Volume 14 Number 3

ACTA CYBERNETICA

Szeged, 2000

ACTA CYBERNETICA

Information for authors. Acta Cybernetica publishes only original papers in the field of Computer Science. Contributions are accepted for review with the understanding that the same work has not been published elsewhere.

Manuscripts must be in English and should be sent in triplicate to any of the Editors. On the first page, the *title* of the paper, the *name(s)* and *affiliation(s)*, together with the *mailing* and *electronic address(es)* of the author(s) must appear. An *abstract* summarizing the results of the paper is also required. References should be listed in alphabetical order at the end of the paper in the form which can be seen in any article already published in the journal. Manuscripts are expected to be made with a great care. If typewritten, they should be typed double-spaced on one side of each sheet. Authors are encouraged to use any available dialect of TEX.

After acceptance, the authors will be asked to send the manuscript's source TEX file, if any, on a diskette to the Managing Editor. Having the TEX file of the paper can speed up the process of the publication considerably. Authors of accepted contributions may be asked to send the original drawings or computer outputs of figures appearing in the paper. In order to make a photographic reproduction possible, drawings of such figures should be on separate sheets, in India ink, and carefully lettered.

There are no page charges. Fifty reprints are supplied for each article published.

Publication information. Acta Cybernetica (ISSN 0324–721X) is published by the Department of Informatics of the University of Szeged, Szeged, Hungary. Each volume consists of four issues, two issues are published in a calendar year. For 2000 Numbers 3–4 of Volume 14 are scheduled. Subscription prices are available upon request from the publisher. Issues are sent normally by surface mail except to overseas countries where air delivery is ensured. Claims for missing issues are accepted within six months of our publication date. Please address all requests for subscription information to: Department of Informatics, University of Szeged, H-6701 Szeged, P.O.Box 652, Hungary. Tel.: (36)-(62)-420-184, Fax:(36)-(62)-420-292.

URL access. All these information and the contents of the last some issues are available in the Acta Cybernetica home page at http://www.inf.u-szeged.hu/local/acta.

Acta Cybernetica **14** (2000) 419-453.

Hybrid Concurrency Control and Recovery for Multi-Level Transactions

Klaus-Dieter Schewe [*] Torsten Ripke [*] Sven Drechsler [*]

Abstract

Multi-level transaction schedulers adapt conflict-serializability on different levels. They exploit the fact that many low-level conflicts (e.g. on the level of pages) become irrelevant, if higher-level application semantics is taken into account. Multi-level transactions may lead to an increase in concurrency.

It is easy to generalize locking protocols to the case of multi-level transactions. In this, however, the possibility of deadlocks may diminish the increase in concurrency. This stimulates the investigation of optimistic or hybrid approaches to concurrency control.

Until now no hybrid concurrency control protocol for multi-level transactions has been published. The new FoPL protocol (**F**orward **o**riented Concurrency Control with **P**reordered **L**ocking) is such a protocol. It employs access lists on the database objects and forward oriented commit validation. The basic test on all levels is based on the reordering of the access lists. When combined with queueing and deadlock detection, the protocol is not only sound, but also complete for multi-level serializable schedules. This is definitely an advantage of FoPL compared with locking protocols. The complexity of deadlock detection is not crucial, since waiting transactions do not hold locks on database objects. Furthermore, the basic FoPL protocol can be optimized in various ways.

Since the concurrency control protocol may force transactions to be aborted, it is necessary to support operation logging. It is shown that as well as multi-level locking protocols can be easily coupled with the ARIES algorithms. This also solves the problem of rollback during normal processing and crash recovery.

Keywords. [H2.4] Transaction Processing, Concurrency [H2.7] Logging and Recovery

General Terms. Algorithms, Reliability

[*]Computer Science Institute, Clausthal Technical University, Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, FRG [schewe|ripke|sdrechsl}@informatik.tu-clausthal.de

1 Introduction

One of the major intentions underlying the development of database systems was data sharing. As a consequence user programs must be realized as atomic units, which leads to the well-known notion of a *transaction*. Roughly spoken a transaction is the sequence of database operations resulting from program execution. Although these sequences must be interleaved to achieve acceptable performance, the effect must be the same as if transactions were executed sequentially.

Transaction throughput is a crucial issue for all databases. The common approach in practice considers conflict-serializable schedules, where conflicts correspond to read- and write-operations on database objects [8, 20]. No matter which granularity is taken for these objects – pages, records or even relations occur in practice – this approach rules out acceptable, but formally not serializable schedules.

In order to increase the rate of concurrency *multi-level transactions* (as a special form of nested transactions) have been introduced. They already occurred in System/R supporting both short-time locking on pages and locking on records [17]. A general theory of multi-level transactions has been developed in [1] and extended to a discussion of suitable protocols in [23, 24]. The basic idea of multi-level conflict-serializability is that sequences of low-level, e.g. page-level, database operations represent application-dependent operations on higher levels, and there are usually less conflicts on higher levels. Consequently, some of the conflicts on lower levels may be ignored. We shall present the gist of the multi-level transaction model in Section 2. In this context we also extend some notions of basic serializability theory to the case of multi-level transactions. These notions consider recoverability, cascade-freeness and strictness.

In distributed databases multi-level transactions occur naturally [4, 19]. E.g., in distributed object bases we may think of a global level, a local logical object level, a local level of physical objects and a page level. This is the view adopted in the DOMOCC project currently under investigation at Clausthal.

The general approach to concurrency control is the use of locking protocols, especially two-phase locking [20]. It will be shown how to generalize lock protocols to multi-level transactions. This will fill Section 3. The major problems with this approach are transaction throughput and the possibility of deadlocks due to transactions waiting for each other to release locks. There are several algorithms for deadlock detection in distributed databases with non-negligible complexity, e.g. [5, 18]. In addition, in interactive systems or applications with long-term transactions, waiting for the release of any lock may be not acceptable.

Therefore, alternatives to locking protocols dominate the research in concurrency control. The solutions comprise timestamp protocols [13, 14], optimistic protocols [2, 7, 9, 12] and hybrid protocols [3, 10, 11] combining at least two of the other approaches. Unfortunately, none of the existing optimistic or hybrid concurrency control protocols has been generalized to multi-level transactions so far. For example, the *optimistic dummy lock* (ODL) protocol [11] is basically organized as an optimistic scheduler using read/write-labels instead of locking objects.

Then certification tests for the existence of these labels and the final write phase locks objects to be updated. Unfortunately, a direct generalization to multi-level transactions is not possible.

In this paper we present a new hybrid protocol called FoPL (*forward oriented concurrency control protocol with preordered locking*), which is a provably correct protocol for multi-level transactions [21]. FoPL exploits that multi-level schedulers can be composed from schedulers for each of the involved levels [23, 24]. Then the ODL idea is refined such that access lists are defined for all such levels. More precisely, labels are kept in a list according to the time points when they have been set. Commit handling then requires the labels of a validating transaction to be shifted to the head of the list. In contrast to ODL the new FoPL protocol will use forward oriented validation. FoPL will be presented in detail in Section 4.

When combined with queueing and deadlock detection, the protocol is not only sound, but also complete for multi-level serializable schedules. This is definitely an advantage of FoPL compared with locking protocols. The complexity of deadlock detection is not crucial, since waiting transactions do not hold locks on database objects.

Given the basic FoPL protocol we are able to discuss several optimizations. These comprise a more optimistic locking strategy, the processing of earlier or partial rollbacks, and specific capabilities related to absorption. Section 5 is devoted to the discussion of these extensions.

In this context we start with initial considerations concerning the comparison of FoPL with locking protocols. We focus on implementation costs and transaction throughput. This will be done in Section 6.

Since the concurrency control protocol may force transactions to be aborted, it is necessary to support operation logging. For this the sophisticated ARIES algorithms [16, 22] are generally accepted as a good starting point. We show how to extend the algorithms to multi-level transactions, both for locking protocols and FoPL. This also solves the problem of rollback during normal processing and crash recovery. The extension called ARIES/ML [6] also enhances the work by Lomet [15]. The solution to recovery will be presented in Section 7. We conclude with a short summary.

2 The Multi-Level Transaction Model

A multi-level transaction is a special kind of an open nested transaction, where the leaves in the transaction tree have the same depth. Each node in the tree corresponds to some operation implemented by its successors. The root is a transaction. The lowest level L_0 corresponds to operations that access directly the physical database. Therefore, we first define the operations of a multi-level system.

Definition 1 An *n-level-system* $\mathfrak{L}$ consists of n levels $L_i = (\mathfrak{D}_i, \mathfrak{F}_i)$ $(i = 0, \dots, n-1)$, where $\mathfrak{D}_i$ is a set of *objects* and $\mathfrak{F}_i$ a set of *operators*. An *L_i-operation* is an element of $\mathcal{O}_i = \mathfrak{F}_i \times \mathfrak{D}_i$. □

We write $\mathfrak{L} = (L_{n-1}, \dots, L_0)$. The levels are numbered in a bottom-up manner.

Example 1 In the DOMOCC project at Clausthal Technical University we investigate distributed object bases. For these it is imaginable to use a 4-level-system. The highest level L_3 should correspond to global logical objects, the next lower level L_2 to local logical objects associated with a unique site, level L_1 to local physical objects, i.e. records, and finally L_0 should correspond to the page level.

Then operations on L_3 as defined before schema fragmentation will be implemented by operations on L_2, these again by operations on the record level L_1, which finally give rise to reading and writing pages of the physical store. $\square$

2.1 Multi-Level Transactions

An n-level transaction is defined next exploiting the notion of an *index tree*, which is a finite set of finite sequences over $\mathbb{N} - \{0\}$. We let $(\mathbb{N} - \{0\})^*$ denote the set of all such sequences. $\mid \alpha \mid$ denotes the length of $\alpha \in \mathbb{N}^*$. Furthermore, we identify numbers with sequences of length 1 and denote the empty sequence by ϵ.

As a syntactic convention we shall use small Greek letters $\alpha, \beta, \mu, \nu, \dots$ for such number sequences and small Latin letters $i, j, k, \ell, \dots$ for the numbers in these sequences.

Definition 2 An *index tree* of depth n is a finite subset $I \subseteq (\mathbb{N} - \{0\})^*$ with

- $\epsilon \in I$,
- $\alpha(k+1) \in I \Rightarrow \alpha k \in I$ and
- $\alpha \in I \wedge \mid \alpha \mid < n \Leftrightarrow \alpha 1 \in I$

for all $\alpha \in (\mathbb{N} - \{0\})^*$ and $k \in \mathbb{N}$.

An *n-level-transaction* T_j consists of

- an index tree I of depth n,
- a mapping which assigns to each $\alpha \in I$ an $L_{n-|\alpha|}$-operation, denoted as $o_{j\alpha}$ and
- partial orders $<_i^{(j)}$ on each $\mathfrak{O}_i^{(j)} = \{o_{j\alpha} \mid \mid \alpha \mid + i = n\}$, such that $o_{j\alpha k} <_i^{(j)} o_{j\beta\ell} \Rightarrow k < \ell$ holds.

We call $<_i^{(j)}$ the $L_i^{(j)}$-*precedence relation* of the transaction T_j. $\square$

By abuse of notation we shall talk of the transaction T_j over the index-tree I. Furthermore, we write $op_{j\alpha}(x)$ for the operation $o_{j\alpha} = (op, x)$. In order to have a uniform notation for all levels we also allow to write o_j for T_j.

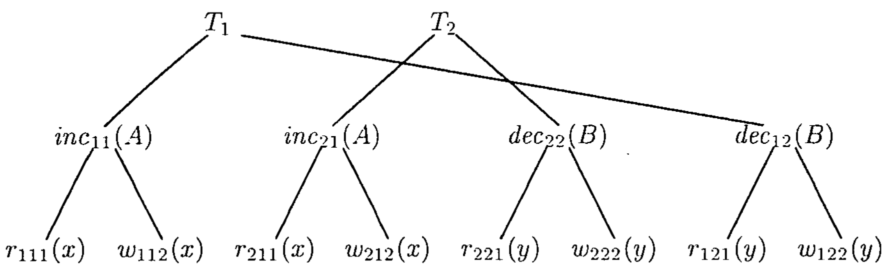


Figure 1: Serializable multi-level schedule

Since precedence relations are meant to express a necessary ordering of implementing operations it is natural to require

$$o_{j\alpha} <_i^{(j)} o_{j\beta} \Leftrightarrow o_{j\alpha k} <_{i-1}^{(j)} o_{j\beta\ell} \quad \text{for all } k \text{ and } \ell, \tag{1}$$

whenever the involved operations exist. In this case, the transaction T_j is *well-defined*. In the sequel we shall tacitly assume that all transactions are well-defined.

