) \leq_R 0,\ x \in C \cap P\},$$
$$(LD): \quad \text{Find} \quad \sup\{\inf\{(\tilde{f} + \tilde{p} + y^T\tilde{g} + z^T\tilde{h})(x) : x \in C \cap P\} : y \in K^*,\ z \in R^*\},$$

where K^* denotes the dual cone of K, that is $K^* := \{y : y^T x \geq 0\ (x \in K)\}$.

The program (LP) is equivalent to the following program $(\hat{P})$:

$$(\hat{P}): \quad \text{Find} \quad \inf\{\hat{f}(\hat{x}) - \hat{g}(\hat{x}) : \hat{x} = (x, b_1, b_2, b_3, b_4)\}.$$

Here

$$\hat{f}(\hat{x}) := \begin{cases} \tilde{f}(x), & \text{if } \hat{x} \in \hat{C}_1, \\ \infty & \text{otherwise,} \end{cases}$$

$$\hat{g}(\hat{x}) := \begin{cases} -\tilde{p}(x), & \text{if } \hat{x} \in \hat{C}_2, \\ -\infty & \text{otherwise,} \end{cases}$$

where

$$\begin{aligned} \hat{C}_1 &:= \{\hat{x} : x \in C,\ \tilde{g}(x) + b_1 \leq_K 0,\ b_2 = b_4,\ b_3 \in K\}, \\ \hat{C}_2 &:= \{\hat{x} : x \in P,\ \tilde{h}(x) + b_2 \leq_R 0,\ b_1 = b_3,\ b_4 \in R\}. \end{aligned}$$

Note that due to our assumptions on the defining functions and mappings, $\hat{f}$ is a convex function, $-\hat{g}$ is a polyhedral convex function, finite on the convex set $\hat{C}_1$ and the polyhedron $\hat{C}_2$, respectively.

The dual of the program $(\hat{P})$ is

$$(\hat{D}): \quad \text{Find} \quad \sup\{\hat{g}^c(\hat{y}) - \hat{f}^c(\hat{y}) : \hat{y} = (a_1, y_1, y_2, y_3, y_4)\}.$$

It can be easily seen, that

$$\hat{g}^c(\hat{y}) = \begin{cases} \inf\{a_1^T x + \tilde{p}(x) + y_2^T b_2 : x \in P,\ \tilde{h}(x) + b_2 \leq_R 0\}, \\ \qquad \text{if } y_1 = -y_3,\ y_4 \in R^*, \\ -\infty \qquad \text{otherwise,} \end{cases}$$

and similarly

$$\hat{f}^c(\hat{y}) = \begin{cases} \sup\{a_1^T x - \tilde{f}(x) + y_1^T b_1 : x \in C,\ \tilde{g}(x) + b_1 \leq_K 0\}, \\ \qquad \text{if } y_2 = -y_4,\ y_3 \in -K^*, \\ \infty \qquad \text{otherwise.} \end{cases}$$

Hence,

$$\begin{aligned} &\hat{g}^c(\hat{y}) - \hat{f}^c(\hat{y}) = \\ &= \begin{cases} \inf\{a_1^T x + \tilde{p}(x) + y_2^T b_2 : x \in P,\ \tilde{h}(x) + b_2 \leq_R 0\} + \\ \qquad + \inf\{-a_1^T x + \tilde{f}(x) + y_3^T b_1 : x \in C,\ \tilde{g}(x) + b_1 \leq_K 0\}, \\ \qquad \text{if } -y_3 = y_1 \in K^*,\ -y_2 = y_4 \in R^*, \\ -\infty \qquad \text{otherwise} \end{cases} \\ &= \begin{cases} \inf\{a_1^T x + \tilde{p}(x) + y_4^T \tilde{h}(x) : x \in P\} + \\ \qquad + \inf\{-a_1^T x + \tilde{f}(x) + y_1^T \tilde{g}(x) : x \in C\}, \\ \qquad \text{if } -y_3 = y_1 \in K^*,\ -y_2 = y_4 \in R^*, \\ -\infty \qquad \text{otherwise.} \end{cases} \end{aligned}$$

We can see that the program (LD) is a relaxation of the program $(\hat{D})$: if the vector $\hat{y}$ is a feasible solution of the program $(\hat{D})$ then $y := y_1$, $z := y_4$ is a feasible solution of the program (LD), for which between the corresponding values the inequality

$$\hat{g}^c(\hat{y}) - \hat{f}^c(\hat{y}) \leq \inf\{(\tilde{f} + \tilde{p} + y^T\tilde{g} + z^T\tilde{h})(x) : x \in C \cap P\}$$

holds.