Example 2 The trees rooted at T_1 and T_2 in Figure 1 define two 2-level-transactions over the same index tree $I = \{\epsilon, 1, 2, 11, 12, 21, 22\}$. Here w and r correspond to read- and write-operations, *inc* and *dec* to incrementation and decrementation. Thus, we may assume $<_0^{(j)}$ to be defined by

$$r_{111}(x) <_0^{(1)} w_{112}(x) \qquad \text{and} \qquad r_{121}(y) <_0^{(1)} w_{122}(y)$$

and $<_1^{(1)}$ as being empty.

Analogously, define $<_0^{(2)}$ by

$$r_{211}(x) <_0^{(1)} w_{212}(x) \qquad \text{and} \qquad r_{221}(y) <_0^{(1)} w_{222}(y)$$

and let $<_1^{(2)}$ be empty.

However, if we claimed also $w_{112}(x) <_0^{(1)} r_{121}(y)$ – i.e., $<_0^{(1)}$ is total – then the well-definedness condition (1) would imply $inc_{11}(A) <_1^{(1)} dec_{12}(B)$. □

The edges in a transaction tree represent the implementation of a L_i-operation by a set of L_{i-1}-operations. If $o_{j\mu k}$ is an L_i-operation of a transaction T_j, then $trans(o_{j\mu k}) = o_{j\mu}$ $(0 \leq i < n)$ is the L_{i+1}-operation that invokes $o_{j\mu k}$. In particular, for $i = n-1$, i.e. μ is empty, we get $trans(o_{jk}) = T_j$. Conversely, $act(o_{j\nu}) = \{o_{j\nu\ell} \mid \nu\ell \in I\}$ defines the set of L_{i-1}-operations implementing the L_i-operation $o_{j\nu}$.

More generally, for $i' > i$ we may define iteratively the $L_{i'}$-operation that indirectly invokes an L_i-operation $o_{j\mu}$ by

$$trans_{i'}(o_{j\mu}) = trans^{i'-i}(o_{j\mu}) \quad . \tag{2}$$

Note that $i' = i+1$ leads to the direct predecessor in the transaction tree as defined by *trans*.

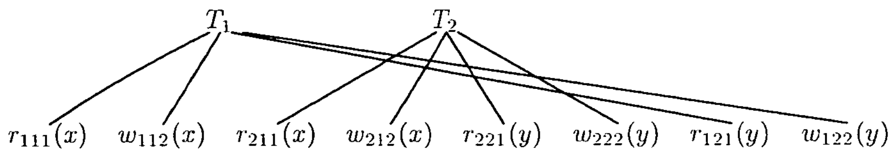

T_1
T_2
$r_{111}(x)$ $w_{112}(x)$ $r_{211}(x)$ $w_{212}(x)$ $r_{221}(y)$ $w_{222}(y)$ $r_{121}(y)$ $w_{122}(y)$

Figure 2: Non-serializable 1-level-schedule

Conversely, for an L_i-operation $o_{j\nu}$ let $act_{i-1}(o_{j\nu}) = act(o_{j\nu})$ and

$$act_{i'}(o_{j\nu}) = \bigcup_{o_{j\nu k} \in act(o_{j\nu})} act_{i'}(o_{j\nu k}) \qquad \text{for } i - i' > 1, \tag{3}$$

i.e. $act_{i'}(o_{j\nu})$ denotes the set of $L_{i'}$-operations implementing $o_{j\nu}$ indirectly through several levels.

Example 3 Consider again transaction T_1 in Figure 1. Here we have

$$\begin{aligned} act(T_1) = act_1(T_1) &= \{inc_{11}(A), dec_{12}(B)\} \quad , \\ act_0(T_1) &= \{r_{111}(x), w_{112}(x), r_{121}(y), w_{122}(y)\} \quad , \\ act(inc_{11}(A)) = act_0(inc_{11}(A)) &= \{r_{111}(x), w_{112}(x)\} \end{aligned}$$

and

$$\begin{aligned} trans(inc_{11}(A)) = trans_2(inc_{11}(A)) &= T_1 \quad , \\ trans(w_{112}(x)) = trans_1(w_{112}(x)) &= inc_{11}(A) \quad , \\ trans_2(w_{112}(x)) &= T_1 \quad . \end{aligned}$$

□

2.2 Multi-Level Schedules

The execution of concurrent transactions is described by an n-level-schedule. These are illustrated by forests in Figures 1 and 2.

Definition 3 For a set $\mathfrak{O}_n = \{T_1, \dots, T_k\}$ of n-level-transactions let $\mathfrak{O}_i = \bigcup_{j=1}^{k} \mathfrak{O}_i^{(j)}$ be the set of all L_i-operations in these transactions ($0 \leq i < n$). Then an *n-level-schedule* on $\mathfrak{O}_n$ is given by a partial order $<_0$ on $\mathfrak{O}_0$ containing all $L_0^{(j)}$-precedence relations. □

We write $S = (\mathfrak{O}_n, \mathfrak{O}_{n-1}, \dots, \mathfrak{O}_0, <_0)$ for such a schedule defined on $\mathfrak{O}_n$. Then $<_0$ induces a partial order $<_i$ on each level by

$$o_\mu <_{i+1} o_\nu \Leftrightarrow \forall o_{\mu k} \in act(o_\mu). \forall o_{\nu \ell} \in act(o_\nu).\ o_{\mu k} <_i o_{\nu \ell} \quad . \tag{4}$$

Using this, we may define the *level-by-level schedule* $S_{i,j}$ ($j < i \leq n$) as the one-level-schedule $(\mathfrak{O}_i, \mathfrak{O}_j, <_j)$.

Example 4 The schedule in Figure 2 is the level-by-level schedule $S_{2,0}$ of the one in Figure 1. We dispense with a discussion of how to reorganize the underlying index-trees. □

The well-definedness assumption for transactions implies two simple properties as shown in the next lemma. The first one was originally used in [24] to define the partial order $<_i$ on level L_i. The second property is the plausible *conformity-*condition from [21]. Informally, it states that whenever two operations in some transaction have to occur in a certain order, then they must do so in every schedule.

Lemma 1 *1. For any two L_i-operations o_μ, o_ν in a n-level-schedule S we have*

$$o_\mu <_i o_\nu \Leftrightarrow \forall o_{\mu\varrho} \in act_0(o_\mu).\forall o_{\nu\sigma} \in act_0(o_\nu).\ o_{\mu\varrho} <_0 o_{\nu\sigma} \quad . \tag{5}$$

2. For each n-level-schedule S we have $<_i^{(j)} \subseteq <_i$ for all i and j.

Proof. For the proof of (i) we proceed by induction on i. For $i = 1$ the claimed equivalence in (5) is just the definition (4). For $i > 1$ we have

$$o_\mu <_i o_\nu \Leftrightarrow \forall o_{\mu k} \in act_{i-1}(o_\mu).\forall o_{\nu\ell} \in act_{i-1}(o_\nu).\ o_{\mu k} <_{i-1} o_{\nu\ell}$$

by definition (4) and

$$o_{\mu k} <_{i-1} o_{\nu\ell} \Leftrightarrow \forall o_{\mu k\varrho} \in act_0(o_{\mu k}).\forall o_{\nu\ell\sigma} \in act_0(o_{\nu\ell}).\ o_{\mu k\varrho} <_0 o_{\nu\ell\sigma}$$

by the induction hypothesis. Taking both equivalences together, the claimed statement (5) follows from the definition (3) of act_0.

For the proof of (ii) we also apply induction on i, the case $i = 0$ being captured by Definition 3. For $i > 0$ and $o_{j\mu} <_i^{(j)} o_{j\nu}$ the well-definedness condition (1) implies $o_{j\mu k} <_{i-1}^{(j)} o_{j\nu\ell}$ for all $o_{j\mu k} \in act(o_{j\mu})$, $o_{j\nu\ell} \in act(o_{j\nu})$. By induction hypothesis we get $o_{j\mu k} <_{i-1} o_{j\nu\ell}$. Hence, the claimed result $o_{j\mu} <_i o_{j\nu}$ follows from the definition of $<_i$ in (4). □

2.3 Partial Schedules

The notion of n-level-schedule describes the interleaved execution of n-level-transactions. Temporal precedence on level L_i is expressed by the partial order $<_i$. Since transactions are built at run-time, we are also interested in *partial schedules*, where some of the later operations are omitted. These will be composed from *n-level-prefixes* of transactions in the same way, as (complete) schedules are composed from transactions.

Definition 4 Let T_j be an n-level-transaction. An *(n-level-)prefix* of T_j consists of subsets $\mathfrak{P}_i^{(j)} \subseteq \mathfrak{O}_i^{(j)}$ $(i = 0, \dots, n)$ such that

- $o_{j\alpha} <_i^{(j)} o_{j\beta} \wedge o_{j\beta} \in \mathfrak{P}_i^{(j)} \Rightarrow o_{j\alpha} \in \mathfrak{P}_i^{(j)}$ and

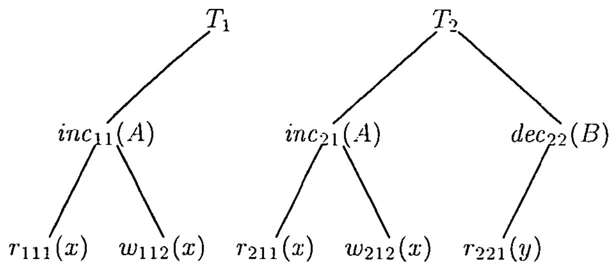


Figure 3: Partial 2-level-schedule

- $o_{j\alpha} \in \mathfrak{P}_i^{(j)} \Rightarrow trans(o_{j\alpha}) \in \mathfrak{P}_{i+1}^{(j)}$

hold, whenever the involved operations are defined. □

Formally, a prefix is different from a transaction unless we have $\mathfrak{P}_i^{(j)} = \mathfrak{O}_i^{(j)}$ for all i. On the other hand the selection of subsets for a prefix also defines an underlying subtree of the index-tree. Therefore, we may treat prefixes, as if they were (complete) transactions. In particular, we also have precedence relations $<_i^{(j)}$ on prefixes which result from restricting the corresponding relations associated with the transaction.

Furthermore, we may define schedules on the basis of prefixes using Definition 3. In this case we talk of a *partial n-level-schedule* and write $(\mathfrak{P}_n, \dots, \mathfrak{P}_0, <_0)$ for this. Here $\mathfrak{P}_n = \{P_1, \dots, P_k\}$ is a set of n-level-prefixes, $\mathfrak{P}_i = \bigcup_{j=1}^{k} \mathfrak{P}_i^{(j)}$ and $<_0$ is a partial order on $\mathfrak{P}_0$ containing all $L_0^{(j)}$-precedence relations restricted to $\mathfrak{P}_0$.

If all P_j are transactions, i.e. $P_j = T_j$, then we talk of a *complete* schedule.

Example 5 Figure 3 shows a partial schedule, where the tree rooted at T_j is a prefix of the 2-level-transaction T_j in Figure 1 ($j = 1, 2$). □

It is easy to see that each partial schedule can always be extended to a complete schedule by simply extending $<_0$ in some way compatible with the required extension of the L_0-precedence relations.

Conversely, given a complete schedule $(\mathfrak{O}_n, \dots, \mathfrak{O}_0, <_0)$, we may choose a subset $\mathfrak{P}_0 \subseteq \mathfrak{O}_0$ such that $o_\alpha <_0 o_\beta$ with $o_\beta \in \mathfrak{P}_0$ implies $o_\alpha \in \mathfrak{P}_0$. Then $\mathfrak{P}_0$ induces a canonical partial schedule $(\mathfrak{P}_n, \dots, \mathfrak{P}_0, <_0 |_{\mathfrak{P}_0})$. Such a partial schedule will be called a *prefix* of the given complete n-level-schedule. In this way partial schedules describe the interleaving of transactions in progress.

2.4 Conflict Serializability

The basic idea of multi-level concurrency control is to use the semantics of operations in level-specific, symmetric *conflict relations* $CON_i \subseteq O_i \times O_i$. Non-conflicting operations should commute. In particular, it is natural to assume that conflicts can only occur on the same object, i.e. $((op_1, x), (op_2, y)) \in CON_i \Rightarrow x = y$.

Same as with precedence relations the intention behind the conflict relations forces us to require the following *conformity condition*: If $(o_\mu, o_\nu) \in CON_i$ holds for some $o_\mu, o_\nu \in \mathfrak{O}_i$, then there should exist $o_{\mu k} \in act(o_\mu)$ and $o_{\nu \ell} \in act(o_\nu)$ with $(o_{\mu k}, o_{\nu \ell}) \in CON_{i-1}$. The fundamental idea of multi-level transactions is that there may be low-level conflicts that do not stem from higher-level conflicts. Thus, the opposite of this condition need not to hold. In the sequel we shall tacitly assume that the conformity condition is satisfied by all schedules.

Example 6 Increments and decrements commute with one another. Therefore, for the transactions in Figure 1 we would like to use

$$((op_1, x), (op_2, y)) \in CON_1 \Leftrightarrow (op_1 = upd \vee op_2 = upd) \wedge x = y$$

assuming $\mathfrak{F}_1 = \{inc, dec, upd\}$. Analogously,

$$((op_1, x), (op_2, y)) \in CON_0 \Leftrightarrow (op_1 = w \vee op_2 = w) \wedge x = y$$

assuming $\mathfrak{F}_0 = \{r, w\}$. Note that the L_0-conflict relation is the usual one used for flat transactions.

Intuitively, the schedule in Figure 1 seems to be acceptable, but the level-by-level schedule $S_{2,0}$ in Figure 2 is not. The reason is that by omitting the L_1-operations we lost the information that the schedule is equivalent to the sequence $T_1; T_2$. Otherwise said, there are no conflicts on level L_1. Thus, multi-level transactions may be expected to increase concurrency, which will be made explicit in the following. □

We have to extend the notion of conflict-serializability to multi-level transactions to make these arguments rigorous. First, an n-level-schedule with a total order $<_n$ is called *serial*. Then *serializability* means equivalence to a serial schedule in the following formal sense.

Definition 5 Let $(\mathfrak{O}_n, \mathfrak{O}_{n-1}, \dots, \mathfrak{O}_0, <_0)$ be an n-level-schedule with induced partial orders $<_i$ on level i. Let CON_i $(i = 0, \dots, n-1)$ be conflict relations. Define

$$o_{j\mu} \rightarrow_i o_{j'\nu} \Leftrightarrow j \neq j' \wedge (o_{j\mu}, o_{j'\nu}) \in CON_i \wedge o_{j\mu} <_i o_{j'\nu} \qquad (6)$$

for $o_\mu, o_\nu \in \mathfrak{O}_i$.

Then two n-level-schedules are called *(conflict-)equivalent* iff their associated relations $\rightarrow_i$ coincide for all $i = 0, \dots, n-1$. An n-level schedule which is conflict-equivalent to a serial one, is called *(n-level-)serializable*. □

From the early studies of multi-level transactions [1, 24] it is well known that n-level-serializability can be detected from the level-by-level schedules $S_{i,i-1}$.

Lemma 2 *An n-level-schedule S is n-level-serializable iff all its level-by-level schedules $S_{i,i-1}$ $(0 < i \leq n)$ are serializable.* □

It is opportune to add a remark on partial schedules here. We shall call a partial schedule *serializable* iff it can be extended to a complete serializable schedule.

Example 7 Using the conflict relations from Example 6 it is easily verified that the schedule in Figure 1 is conflict serializable, whereas the one in Figure 2 is not. This was already stated above.

The partial schedule in Figure 3 can be extended to the one in Figure 1, hence is also serializable. □

Note that transactions in a serial schedule may leave the system and need not be considered any more. Serializability implies that transactions – not prefixes – may leave the system, if they can be brought into the first position in an equivalent serial schedule.

2.5 Recoverable Schedules

One desirable property of schedules for the flat transaction model was *recoverability*. Informally, this means that committed transactions should never be rolled back later. This can be expressed by the fact that if a transaction T_j reads from another transaction T_i, i.e. $w_{ik_1}(x) \rightarrow_0 r_{jk_2}(x)$ holds for some L_0-object x and suitable indices k_1, k_2, then whenever T_j commits, T_i must do so, too. In order to guarantee this property the commit of T_i must occur before the commit of T_j.

In order to generalize these notions to multi-level transactions, we first consider the read-from-relation. $w_{ik_1}(x) \rightarrow_0 r_{jk_2}(x)$ represents a strong conflict in the sense that an *abort-dependency* is implied: if T_i aborts, then T_j must do so, too. It is not sufficient to consider just the associated relations $\rightarrow_i$. For example, we could also have $r_{ik_1}(x) \rightarrow_0 w_{jk_2}(x)$ without abort-dependency. Hence, T_j may commit before T_i. If accidentally T_i aborts later on, this will not influence T_j anymore.

The difference between these two situations cannot be explained without regarding the "effects" of the operations. Roughly spoken, an object x on any level L_i has a value, say $\sigma(x)$ before the execution of an L_i-operation $op_\alpha(x)$ and a value $\tau(x)$ after that execution. The effect of the operation can therefore be expressed by the set $\{-\sigma(x), +\tau(x)\}$ or by $\emptyset$ in the case we have $\sigma(x) = \tau(x)$.

Now note that in our motivating example $w_{ik_1}(x) \rightarrow_0 r_{jk_2}(x)$ for L_0-operations the effect of the sequence $w_{ik_1}(x); r_{jk_2}(x)$ differs from the effect of $r_{jk_2}(x)$, whereas for $r_{ik_1}(x) \rightarrow_0 w_{jk_2}(x)$ the effects of the sequence $r_{ik_1}(x); w_{jk_2}(x)$ and of $w_{jk_2}(x)$ coincide. We now take this observation as a cornerstone for the generalization of recoverability on level L_i.

Definition 6 Let $\mathfrak{L} = ((\mathfrak{D}_{n-1}, \mathfrak{F}_{n-1}), \dots, (\mathfrak{D}_0, \mathfrak{F}_0))$ be an *n-level-system* and assume sets V_i of values for each level L_i $(i = 0, \dots, n-1)$. A *state* of an L_i-object $x \in \mathfrak{D}_i$ is an element $\sigma(x) \in V_i$. An *effect* on an L_i-object $x \in \mathfrak{D}_i$ is either a set $\{-\sigma(x), \tau(x)\}$, where $\sigma(x)$ and $\tau(x)$ are different states of x, or $\emptyset$. □

Now, we may assume that each L_i-operation $op_\alpha(x)$ – more generally: each sequence of L_i-operations on the same object x – has an effect on x. Of course, this effect depends on the content of the database. With these initial remarks we can now generalize the read-from-relation.

Definition 7 Let T_j, $T_{j'}$ be two n-level-transactions ($j \neq j'$) and $o_{j\mu}(x)$, $o_{j'\nu}(x)$ be two of their L_i-operations. We say that $o_{j'\nu}(x)$ *strongly depends* on $o_{j\mu}(x)$ (notation: $o_{j\mu}(x) \twoheadrightarrow_i o_{j'\nu}(x)$) iff $o_{j\mu}(x) \rightarrow_i o_{j'\nu}(x)$ holds and the effect of the sequence $o_{j\mu}(x); o_{j'\nu}(x)$ differs from the effect of $o_{j'\nu}(x)$. □

Note that for the flat transaction model the chosen definition turns only write-read-conflicts into strong dependencies – as desired.

Example 8 Consider L_1-operators *upd* for update, *inc* and *dec* for increment and decrement and a read-only operator $fetch$. Then again, we have $upd_{ik_1}(A) \twoheadrightarrow_1 fetch_{jk_2}(A)$, but $fetch_{ik_1}(A) \not\twoheadrightarrow_1 inc_{jk_2}(A)$. □

The second task is to generalize the abort-dependency resulting from $o_{j\mu}(x) \twoheadrightarrow_i o_{j'\nu}(x)$. For this we may assume that each operation in a partial schedule may abort or commit. This can be expressed by marking the operations in a partial schedule by c or a, respectively. Let $m(o)$ be the marking of the operation o. If we consider transactions in progress, it may happen that some operation which implements o has not yet been committed nor aborted. In this case we cannot assign a mark to o, which turns a marking m into a partial mapping.

Furthermore, all operations that implement an operation o, i.e. all operations $o' \in act(o)$, must commit before o can commit. Formally, this can be expressed by $m(o) = c \Rightarrow m(o') = c$. Analogously, all operations o' that must preced o, expressed by the precedence relation $o' <_i^{(j)} o$, must commit before o. This leads to the following definition.

Definition 8 Let $S = (\mathfrak{P}_n, \ldots, \mathfrak{P}_0, <_0)$ be a partial schedule. A *marking* of S is a partial mapping $m : \bigcup_{i=0}^{n} \mathfrak{P}_i \nrightarrow \{c, a\}$ such that the following holds:

1. If $(o) \subseteq \mathfrak{P}_{i-1}$ holds for $o \in \mathfrak{P}_i$, then $m(o)$ must be defined.
2. Whenever $m(o) = c$ and $o' \in act(o)$ hold, $m(o')$ is also defined with $m(o') = c$. Whenever $m(o) = a$ holds, there must exist some $o' \in act(o)$ with $m(o') = a$.
3. Whenever $o' <_i o$ holds, then $m(o') = c$ must hold.

A pair (S, m) with a partial schedule S and a marking m of S will be called a *marked schedule*. □

The first condition simply restricts attention to marked schedules, in which all operations are marked if they can be marked. The second condition expresses the requirement that all operations that implement a committed operation must have

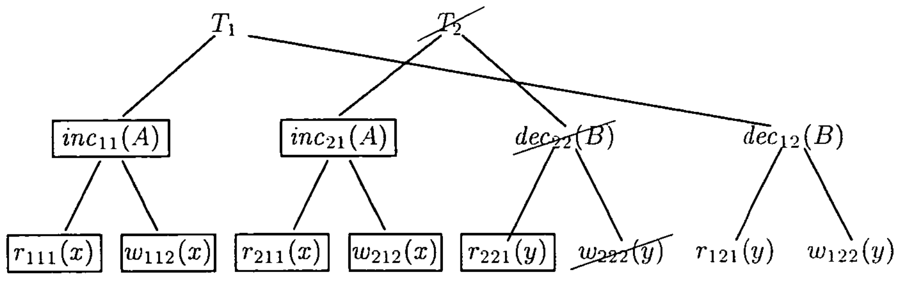


Figure 4: Marked multi-level schedule

been committed, too. Secondly, this condition expresses the analogue that among the operations that implement an aborted operation there must be at least one which has been aborted, too. The third condition expresses that if an operation has been completed before another one, it must have committed.

Example 9 Consider the marked schedule in Figure 4 with a boxed entry marking a committed operation and a crossed out operation marking an aborted one. The underlying schedule is the one from Figure 1. □

The notion of a marked schedule now allows recoverability to be generalized to multi-level schedules.

Definition 9 A schedule $(\mathfrak{O}_n, \dots, \mathfrak{O}_0, <_0)$ is called *recoverable on level L_i* iff for all prefixes S, all markings m of S and all $j \neq j'$

$$o_{j\mu k} \twoheadrightarrow_{i-1} o_{j'\nu\ell} \wedge m(o_{j'\nu}) = c \Rightarrow m(o_{j\mu}) = c$$

holds. □

Note that in contrast to recoverability for flat schedules recoverability on level L_i does not completely exclude committed operations from being rolled back later. However, the abort of a committed L_i-operation will only be triggered by the abort of $trans(o_\mu)$. We discuss recoverability together with the protocols presented in Sections 3 and 4.

Finally, we may also generalize the stronger notions of *cascade-freeness* and *strictness* to multi-level schedules.

Definition 10 Let $S = (\mathfrak{O}_n, \dots, \mathfrak{O}_0, <_0)$ be an n-level-schedule.