From these considerations immediately follows

Lemma 4.3. *For the optimal values of the programs (LP), (LD), $(\hat{P})$, and $(\hat{D})$ defined above, the following statements hold:*

a) $v_{\hat{P}} = v_{LP} \geq v_{LD} \geq v_{\hat{D}}$ (weak duality),

b) if $v_{\hat{P}} = v_{\hat{D}}$, then $v_{LP} = v_{LD}$,

c) if $v_{\hat{P}} = v_{\hat{D}}$ and the optimal value of the program $(\hat{D})$ is attained, then the optimal value of program (LD) is attained as well. □

Now, we can state our strong duality result. The program (LP) is said to satisfy the *weak Slater condition* if there exists a point $x_0 \in \mathcal{R}^n$ such that

$$x_0 \in P \cap \operatorname{ri} C,\ \tilde{g}(x_0) <_K 0,\ \tilde{h}(x_0) \leq_R 0.$$

Then x_0 is called a *weak Slater point.* (Here $x <_K y$ denotes that $y - x \in \operatorname{ri} K$.)

Theorem 4.1. *Let us suppose that the program (LP) satisfies the weak Slater condition. Then the optimal values of programs (LP) and (LD) are equal. Furthermore, the dual optimal value v_{LD} is attained if it is finite.*

Proof. It is proved in [1] (see Theorem 2.3) that

$$\operatorname{ri}\{(x, b_1) : x \in C,\ \tilde{g}(x) + b_1 \leq_K 0\} = \{(x, b_1) : x \in \operatorname{ri} C,\ \tilde{g}(x) + b_1 <_K 0\}.$$

Consequently,

$$\operatorname{ri} \hat{C}_1 = \{\hat{x} : x \in \operatorname{ri} C,\ \tilde{g}(x) + b_1 <_K 0,\ b_2 = b_4,\ b_3 \in \operatorname{ri} K\},$$

and we can see that

$$\hat{x}_0 := (x_0, -\tilde{g}(x_0)/2, -\tilde{h}(x_0), -\tilde{g}(x_0)/2, -\tilde{h}(x_0)) \in (\operatorname{ri} \hat{C}_1) \cap \hat{C}_2$$

for any weak Slater point x_0 of the program (LP). Hence, $\hat{x}_0$ is a strictly feasible solution of program $(\hat{P})$, and we can apply Lemma 4.2 to the programs $(\hat{P})$ and $(\hat{D})$. We obtain that $v_{\hat{P}} = v_{\hat{D}}$, and that the optimal value of the program $(\hat{D})$ is attained if it is finite. The statement now follows from Lemma 4.3. □

We remark that an analogue of Corollary 4.1 in [2], for programs (LP) and (LD), can be derived as a consequence of Theorem 4.1: the existence of a weak Slater point x_0 and a primal optimal solution $\overline{x}$ implies the existence of a saddle point $(\overline{x}, \overline{y}, \overline{z})$

of the Lagrangian function. (The *Lagrangian function* $L : (C \cap P) \times K^* \times R^* \to \mathcal{R}$ is defined as

$$L(x, y, z) := \tilde{f}(x) + \tilde{p}(x) + y^T \tilde{g}(x) + z^T \tilde{h}(x).$$

A point $(\overline{x}, \overline{y}, \overline{z}) \in (C \cap P) \times K^* \times R^*$ is called a *saddle point* of the Lagrangian function L if

$$L(\overline{x}, y, z) \leq L(\overline{x}, \overline{y}, \overline{z}) \leq L(x, \overline{y}, \overline{z}),$$

for every $x \in C \cap P$, $y \in K^*$, $z \in R^*$.) The proof is an adaptation of the proof of Corollary 4.1 in [2], and is left to the reader.

Finally, we mention an open problem: Similarly as in the case of the weak Slater condition in Theorem 4.1 (sufficient for the strict solvability condition), find sufficient conditions for the stability and closedness conditions in the duality theorems 1.2, 1.3, and 3.2 for the special case of programs $(\hat{P})$ and $(\hat{D})$, which are formulated in terms of the data describing the programs (LP) and (LD).