1. S is called *cascade-free on level L_i* iff for all prefixes S' of S, all markings m of S' and all $j \neq j'$ it is true that whenever $o_{j\mu k} \twoheadrightarrow_{i-1} o_{j'\nu\ell}$ holds, then $m(o_{j\mu})$ must be defined.

2. S is called *strict on level L_i* iff for all prefixes S' of S, all markings m of S' and all $j \neq j'$ it is true that whenever $o_{j\mu k} \rightarrow_{i-1} o_{j'\nu\ell}$ holds, then $m(o_{j\mu})$ must be defined. □

We shall discuss cascade-freeness and strictness together with the protocols presented in the next two sections. As a first result which is obvious from the definitions we notice that strictness implies cascade-freeness.

Example 10 Consider the schedule from Figure 1 with a total order $<_0$. Then we have the strong dependencies $w_{112}(x) \twoheadrightarrow_0 r_{211}(x)$ and $w_{222}(y) \twoheadrightarrow_0 r_{121}(y)$ on level L_0 and no such dependencies on level L_1. Obviously, in the marked schedule in Figure 4 the conditions for strictness, cascade-freeness and recoverability are satisfied for level L_1.

More generally, we can show that the schedule from Figure 1 is indeed recoverable on level L_1. If $w_{112}(x)$, $r_{211}(x)$ and $inc_{21}(A)$ with $m(inc_{21}(A)) = c$ (or $w_{222}(y)$, $r_{121}(y)$ and $dec_{12}(B)$ with $m(dec_{12}(B)) = c$, respectively) occur in a marked prefix, then $m(inc_{11}(A)) = c$ (or $m(dec_{22}(B)) = c$, respectively) must hold by the third condition in Definition 8.

We can also show that the schedule is cascade-free on level L_1. If we consider a prefix, in which $w_{112}(x)$ (or $w_{222}(y)$, respectively) occurs, then by the first condition in Definition 8 $m(inc_{11}(A))$ (or $m(dec_{22}(B))$, respectively) must be defined.

The same argument applies, if we consider $\rightarrow_0$, which gives

$$r_{111}(x) \rightarrow_0 w_{212}(x),\ w_{112}(x) \rightarrow_0 r_{211}(x),\ w_{112}(x) \rightarrow_0 w_{212}(x)$$

and

$$r_{221}(y) \rightarrow_0 w_{122}(y),\ w_{222}(y) \rightarrow_0 w_{121}(y),\ w_{222}(y) \rightarrow_0 w_{122}(y)\ .$$

This shows that the schedule is even strict on level L_1. □

3 Locking Protocols

Locking protocols for multi-level transactions have been investigated from the very beginning [24]. Therefore, we shall only describe very briefly the gist of these protocols.

According to our assumption that only those operations give rise to conflicts, which access the same object, it is sufficient to concentrate on the operators. Thus, for each L_i-operator $op \in \mathfrak{F}_i$ we define a specific lock $lock_{op}$. Then, each L_i-operation $op_{\mu k}(x)$ may only be executed after setting a lock, namely $lock_{op}$, on the object x. In addition we associate with this lock the index μ of the issuing operation $o_\mu = trans(o_{\mu k})$. After its commit, o_μ must release all its locks.

Same as with read-locks for flat transactions, an L_i-object x may hold several locks at a time, provided the associated operations do not conflict with each other.

Definition 11 Let $lock_{op_1}$ and $lock_{op_2}$ be locks on object $x \in \mathfrak{D}_i$ issued by the L_i-operations $o_{\mu k}$ and $o_{\nu \ell}$, respectively. These locks are called *incompatible* iff $o_{\mu k} \rightarrow_i o_{\nu \ell}$ or $o_{\nu \ell} \rightarrow_i o_{\mu k}$ holds. □

Thus, an operation may only set a lock on x, if this is not incompatible with any existing lock on x. Otherwise, the operation has to be aborted or must wait until all incompatible locks on x are released.

This basic idea underlying multi-level locking protocols can be extended in the usual way to define *two-phase locking* (2PL) as well as conservative or strict variants. In 2PL we have a *growing phase*, in which all locks are acquired, but none can be released, followed by a *shrinking phase* in which existing locks will be released, but no new lock can be acquired. In conservative 2PL (con-2PL) all locks are set before the operation actually starts. In strict 2PL (str-2PL) no lock will be released before commit or abort.

Example 11 Consider the schedule in Figure 1 assuming a total order $<_0$ from left to right. On level L_0 we have the usual read- and write-locks, i.e. $lock_r$ and $lock_w$ using our current notation. Only two read-locks are compatible with each other. Thus, all locks on L_0-objects can be set and released by 2PL without any problems.

On level L_1 we have locks $lock_{inc}$, $lock_{dec}$ and $lock_{upd}$ for the increment-, decrement- and general update-operation. Only the update-lock is incompatible to all other locks. Then, also all locks on L_1-objects can be set and released by 2PL. Hence, the schedule will be accepted by 2PL. □

The example indicates that schedules accepted by 2PL will be serializable. Such a result stating the correctness of 2PL for multi-level schedules is well-known from the early literature [24].

Theorem 1 *A multi-level-schedule accepted by the use of 2PL on each level is always serializable.*

Proof. Suppose we have $o_{j\mu k} \rightarrow_i o_{j'\nu\ell}$ for $j \neq j'$ on level L_i. The conformity assumption for conflict relations implies $(o_{j\mu}, o_{j'\nu}) \in CON_{i+1}$. The incompatibility of the corresponding locks and the 2PL-strategy to keep the first of these locks until $o_{j\mu}$ has committed implies $o_{j\mu} <_{i+1} o_{j'\nu}$.

Taken together, we obtain $o_{j\mu} \rightarrow_{i+1} o_{j'\nu}$ and by induction $T_j <_n T_{j'}$.

If 2PL accepted a non-serializable schedule, we would also have $o_{j'\mu'k'} \rightarrow_{i'} o_{j\nu'\ell'}$ on some level $L_{i'}$. Hence, $T_{j'} <_n T_j$ holds, too, which is impossible for a partial order. □

Example 12 Now consider the schedule in Figure 5. Taking the same locks and incompatibility relations as before, T_2 will not be able to set the update-lock on object A before the commit of T_1, because T_1 holds an incompatible increment-lock on A. This implies that the shown interleaving in Figure 5 is not acceptable by 2PL.

Nevertheless, the shown schedule is serializable, which demonstrates that the converse of Theorem 1 does not hold. □

As a straightforward result we show that strict 2PL leads to recoverable and strict schedules.

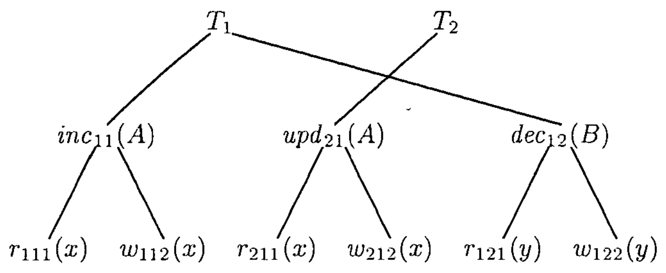


Figure 5: Serializable schedule, not acceptable by 2PL

Proposition 1 *If strict 2PL is used on level L_i, then the resulting schedule is recoverable and strict on level L_i.*

Proof. Assume $o_{j\mu k}(x) \twoheadrightarrow_{i-1} o_{j'\nu\ell}(x)$ and $m(o_{j'\nu}) = c$. Then $o_{j'\nu}$ must have acquired a lock on x, which is only possible in the shrinking phase of $o_{j\mu}$. According to the definition of str-2PL this happens after the commit of $o_{j\mu}$. Hence $m(o_{j\mu}) = c$ holds, i.e. the schedule is recoverable on level L_i.

Next assume $o_{j\mu k}(x) \rightarrow_{i-1} o_{j'\nu\ell}(x)$. According to the definition of str-2PL, $o_{j'\nu}$ can only appear in marked schedules with $m(o_{j\mu})$ being defined. Hence the schedule is strict on level L_i. □

4 A Hybrid Concurrency Control Protocol

We now present the FoPL (Forward oriented Concurrency Control with Preordered Locking) protocol, which ensures serializability by exploiting the level-by-level schedules $S_{i,i-1}$. Then we shall discuss its correctness and completeness with respect to serializability and the issues of recoverability and strictness.

4.1 The Basic FoPL Protocol

The basic structure follows the idea of optimistic protocols or hybrid protocols such as ODL [11]. Thus, FoPL consists of three phases: the propagation, validation and commit-phase. In the *propagation-phase* the operations at the various levels L_i are executed. In addition, some kind of control-structure consisting *flaglists* for the objects and *access-lists* for the operations is built up and will be used later to decide, whether on operation commits or aborts.

The task of the *validation-phase* is to perform this decision. The flaglists are used to detect, whether the interleaved execution of the operations has lead to a situation that forces an abort or not. Finally, in the *commit-phase* the commit or abort is executed. We shall see that the commit-case is the easier one: if in-place updates are used, then the only task is to remove flags from flaglists. The abort-case requires additional efforts for rollback. This will be postponed to Section 7 on recovery.

4.1.1 Propagation

In the propagation phase the operations of a schedule are executed according to some order $<$ which extends $<_0$. In practice this order is built dynamically according to the invocation of transactions. In centralized systems $<$ may be assumed to be total, but in distributed systems this is not necessary. In contrast to other optimistic or hybrid protocols changes to the database are made persistent immediately. We shall also discuss what happens, if changes are only stored in private buffers made persistent in the commit phase if at all.

Since an L_i-operation o_μ is implemented by $act(o_\mu)$, we mark the objects in $\mathfrak{D}_{i-1}$ that are accessed by o_μ. If $o_{\mu k} \in act(o_\mu)$ is the operation (op, A), then we use the *flag* (op, μ) on A. We use a *flaglist* ZL_A for each object $A \in \mathfrak{D}_{i-1}$ (and each $i = 1, \dots, n$), which is built dynamically extending $<_{i-1}$.

In addition, we use *access lists* $AS_i^{(\mu)}$ to keep track of the objects accessed by o_μ. In order to see not only the accessed objects but also the way they are accessed, we take $AS_i^{(\mu)} = act_{i-1}(o_\mu)$, i.e. we use the implementing operations.

Example 13 In Figure 1 the flaglists ZL_A and ZL_B on level L_1 are constructed as $ZL_A = inc_1 inc_2$ and $ZL_B = dec_2 dec_1$. □

When appending a flag to a flaglist an exclusive *short-term-lock* on the flaglist is used. This guarantees that the append-operation is atomic. In particular, concurrent access to the same flaglist can be executed without the risk to loose flags. Deadlocks are not possible, because an operation holds only one lock at a time. Flags will be removed again from flaglists during the commit-phase.

In addition, we may assume that setting the flag is executed before the execution of the operation. For L_0-operations it is necessary to keep this short-term-lock until the operation itself is finished, because this guarantees that there is is no undesired interference with other L_0-operations.

4.1.2 Validation

If all operations in $act(o_\mu)$ have been executed, o_μ initiates its validation. For this, FoPL has to test if all flags that stem from $act(o_\mu)$ are still set. As we shall see below in the paragraph on the commit-phase, flags may have been discarded from a flaglist by another operation.

For the flaglists of all objects $A \in \mathfrak{D}_{i-1}$, which were accessed by $act(o_\mu)$ during the propagation phase, exclusive locks will be requested and kept until the end of the commit-phase. To avoid deadlocks the locks are requested in a total order, which justifies the naming of the protocol. It is not necessary to request locks on the L_{i-1}-objects themselves, since only the flaglists are analyzed. In Section 5 we shall discuss an alternative strategy, which dispenses completely with locks.

The involved objects can be recognized from the access list $AS_i^{(\mu)}$. In particular, $AS_i^{(\mu)}$ indicates all the flags that should still be set.

If at least one flag is missing, the operation o_μ must abort. Otherwise, FoPL tests, whether o_μ was *successful*. This is the case, if none of the objects in $\mathfrak{D}_{i-1}$

accessed by o_μ was accessed by some other operation o_ν before. This can be detected from the flaglists.

Definition 12 An L_i-operation o_μ is *blocked* on an object $A \in \mathfrak{D}_{i-1}$ iff there are flags (op_1, ν) and (op_2, μ) in ZL_A with $\nu \neq \mu$ such that (op_1, ν) precedes (op_2, μ) and $((op_1, A), (op_2, A)) \in CON_{i-1}$ holds.

An L_i-operation o_μ is *successful on an object* $A \in \mathfrak{D}_{i-1}$ iff it is not blocked on A. An L_i-operation o_μ is *successful* iff it is successful on all objects accessed by $act(o_\mu)$. □

If the operation o_μ is successful, it can commit, otherwise it must abort. Both *actions* (commit/abort) are accomplished during the commit phase.

4.1.3 Commit

If an L_i-operation o_μ may commit, the flaglists of all objects $A \in \mathfrak{D}_{i-1}$, which were accessed by $act(o_\mu)$ during the propagation phase, have to be updated. For this the locks requested in the validation-phase are kept. Then all flags from $act(o_\mu)$ have to be removed. After removing the flags, the locks will be released thereby terminating the commit-phase.

If an L_i-operation o_μ must abort, all operations in $act(o_\mu)$ must abort. In this case the flags from $o_{\mu k}$ may still be set or not. In the first case, a compensation is executed, if possible. If not, the object updated by $o_{\mu k}$ has to be replaced by its *before image*. Finally, all remaining flags and all dependent flags have to be deleted.

Definition 13 A flag z from o_μ *depends* on another flag z' from o_ν, iff z' precedes z in ZL_A and $(o_\nu, o_\mu) \in CON_i$ holds or z depends on z'' and z'' depends on z' for some flag z''. □

If a compensation operation $o_{\mu k}^{-1}$ is initiated to abort $o_{\mu k}$, it must be applied to the before image of the *first* operation $o_{\nu l}$, whose flag depends on the flag of $o_{\mu k}$. Because all operations which depend on $o_{\mu k}$ have to abort later on, it is also possible to abort those operations before aborting $o_{\mu k}$. Therefore, a rollback recovery can be invoked.

In the second case there is nothing to do, because an earlier abort from another operation has overwritten the update from $o_{\mu k}$ or the operation was already aborted by the rollback-recovery.

4.2 Lazy Aborts: The FoPL$^+$ Protocol

In order to minimize the number of aborts we may employ the alternative to force an operation to wait and to restart after some time period. We call this *lazy abort*. If FoPL is combined with lazy-abort, the resulting protocol is called FoPL$^+$.

Since conflicts on higher levels are assumed to occur not too often, we may hope that the preceding conflicting flag has been deleted in the meantime. Thus, aborts will only occur, if they are really unavoidable.

As a disadvantage note that deadlocks may occur, if the (transitive closure of the) waiting-for-relation contains a cycle, e.g. an operation o_μ waits for o_ν and o_ν waits for o_μ. In this case the easiest solution is to abort both operations, because o_ν has read from o_μ and o_μ has read from o_ν. We shall discuss alternatives in the next section. Thus, phantom deadlocks cannot occur. If a deadlock is detected, it can be resolved by deleting one flag, which is involved in the deadlock. Deadlocks can be detected with known techniques [5, 18].

Note, however, that a waiting operation does not prevent any object from being accessed. Thus, the possibility of deadlocks is less critical compared with lock protocols.

Example 14 First consider the schedule in Figure 1. Then the progress of the flaglists on L_0-objects x, y and L_1-objects A, B is as follows:

	1	2	3	4	5	6	7	8	9	10	
ZL_A	inc_1	inc_1	$inc_1\ inc_2$	$inc_1\ inc_2$	$inc_1\ inc_2$	$inc_1\ inc_2$	inc_1	inc_1	inc_1		ZL_A
ZL_B					dec_2	dec_2		dec_1	dec_1		ZL_B
ZL_x	r_{11}	$r_{11}\ w_{11}$	r_{21}	$r_{21}\ w_{21}$							ZL_x
ZL_y					r_{22}	$r_{22}\ w_{22}$		r_{12}	$r_{12}\ w_{12}$		ZL_y

Here we assume that the commit of $inc_{11}(A)$ occurs between columns 2 and 3, the commit of $inc_{21}(A)$ between columns 4 and 5, the commit of $dec_{22}(B)$ and T_2 occur between columns 6 and 7, and finally, the commits of $dec_{12}(B)$ and T_1 occur between columns 9 and 10. Thus, the schedule will be accepted. □

Example 15 Consider the schedule in Figure 5, which was not acceptable for 2PL. Looking only at flaglists on L_1-objects we obtain (with FoPL$^+$):

	1	2	3	4
ZL_A	inc_1	$inc_1\ upd_2$	$inc_1\ upd_2$	upd_2
ZL_B			dec_1	

with T_1 committing between 3 and 4, T_2 committing after 4 and T_2 waiting from the beginning of 3 to the end of 4.

Thus, FoPL$^+$ will accept this schedule, but FoPL would abort T_2, since ZL_A cannot be permuted. □

4.3 Private Buffers

As an alternative to immediate in-place updates we may think of using private buffers as in other optimistic or hybrid protocols [11, 12]. Changes to the database objects are only stored in these private buffers during the propagation phase and made persistent in the commit phase if at all.

Since higher-level operations are by assumption implemented by lower-level operations, there is only a need for such buffers on the level L_0. Consequently, we may only expect changes to the protocol on that level.

The crucial point is now that results of operations affecting the database are not visible to other L_0-operations, as long as the corresponding L_1-operation has not finished its commit phase. Hence, after a successful commit of an L_1-operation, all its flags and all dependent flags on level L_0 have to be deleted. Furthermore, since the actual changes to the object are only performed at commit-time, it is insufficient to lock only flaglists. The referred objects must be locked, too. On the other side, the deletion of dependent flags is no longer necessary in the abort case.

Example 16 Consider the development in Example 14, but now assume assume that $inc_{11}(A)$ validates and commits after $r_{211}(x)$ has been performed. In this case the flag r_{21} in ZL_x will be removed causing the later abort of $inc_{21}(A)$. This is correct, since otherwise a wrong value might be used by $inc_{21}(A)$ ("dirty read"), and the update by $inc_{11}(A)$ will get lost ("lost update"). □

Whether it is advantageous to apply FoPL with in-place updates or private buffers on level L_0 depends on the probability of conflicts occurring on level L_0. Note that it is even possible to mix both strategies, i.e. to let some transactions – or L_1-operations – use private L_0-buffers, whereas others use in-place updates. As a rule of thumb, if it can be expected that L_1-operations will commit, then choose in-place updates, because this will trigger less rollbacks.

4.4 Correctness and Completeness

Let us now investigate the correctness and completeness of the FoPL protocol with respect to serializability. In order to distinguish between the basic FoPL protocol and the optimization through lazy aborts we use FoPL$^+$ to indicate the enhanced protocol.

Theorem 2 *Every n-level-schedule accepted by the FoPL protocol is n-level-serializable.*

Proof. If a transaction T_j commits, this also applies to all operations o_μ defining it – at different levels. This is only possible, if all these operations are successful on all objects. These implies that $o_\nu \not\to_i o_\mu$ holds for all other operations o_ν issued by different transactions, and the schedule is conflict-equivalent to a serial schedule with first transaction T_j.

Proceeding inductively and exploiting the fact that flags from submitted transactions will be removed, we obtain an equivalent serial schedule. □

Obviously, since commit for FoPL$^+$ works in the same way as for FoPL, the correctness result carries over to the optimized version with lazy abort.

Corollary 1 *Every n-level-schedule accepted by the FoPL$^+$ protocol is n-level-serializable.* □

In addition to this correctness result, we may also obtain a completeness result for FoPL$^+$, i.e. if we adopt the alternative waiting strategy discussed above. The central argument of the proof states that a deadlock in the waiting graph may only occur iff $o_\mu \dashrightarrow_i o_\nu$ and $o_\nu \dashrightarrow_i o_\mu$ both hold. But this means that the schedule is not serializable.

Lemma 3 *A deadlock in FoPL$^+$ occurs iff the corresponding partial schedule is not serializable.*

Proof. Suppose we have a deadlock between L_i-operations indicated by flaglists $ZL_A = o_\mu p_\nu$ and $ZL_B = q_\nu s_\mu$. For the corresponding L_{i-1}-operations we obtain $o_{\mu k}(A) \dashrightarrow_{i-1} p_{\nu\ell}(A)$ and $q_{\nu m}(B) \dashrightarrow_{i-1} s_{\mu n}(B)$ for suitable indices k, ℓ, m, n. This states that the level-by-level schedule $S_{i,i-1}$ is not serializable.

Conversely, assume a non-serializable level-by-level schedule $S_{i,i-1}$, i.e. $o_{\mu k}(A) \dashrightarrow_{i-1} p_{\nu\ell}(A)$ and $q_{\nu m}(B) \dashrightarrow_{i-1} s_{\mu n}(B)$ holds for L_{i-1}-objects A, B, L_i-operations o_μ, o_ν and $o_{\mu k}(A), s_{\mu n}(B) \in act(o_\mu)$, $p_{\nu\ell}(A), q_{\nu m}(B) \in act(o_\nu)$. This implies the flaglist to contain $ZL_A = o_\mu p_\nu$ and $ZL_B = q_\nu s_\mu$. Since no permutation is possible in ZL_A nor ZL_B, o_μ waits for o_ν and vice versa. Hence, there is a deadlock. □

From this lemma and the preceding remarks the claimed completeness result follows immediately.

Theorem 3 *Every n-level-serializable schedule will be accepted by the FoPL$^+$ protocol.* □

4.5 Recoverability and Strictness

Finally, we investigate recoverability and strictness.

Proposition 2 *If FoPL (or FoPL$^+$) is used on level L_i, then the resulting schedule is recoverable.*

Proof. Assume $o_{j\mu k}(x) \twoheadrightarrow_{i-1} o_{j'\nu\ell}$ and $m(o_{j'\nu}) = c$. The first assumption implies that ZL_x contains $p_{j\mu}q_{j'\nu}$ with the corresponding L_{i-1}-operators p, q. The second assumption implies that $q_{j'\nu}$ could be removed from ZL_x. According to the definition of FoPL this is only possible, if $p_{j\mu}$ was removed earlier from ZL_x, i.e. $m(o_{j\mu})$ is defined.

If we had $m(o_{j\mu}) = a$, then the removal of $p_{j\mu}$ would have triggered the removal of the dependent flag $q_{j'\nu}$ which contradicts the fact that $o_{j'\nu}$ committed. □

However, in contrast to strict 2PL, FoPL (and FoPL$^+$) cannot guarantee strictness, not even cascade-freeness as can be seen from the next example. This is reflected

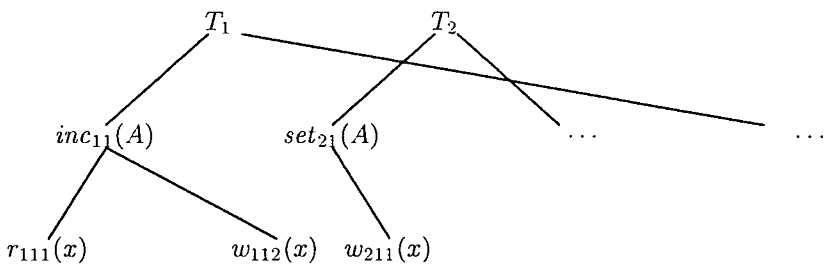


Figure 6: Partial FoPL schedule with cascade

in the protocol by the removal of dependent flags. Non-cascade-freeness is usual with optimistic or hybrid protocols. It is the price to be paid for the increase in transaction throughput resulting from the visibility of operation results before the final commit of a transaction.

Example 17 Consider the schedule sketched in Figure 6. Omitting the dotted parts we obtain a partial schedule with $inc_{11}(A) \twoheadrightarrow_1 set_{21}(A)$, but without $m(T_1)$ being defined. Hence, the schedule is not cascade-free on the top level L_2. □

5 Optimization of the Basic FoPL Protocol

We shall now discuss various optimizations of the basic FoPL protocol or the FoPL$^+$ protocol with lazy aborts. First we ask, whether the exclusive locks in the validation and commit phase are really needed. This will lead to the debatable noPL-strategy.

Next we shall handle rollbacks. The first optimization concerns the ability to detect necessary aborts before entering the validation phase. The second optimization discusses the use of partial rollbacks.