Acknowledgements. I am indebted to Margit Kovács for the several consultations. I thank the two anonymous referees for their remarks that helped me to improve the presentation of the paper.

References

[1] Boţ, R.I., Grad, S.M., and Wanka, G. A new constraint qualification and conjugate duality for composed convex optimization problems. *Journal of Optimization Theory and Applications*, 135(2):241–255, 2007.

[2] Frenk, J.B.G., and Kassay, G. On classes of generalized convex functions, Gordan-Farkas type theorems, and Lagrangian duality. *Journal of Optimization Theory and Applications*, 102(2): 315–343, 1999.

[3] Pataki, G. On the closedness of the linear image of a closed convex cone. *Mathematics of Operations Research*, 32(2):395–412, 2007.

[4] Rockafellar, R.T. *Convex Analysis.* Princeton University Press, Princeton, 1970.

[5] Stoer, J., and Witzgall, C. *Convexity and Optimization in Finite Dimensions I.* Springer-Verlag, Berlin, 1970.

[6] Strang, G. *Linear Algebra and its Applications.* Academic Press, New York, 1980.

[7] Ujvári, M. On a closedness theorem. *Pure Mathematics and Applications*, 15(4):469–486, 2006.

[8] Ujvári, M. On Abrams' theorem. *Pure Mathematics and Applications*, 18(1-2):177–187, 2008.

[9] Wolkowicz, H. Some applications of optimization in matrix theory. *Linear Algebra and its Applications*, 40:101–118, 1981.

Received 10th July 2012

ACTA CYBERNETICA

Information for authors. Acta Cybernetica publishes only original papers in the field of Computer Science. Manuscripts must be written in good English. Contributions are accepted for review with the understanding that the same work has not been published elsewhere. Papers previously published in conference proceedings, digests, preprints are eligible for consideration provided that the author informs the Editor at the time of submission and that the papers have undergone substantial revision. If authors have used their own previously published material as a basis for a new submission, they are required to cite the previous work(s) and very clearly indicate how the new submission offers substantively novel or different contributions beyond those of the previously published work(s). Each submission is peer-reviewed by at least two referees. The length of the review process depends on many factors such as the availability of an Editor and the time it takes to locate qualified reviewers. Usually, a review process takes 6 months to be completed. There are no page charges. An electronic version of the puplished paper is provided for the authors in PDF format.

Manuscript Formatting Requirements. All submissions must include a title page with the following elements:

- title of the paper
- author name(s) and affiliation
- name, address and email of the corresponding author
- An abstract clearly stating the nature and significance of the paper. Abstracts must not include mathematical expressions or bibliographic references.

References should appear in a separate bibliography at the end of the paper, with items in alphabetical order referred to by numerals in square brackets. Please prepare your submission as one single PostScript or PDF file including all elements of the manuscript (title page, main text, illustrations, bibliography, etc.). Manuscripts must be submitted by email as a single attachment to either the most competent Editor, the Managing Editor, or the Editor-in-Chief. In addition, your email has to contain the information appearing on the title page as plain ASCII text. When your paper is accepted for publication, you will be asked to send the complete electronic version of your manuscript to the Managing Editor. For technical reasons we can only accept files in LaTeX format.

Subscription Information. Acta Cybernetica is published by the Institute of Informatics, University of Szeged, Hungary. Each volume consists of four issues, two issues are published in a calendar year. Subscription rates for one issue are as follows: 5000 Ft within Hungary, €40 outside Hungary. Special rates for distributors and bulk orders are available upon request from the publisher. Printed issues are delivered by surface mail in Europe, and by air mail to overseas countries. Claims for missing issues are accepted within six months from the publication date. Please address all requests to:

Acta Cybernetica, Institute of Informatics, University of Szeged
P.O. Box 652, H-6701 Szeged, Hungary
Tel: +36 62 546 396, Fax: +36 62 546 397, Email: acta@inf.u-szeged.hu

Web access. The above informations along with the contents of past issues are available at the Acta Cybernetica homepage http://www.inf.u-szeged.hu/actacybernetica/ .

CONTENTS

ISSN 0324—721 X

Felelős szerkesztő és kiadó: Csirik János
Nyomdai kivitelezés: E-press Nyomdaipari Kft.