Finally, we consider the absorption of operations. If the effect of an operation does not depend on the execution of a preceding operation, this enables some rollbacks to be dispensed with or the enforcement of validation success.

5.1 Optimistic Locking

In principle, since validating operations only read flaglists, it is not necessary to lock these lists during validation. Furthermore, any other active operation may only add new flags at the end of the lists. Such new flags do not influence the validation result and consequently do not require locks either.

As a further optimization related to optimistic locking [21] it is not even necessary to keep the used exclusive locks during the whole commit-phase, but to release them immediately after changing the flaglist, since all other changes to the objects in question have been detected in the validation phase to commute. Thus, it is only necessary to guarantee the atomicity of the changes to the flaglists via short-term-locks. We shall talk of the noPL-strategy (**no** **p**reordered **l**ocking).

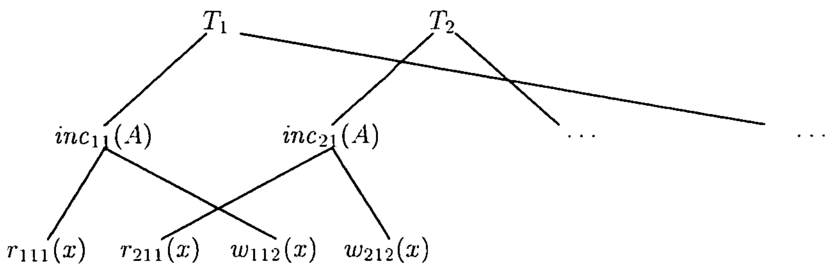


Figure 7: Deadlock in a FoPL$^+$ schedule

However, in the case of an abort such an early release of locks may lead to the removal of flags from operations that are uncritical otherwise. For example, it might not be possible to execute the operation at all due to the locked flaglist. In this case the noPL-strategy may lead to unnecessary aborts. The same applies for the commit, if private buffers are used.

On the other hand, with such an early release of locks we risk the removal of flags from operations that are uncritical otherwise, which may lead to unnecessary aborts.

Example 18 Consider the development in Example 14. Since all operations will commit, there will be no change, if we adopt the noPL-strategy.

However, things change, if we decide to abort $inc_{11}(A)$ and this decision is taken before the execution of $r_{211}(x)$. With the noPL-strategy flaglists are not locked, so it would be possible to execute $r_{211}(x)$ before changing the flaglist ZL_x. Then the flag r_{21} in ZL_x would be removed causing the later abort of $inc_{21}(A)$. Thus, with noPL we risk the unnecessary abort of T_2.

Similarly, consider the schedule in Example 15. With the noPL-strategy the commit of $inc_{11}(A)$ does not lead to a problem, but in the case of an abort the changes to the unlocked flaglist ZL_x may occur after $r_{211}(x)$. Then r_{21} will be deleted from ZL_x, which causes $upd_{21}(A)$ and T_2 to abort. □

It depends on the probability of concurrent access to the same object, whether the noPL-strategy is advisable or not.

5.2 Early and Partial Rollback

In the basic FoPL protocol the necessity to abort an operation and to trigger a rollback will be detected in the validation phase, if a corresponding flag is missing. As an alternative it is possible to inform an operation immediately, when one of its flags will be removed. This strategy of *early rollbacks* will probably prevent further operation from being executed, if we already know about their later abort.

It depends on the duration of operations, whether the communication overhead caused by early rollbacks is small compared with the time waste for operations to be aborted later. In general, early rollback may be advantageous on higher levels.

No matter, whether early rollbacks are applied or not, it is not necessary to rollback operations completely. Since only dependent flags are deleted, it is sufficient to do a partial rollback to the earliest time point, where none of these flags were set.

Partial rollbacks are also useful for removing deadlocks as seen in the next example.

Example 19 Consider the partial schedule in Figure 7, which leads to a deadlock with the flaglist $ZL_x = r_{11}r_{21}w_{12}w_{21}$. It is only necessary to partially rollback until we have $ZL_x = r_{11}$, and then to restart $w_{112}(x)$ again. In this case T_2 has been aborted, but not T_1. □

5.3 Absorption

Consider the case of a conflict $o_{j\mu}(x) \rightarrow_i o_{j'\nu}(x)$, where the second operation does not strongly depend on the first one. According to the definition of $\twoheadrightarrow_i$ this means that the second operation absorbs the first one.

Definition 14 Let T_j, $T_{j'}$ be two n-level-transactions ($j \neq j'$) and $o_{j\mu}(x)$, $o_{j'\nu}(x)$ be two of their L_i-operations. Then $o_{j'\nu}(x)$ *absorbs* $o_{j\mu}(x)$ (notation: $o_{j\mu}(x) \rightsquigarrow_i o_{j'\nu}(x)$) iff $o_{j\mu}(x) \rightarrow_i o_{j'\nu}(x) \wedge o_{j\mu}(x) \not\twoheadrightarrow_i o_{j'\nu}(x)$ holds. □

Absorption $o_{j\mu k}(x) \rightsquigarrow_{i-1} o_{j'\nu\ell}(x)$ allows a brute force strategy to be used when validating $o_{j'\nu}$. We simply remove the flag $p_{j\mu}$ set by $o_{j\mu k}(x)$ in ZL_x, if this makes $o_{j'\nu}$ successful on x. Of course, the deletion of $p_{j\mu}$ will cause $o_{j\mu}$ to be aborted later. Furthermore, we delete all dependent flags which stem from operations that strongly depend on $o_{j\mu k}(x)$.

This strategy immitates a schedule, where $o_{j\mu k}(x)$ was not executed. The strategy will be called *commit enforcement strategy*.

Example 20 As an alternative to the processing in Example 19 we could have used the commit enforcement strategy with $inc_{11}(A)$ which immediately gives $ZL_x = w_{21}$. This will also cause T_2 to abort, but without rolling back $w_{112}(x)$. □

6 Comparison of FoPL$^+$ with Locking

We start with a comparison of FoPL$^+$ with strict two-phase locking (str-2PL). As a probabilistic model for multi-level transactions is still missing, this discussion will necessarily remain preliminary. Nevertheless, we discuss both protocols with respect to implementation costs and transaction throughput.

6.1 Implementation Costs

FoPL$^+$ uses access lists $AS_i^{(\mu)}$ to keep track of the objects accessed by the L_i-level operation o_μ. If str-2PL is used, we must also keep track of the accessed objects to be able to request and release locks. So, with respect to the costs of implementing these access-lists there is no difference between the protocols.

FoPL$^+$ uses flaglists for concurrency-control, and short-term-locks are always necessary when flaglists are accessed. Similarly, str-2PL must support a lock-table to keep track of the locks. This could be arranged as a list of locks for each object.

The first task for str-2PL is to check a locklist for conflicts each time a new lock is requested. This can be achieved by linear search. Even, if an L_i-operation o_μ already holds a lock on an L_{i-1}-object x requested by some $o_{\mu k}$, it is in general not possible to avoid conflict checking, when another operation $o_{\mu\ell}$ wants to access the same object x. Let us illustrate this by a simple example.

Example 21 Suppose $o_{j\alpha}$ holds a *fetch*-lock on the L_1-object A due to some operation $fetch_{j\alpha k}(A)$. Then it is possible, that another operation $o_{j'\beta}$ also holds a *fetch*-lock on A due to some operation $fetch_{j'\beta\ell}(A)$. The two *fetch*-locks are compatible to one another.

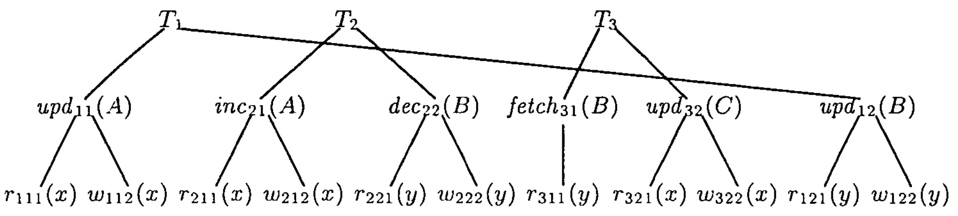
If $o_{j\alpha}$ now requires another lock on A, say an *inc*-lock due to $inc_{j\alpha k'}(A)$, this request must be rejected, as the required lock is incompatible with the *fetch*-lock held by $o_{j'\beta}$. □

On the other hand, FoPL$^+$ does not check anything on appending a flag to a flaglist. The check for conflicts is done in the validation-phase. For each operation $op_{j\alpha k}(x) \in act(o_{j\alpha})$ the first entries in the flaglist preceding $(op, j\alpha)$ have to checked for conflicts. This again leads to linear search.

Thus, for conflict-checking we may state that FoPL$^+$ produces an overhead over str-2PL: flaglists may be longer than locklists and they are accessed more frequently. However, this overhead seems not to be dramatic. In particular, the main parameter to validate this overhead is the number of different operations accessing the same object x within a short period of time. One major assumption for introducing multi-level transactions was that this number is rather small except for level L_0. So the only critical overhead could appear on level L_0, but here we usually have only short read-write sequences.

After commit, str-2PL has to access the lock-table again to release locks. This can be realized by linear searching the locklists associated with the relevant objects. FoPL$^+$ has to delete flags in the case of commit and abort. For abort – and also for L_1-commit, if private buffers are used – dependent flags have to removed either. For this there is no significant difference concerning the implementation costs of str-2PL and FoPL$^+$.

Finally, we must look at the implementation costs for deadlock detection. For this str-2PL has to implement a waiting graph on L_i-operations, which is updated each time a lock-request has been rejected and on commit and abort. The same applies to FoPL$^+$. In particular, the costs for deadlock detection are the same for both protocols. The major difference, however, is that with str-2PL locks are held on objects, whereas with FoPL$^+$ the operations on the waiting graph are independent from the execution of the transactions. This may have an impact on transaction throughput, as we shall discuss next.



T_1
T_2
T_3
$upd_{11}(A)$
$inc_{21}(A)$
$dec_{22}(B)$
$fetch_{31}(B)$
$upd_{32}(C)$
$upd_{12}(B)$
$r_{111}(x)$ $w_{112}(x)$ $r_{211}(x)$ $w_{212}(x)$ $r_{221}(y)$ $w_{222}(y)$ $r_{311}(y)$ $r_{321}(x)$ $w_{322}(x)$ $r_{121}(y)$ $w_{122}(y)$

Figure 8: Non-serializable multi-level schedule with FoPL$^+$-overhead

As a first conclusion we may already state that there is no significant increase in implementation costs, when FoPL$^+$ is used instead of str-2PL. In any case, an increase in transaction throughput would justify the implementation overhead.

6.2 Transaction Troughput

As to transaction throughput the discussion will be based on three examples. The first one is Example 15, where we could show that FoPL$^+$ accepts the serializable schedule from Figure 5, whereas str-2PL would not. According to our completeness result (Theorem 3) for FoPL$^+$ this is no longer astonishing. With str-2PL the operation $upd_{21}(A)$ could first be started after the commit of T_1. This causes an overhead of one L_1-operation and two L_0-operation. In this case the advantages of FoPL$^+$ are evident.

Next we consider two other examples shown in Figures 8 and 9. The first of these examples shows an non-serializable schedule with an overhead for FoPL$^+$. The last example demonstrates the power of FoPL$^+$, when the optimizations with absorption on level L_0 and early rollbacks are employed.

Example 22 Consider the non-serializable schedule from Figure 8. In this case FoPL$^+$ would produce the following flaglists on L_1-objects:

ZL_A	upd_1 inc_2
ZL_B	dec_2 $fetch_3$ upd_1
ZL_C	upd_3

Then FoPL$^+$ would abort o_{22}, because it removes the shortest cycle in the waiting graph, and consequently also o_{31}, o_{32} and o_{12}. A restart – o_{22} must be restarted later than o_{31} and o_{12} may lead to the flaglists

ZL_A	upd_1 inc_2
ZL_B	$fetch_3$ upd_1 dec_2
ZL_C	upd_3

and all transactions would commit now. In this case four L_1-operations composed from seven L_0-operations must be repeated.

For str-2PL the schedule could not occur. Howeverr, due to the *upd*-lock on A held by T_1 the operation o_{21} can only start after T_1 has committed. Thus, the overhead of FoPL$^+$ consists only of two L_1-operations composed from three L_0-operations. □

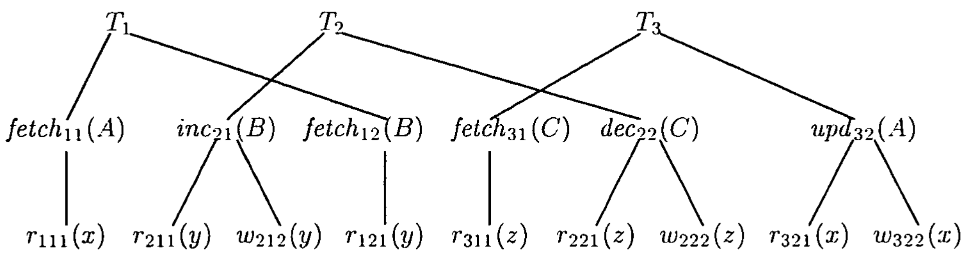


Figure 9: Non-serializable multi-level schedule with str-2PL-overhead

Note that the overhead for FoPL$^+$ occurring in Example 22 is only possible, if we have concurrent access to the same object. On the other hand, the overhead is not as large as expected. Again, the decisive parameter is the number of different operations accessing the same object x within a short period of time.

Example 23 Now consider the schedule in Figure 9. Here str-2PL would request the following locks on L_1-objects:

A	$fetch_1$ upd_3
B	inc_2 $fetch_1$
C	$fetch_3$ dec_2

Since all these pairs of locks are incompatible, the request for the second lock would be rejected. This leads to a deadlock, which can be resolved by aborting and restarting T_3. In addition, $fetch_{12}(B)$ and $dec_{22}(C)$ could first be started after this abort. This means that four L_1-operations had to be repeated. These were composed from six L_0-operations.

If FoPL$^+$ were taken instead, this would result in the following flaglists:

ZL_A	$fetch_1$ upd_3
ZL_B	inc_2 $fetch_1$
ZL_C	$fetch_3$ dec_2

As in the previous example we have to abort and redo $fetch_{31}(A)$, $upd_{32}(A)$ and $dec_{22}(A)$, i.e. three L_1-operations composed from five L_0-operations.

However, with could apply the absorption optimization to commit $dec_{22}(A)$ and hence T_2 immediately. In addition, T_1 would also commit. Since the flag $fetch_3$ would be removed, $fetch_{31}(A)$, and $upd_{32}(A)$ still must be aborted and redone, but this causes only two L_1-operations composed from three L_0-operations.

Finally, since T_2 validates before $upd_{32}(A)$ started, we could even apply early rollback. This means that only $fetch_{31}(A)$ would be repeated, i.e. one L_1-operation composed from one L_0-operation. With these optimizations the overhead caused by str-2PL occurs to be even worse. □

It is not yet possible to draw a general conclusion from these three examples in the sense that FoPL$^+$ is preferable. We could only see, that FoPL$^+$ had advantages, if no abort occurs or an abort occurs for both FoPL$^+$ and str-2PL. Only the situation, where FoPL$^+$ acted "too optimistically" lead to slight advantages for str-2PL. In order to base such investigations on solid theoretical grounds, a probabilistic model for multi-level transactions must be used.

7 Recovery

In our discussion of concurrency control protocols in the preceding three sections we always provided the necessicity of aborting operations or transactions. This means that we have to undo all the effects issued by such operations, which is a significant part of the recovery component. We usually talk of the *rollback* of an operation.

One possible solution to this problem is to employ the principle of write-ahead-logging (WAL), i.e. before updating the database rollback data are stored at some safe place, which is usually a log-file. A accepted good solution based on WAL is ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) [16]. and we shall adopt ARIES to our purposes here. We start giving a short list of the fundamental features of ARIES:

- Recording is not restricted to normal transaction processing, but also happens during rollback through so-called *compensation log records* (CLRs), which prevent UNDO-operations to be executed more than once.
- The storage overhead – besides the logging data – is kept small. On each page only the number of the log record which marks the last change to that page has to be stored.
- ARIES supports partial rollbacks through *savepoints* and fast crash recovery through *checkpoints*, at which information about buffered pages are stored.
- ARIES uses only short-term-locks – so-called *latches* – to access pages, whereas long-term-locks as required by locking protocols are reserved for records.

In [22] an extension ARIES/NT of ARIES to nested transactions has been presented. This extension is tighly coupled with locking protocols and does not employ inverse operations, which are possible in multi-level transactions. In particular, locks are not released after finishing operations that are not transactions. The alternative MLR discussed in [15] exploits inverse operations, but unfortunately assumes them to exist in any case. If they do not exist, the restrictions of ARIES/NT are kept.

In the following we present the extension ARIES/ML for multi-level transactions [6]. ARIES/ML is rather close to MLR, but is not necessarily coupled with a locking protocol. Furthermore, we explicitly differentiate between operations for which there exists an inverse and those for which there exists none.

The major features of ARIES will be preserved. We describe necessary extensions to the data structures and their usage during normal processing and rollback.

The extension allows a coupling with a locking protocol and FoPL and provides the necessary extensions to FoPL with respect to operation aborts. In this way we are also able to support crash recovery.

The data structures used in ARIES/ML comprise various types of *log records* stored in the log-file, an *operation table* and a *dirty pages table.* Each log record has a *log serial number* (LSN) and a field indicating its type, which is ULR, CLR, CCR, RCR, CR, SP or CP. Concretely, we distinguish *update log records* (ULRs), *compensation log records* (CLRs), *committed child records* (CCRs), *reactivate child records* (RCRs), *commit records* (CRs), *savepoints* (SPs) and *checkpoints* (CPs).

Update log records are created during normal transaction processing. Compensation log records record UNDO-operations corresponding to some operation. Committed child records are created, when an operation on a level L_i $(i \neq n)$ has finished. Reactivate child records are created during rollback; they correspond to CCRs. Commit records are created, when a transaction commits.

Savepoints are only used to support partial rollback. Thus, it is sufficient to provide their LSN and their type. Checkpoints are used to fasten crash recovery. They are created regularly. Besides LSN and type they contain the dirty pages table, the operation table and some additional data about the database files. The actual storage of buffered pages is left to the buffer manager. We dispense with an intensive discussion of savepoints and checkpoints.

7.1 Log Records for Normal Processing

In order to define the structure of these records for an L_i-operation o we assume a total order $\sqsubseteq_i^{(j)}$ on $\mathfrak{O}_i^{(j)}$ that includes $<_i^{(j)}$. For simplicity assume that the indices are chosen in such a way that $o_{j\beta k} \sqsubseteq_i^{(j)} o_{j\beta\ell} \Rightarrow k \leq \ell$ holds.

Definition 15 Let $o = op_{j\alpha k}(x)$ be an L_i-operation $(i \neq n)$ of the n-level-transaction T_j. The *update log record* $ulr_{j\alpha k}$ corresponding to o has the form

$$ulr_{j\alpha k} \quad = \quad (lsn_{j\alpha k},\ \text{ULR},\ j\alpha,\ lsn_{j\alpha k-1},\ p,\ \mathit{eff}_{j\alpha k})$$

with the log serial number $lsn_{j\alpha k}$, the type ULR, the identifier $j\alpha$ of the parent operation $trans(o)$, the log serial number $lsn_{j\alpha k-1}$ of the previous operation in $act(o_{j\alpha})$, a pointer p to the page containing the object x affected by o and the effect of o according to Definition 6. □

In general, to refer to the components of a ULR, we write (LSN, type, OpId, PrevLSN, PageId, data). If o is the first operation in $act(o')$, then PrevLSN is undefined, indicated by the null value $\perp$. PageId may also be left undefined, if the object is only virtual, i.e. realized by a set of other objects. Note that ULRs were already present in the basic ARIES algorithms.

CCRs are created, when an L_i-operation o $(0 < i < n)$ has finished. Same as ULRs they contain LSN, type, OpId and PrevLSN. Furthermore, they have a field LastLSN containing a pointer to the last log record created by some operation in $act(o)$, a field ChildId containing the identifier of o itself and a field Op containing the operator of o to indicate, whether a compensation will be possible or not. Thus, we may write (LSN, type, OpId, LastLSN, ChildId, LastLSN, Op).

Definition 16 Let $o = op_{j\alpha k}(x)$ be an L_i-operation $(0 < i < n)$ of the n-level-transaction T_j. The *committed child record* $ccr_{j\alpha k}$ corresponding to o has the form

$$ccr_{j\alpha k} \quad = \quad (lsn_{j\alpha},\ \text{CCR},\ j\alpha,\ lsn_{j\alpha k\ell},\ j\alpha k,\ lsn_{j\alpha k-1},\ op)$$

with the log serial number $lsn_{j\alpha}$, the type CCR, the identifier $j\alpha$ of the parent operation $trans(o)$, the log serial number $lsn_{j\alpha k\ell}$ corresponding to the last operation in $act(o_{j\alpha k})$, the identifier $j\alpha k$ of the operation itself, the log serial number $lsn_{j\alpha k-1}$ of the previous operation in $act(o_{j\alpha})$ and the operator op. □

Commit records are created, when a transaction T_j commits. They are described by LSN, type and OpId. Formally, a *commit record* for an n-level transaction T_j has the form $cr_j = (lsn_j,\ \text{CR},\ j,\ lsn_{jk})$ with the meaning of these components as in Definition 15 before.

7.2 Log Records for UNDO

Since CLRs record UNDO-operations, they also contain LSN, type, OpId, PrevLSN, PageId and a field containing the data which is necessary for REDO. This can be either a before image expressed by the effect as in ULRs or a compensation operation. In addition, CLRs have a field UNDOnextLSN containing the LSN of the log record for the next operation to be undone. Thus, we have the form (LSN, type, OpId, PrevLSN, UNDOnextLSN, PageId, data)

Definition 17 Let $o = o_{j\alpha k}(x)$ be an L_i-operation $(i \neq n)$ of the n-level-transaction T_j. A *compensation log record* $clr_{j\alpha k}$ corresponding to o has the form

$$clr_{j\alpha k} \quad = \quad (lsn^{clr}_{j\alpha k},\ \text{CLR},\ j\alpha k,\ lsn_{j\alpha k},\ lsn^{clr}_{j\alpha k-1},\ p,\ d)$$

with the log serial number $lsn^{clr}_{j\alpha k}$, the type CLR, the identifier $j\alpha k$ of the rolled back operation, the log serial number $lsn_{j\alpha k-1}$ of the ULR for the previous operation in $act(o_{j\alpha})$ the log serial number $lsn^{clr}_{j\alpha k-1}$ of the log record for the next operation to be undone and a pointer p to the page containing the object x affected by o. The last field d is either the effect $\mathit{eff}_{j\alpha k}$ of o according to Definition 6 or a compensation operation o^{-1}. □

CLRs existed already in ARIES. The only difference here is that the data part of a CLR may now contain a compensation operation, unless o resides on level L_0.

Reactivate child records are also created during rollback, when a finished L_i-operation has to be reinstalled in the operation table. Besides LSN and type a RCR has fields OpId, PrevLSN, ChildId, LastLSN and UNDOnextLSN with the same meaning as for the other kinds of log records.

7.3 Normal Transaction Processing

During normal transaction processing the corresponding ULRs, CCRs, CRs, SPs and CPs are written into the log-file. In addition, each page will contain a field PageLSN, in which the LSN of the last entry writing to that page is recorded. For page access, latches are used also by ARIES/ML.

Finally, ARIES/ML manages an operation table and a dirty pages table. The operation table contains information about active operations. Each record in this table contains

- an operation identifier OpId,
- the status of that operation, which may be 'propagate' (p), 'validate' (v) – not used with locking protocols – 'commit' (c) or 'abort' (a),
- LastLSN and UNDOnextLSN.

Whenever a CCR is created the corresponding operation does not need to be kept in the operation table. The same applies to CRs for top-level operations, i.e. transactions.

The dirty pages table contains information about buffered pages. Each of its records contains a PageId and a recovery LSN (RecLSN), which marks the first entry in the log file from which updates to that page were not yet made persistent.

Example 24 Consider the schedule from Figure 1. Assume that x is stored on page p, y on page q and that p is made persistent by the buffer manager after finishing o_{21}. Then the log records in the following list will be created. The list also indicates the dirty pages table (abbreviated as d.p.t.), the operation table and the pair of PageLSNs for p and q.

log-entry	operation table	d.p.t.	Page LSNs
(1,ULR,11,$\perp$,p, ...)	(1,p,$\perp$,$\perp$) (11,p,1,1)		($\perp$,$\perp$)
(2,ULR,11,1,p, ...)	(1,p,$\perp$,$\perp$) (11,p,2,2)	(p,2)	(2,$\perp$)
(3,CCR,1,$\perp$,11,2,*inc*)	(1,p,3,3)	(p,2)	(2,$\perp$)
(4,ULR,21,$\perp$,p, ...)	(1,p,3,3) (2,p,$\perp$,$\perp$) (21,p,4,4)	(p,2)	(2,$\perp$)
(5,ULR,21,4,p, ...)	(1,p,3,3) (2,p,$\perp$,$\perp$) (21,p,5,5)	(p,2)	(2,$\perp$)
(6,CCR,2,$\perp$,21,5,*inc*)	(1,p,3,3) (2,p,6,6)		($\perp$,$\perp$)
(7,ULR,22,$\perp$,q, ...)	(1,p,3,3) (2,p,6,6) (22,p,7,7)	(q,7)	($\perp$,7)
...	...	...	...

Dots indicate some data which are left unspecified. □

7.4 Rollback

Rollback may be started at any time and can be executed until a specified savepoint is reached. Thus, to start a rollback we need a set OpIdSet of operation identifiers and a SaveLSN with SaveLSN = 0 corresponding to a complete rollback.

The first activity is to create a *rollback list* containing the LastLSN from all active operations with a parent in OpIdSet. For this the operation table has to be accessed. Then UNDO-operations will be processed by decreasing LSN following the PrevLSN-entries in log records. Only LSNs that are larger than the given SaveLSN will be considered. Rollback stops, when the rollback list becomes empty.

Depending on the type and the content of the log record r with LSN in the rollback list different actions will be triggered:

- In the case $r.type$ = ULR an UNDO-operation will be performed and the PageLSN of the page affected by the operation underlying r will be reset. The necessary data are kept in the ULR. Furthermore, r.PrevLSN will be added to the rollback list and a CLR r' with r'.UNDOnextLSN = r.PrevLSN will be created. Finally, the fields LastLSN and UNDOnextLSN in the corresponding operation table record will be updated. In this case there is no difference to ARIES.
- In the case $r.type$ = CCR we have to distinguish two different subcases.

  If there exists a compensation operation, it will be executed. If we assume a locking protocol for concurrency control, there is a risk for deadlocks now. ARIES/ML circumvents this problem by allowing only one compensation operation to be active. If it is involved in a deadlock, one of the other operations will be chosen for abort. Thus, in this subcase there is not a big difference to the ULR-case before. In particular, a single CLR will be created.

  Now assume that there is no compensation operation. In this subcase the child operation has to be reactivated and an RCR will be created. Both LastLSN and PrevLSN give rise to new entries in the rollback list.
- The cases $r.type$ = CLR and $r.type$ = RCR can only occur, if a partial rollback has already been performed. In both cases there is nothing to do; just add PrevLSN to the rollback list.

Example 25 Consider the following sequence of log records:

(1,ULR,111,⊥, ...) (2,$\underline{\text{CCR}}$,11,⊥,111,1, ...) (3,ULR,112,⊥, ...)
(4,CCR,11,2,112,3, ...) (5,$\underline{\text{CCR}}$,1,⊥,11,4, ...) (6,ULR,121,⊥, ...)
(7,ULR,121,6, ...) (8,CCR,12,⊥,121,7, ...) (9,ULR,122,⊥, ...)
(10,$\underline{\text{CCR}}$,12,8,122,9, ...) (11,ULR,123,⊥, ...) ,

where the underlined type CCR refers to a compensable operation and dots are used to indicate page identifiers and data entries we are not interested in in this example. A complete rollback of o_1 will start with the rollback list (11,10,5) and create the following continuation of the log sequence:

(12,CLR,123,11,⊥, ...) (13,CLR,12,10,8, ...) (14,RCR,12,13,121,7,⊥)
(15,CLR,121,7,6, ...) (16,CLR,121,15,⊥, ...) (17,CLR,1,5,⊥, ...) .

Here the fifth field in CLRs contains the UNDOnextLSN. Fields in RCRs are listed in the order described above. □

7.5 Crash Recovery

Crash recovery in ARIES/ML follows the same ground procedure as ARIES, i.e. we have three consecutive passes for analysis, REDO and UNDO.

The analysis pass is based on log records starting from the last checkpoint. The goal is to discover where to start the REDO-pass and the set of operations to be undone. The last checkpoint allows an initial reconstruction of the operation table and the dirty pages table. Then log records r following the checkpoint entry are read one after the other. Depending on the type and content of r different actions will be triggered:

- If r.OpId exists, then an entry for OpId must be added to the operation table unless a corresponding record exists. In both cases, the LastLSN will be set to r.LSN.
- If $r.type$ = ULR or $r.type$ = CLR, then the dirty pages table may contain a wrong RecLSN entry for the page indicated by r.PageId. If this is the case RecLSN will be set to r.LSN.
- If $r.type$ = CCR, then the entry for r.ChildId will be deleted in the operation table.
- If $r.type$ = RCR, then (r.ChildId, p, r.LastLSN, r.LastLSN) has to be added to the operation table.
- If $r.type$ = CR, then the entry for r.OpId has to be removed from the operation table.

After analysing these log records, the starting LSN for the REDO-pass will be set to the minimum of all RecLSNs in the dirty pages table. The set OpIdSet of operations to be undone contains all operation identifiers from the operation table which do not have the status 'commit'.

For the REDO-pass there are no changes to ARIES, i.e. log records r starting from REDO-LSN as discovered in the analysis pass will be excuted again, if r.PageId occurs in the dirty pages table and RecLSN $\leq r$.LSN $\wedge$ PageLSN $< r$.LSN holds.

In the UNDO-pass ARIES/ML starts a complete rollback with OpIdSet from the analysis pass and SaveLSN = 0.

8 Conclusion

In this paper we investigated concurrency control and recovery for multi-level transactions which occur naturally in distributed databases. The general idea is to exploit application semantics to reduce the number of conflicts.

Two-phase locking (2PL) can be easily generalized to the multi-level case keeping the advantages of locking protocols. All schedules accepted by 2PL will be serializable. Furthermore, strict 2PL leads to schedules that are recoverable and strict on all levels. As with locking for flat transaction systems the major drawback results from the possibility of deadlocks with the well-known time-consuming detection algorithms.

As an alternative we developed the hybrid FoPL protocol (Forward oriented Concurrency Control with Preordered Locking). Same as 2PL, FoPL only accepts serializable schedules. If combined with a waiting strategy for the case of not successful validation (lazy abort), the modified FoPL$^+$ protocol will accept all serializable schedules. Possible deadlocks in the waiting graph are not critical, since objects are not locked. Moreover, the accepted schedules will be recoverable on all levels. In contrast to 2PL the FoPL protocol is deadlock-free. However, as with other optimistic or hybrid protocols strictness nor cascade-freeness cannot be guaranteed. Finally, we were able to discuss several optimizations of the basic FoPL protocol.

Which choice – strict 2PL or FoPL/FoPL$^+$ – is the better one, depends on various factors. The most important one concerns the probability of conflicts. In general, it is assumed – and this is one of the major motivations behind multi-level transactions – that at least on higher levels the conflict rate will tremendously decrease, which is an argument favouring FoPL. We currently start to realize a test bed in order to compare transaction throughput for various multi-level protocols. We plan to extend these examinations also to generalizations of hybrid protocols that employ time-stamps [3, 10].

The basic idea underlying FoPL stems from the ODL (Optimistic Dummy Lock) protocol [11]. Therefore, it is worth to spend a few words on a comparison. Since ODL has been developed for flat transactions, we must base this comparison on this special case. ODL also uses flags – the so-called "dummy locks" – in the propagation phase. When a transaction T_j issues a read-operation on object x, a *flag* F_j is set on the object x. F_j can be deleted by T_j itself during its validation phase or by another transaction T_k, when T_k performs an actual write-operation on x. Validation basically consists in checking, whether flags are still set.

Compared with FoPL (applied to 1-level-transactions) the major differences are that FoPL uses flaglists, whereas ODL uses a single flag, and that ODL employs a backward validation strategy. Thus, for each commit ODL will force all other operations accessing the same object to abort, no matter whether this is necessary or not. Furthermore, as shown in [21] the backward validation strategy makes a generalization of ODL to multi-level transactions nearly impossible.

As to recovery we adapted ARIES [16] to work both with multi-level locking protocols and FoPL. In the former case one crucial point was to avoid deadlocks during rollback. The extension ARIES/ML preserves the advantages of ARIES such as partial rollbacks, different locking granularities, small storage overhead and the avoidance of multiple UNDO.

References

[1] C. Beeri, A. Bernstein, N. Goodman. A Model for Concurrency in Nested

Transactions Systems. *Journal of the ACM*, 36(2) : 230–269, 1989.

[2] B. Bhargava. Performance Evaluation of the Optimistic Approach to Distributed Database Systems and its Comparison to Locking. In *Proc. 3rd Intern. Conference on Distributed Computing Systems*. IEEE 1982.

[3] C. Boksenbaum, M. Cart, J. Ferrie, J.-F. Pons. Concurrent Certifications by Intervals of Timestamps in Distributed Database Systems. *IEEE Transactions on Software Engineering*, 13(4) : 409–419, 1987.

[4] A. Bernstein, N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Transactions on Computer Systems*, 13(2) : 121–157, 1981.

[5] K. Chandry, J. Misra. A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems. In *Proc. ACM Conference on Principles of Distributed Computing* : 157–164, 1982.

[6] S. Drechsler. *Kopplung des ARIES-Recovery-Systems mit hybriden Mehrschichtschedulern*. Master Thesis, Clausthal Technical University, 1998.

[7] A. Elmargamid, Y. Leu. An Optimistic Concurrency Control Algorithm for Heterogenous Distributed Database Systems. *IEEE Transactions on Data and Knowledge Engineering*, 10(6) : 26–32, 1987.

[8] J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishing, 1993.

[9] T. Härder. Observations on Optimistic Concurrency Control Schemes. *Information Systems*, 9(2) : 111–120, 1984.

[10] U. Halici, A. Dogac. Concurrency Control in Distributed Databases Through Time Intervals and Short-Term Locks. *IEEE Transactions on Software Engineering*, 12(8) : 994–1003, 1989.

[11] U. Halici, A. Dogac. An Optimistic Locking Technique for Concurrency Control in Distributed Databases. *IEEE Transactions on Software Engineering*, 17(7) : 712–124, 1991.

[12] H. Kung, J. Robinson. On Optimistic Methods for Concurrency Control. In *Proceedings of the 5th VLDB-Conference*, 1979.

[13] V. Li. Performance Model of Timestamp Ordering Concurrency Control Algorithms in Distributed Databases. *IEEE Transactions on Computing*, 1987.

[14] W.-T. Lin, J. Nolte. Basic Timestamping, Multiple Version Timestamp, and Two-Phase Locking. In *Proccedings of the 9th VLDB-Conference* : 109–119, 1983.

[15] D. B. Lomet. MLR: A Recovery Method for Multi-level Systems. In M. Stonebraker (Ed.). *Proc. SIGMOD 1992*: 185–194, San Diego 1992.

[16] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write Ahead Logging. *ACM Transactions on Database Systems*, 17(1) : 94–162, 1992.

[17] C. Mohan, B. Lindsay, R. Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems*, 11(4), 1986.

[18] R. Obermarck. Distributed Deadlock Detection Algorithm. *ACM Transactions on Database Systems*, 7(2), 1982.

[19] M. Özsu, P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall International, 1994.

[20] C. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[21] T. Ripke. *Verteilte hybride Synchronisationstechniken in Mehrschichten-Transaktionssystemen*. Ph.D. Thesis. Clausthal Technical University, 1998.

[22] K. Rothermel, C. Mohan. ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions. In P. M. G. Apers, G. Wiederhold (Eds.). *Proc. 15th VLDB*: 337–346, Amsterdam 1989.

[23] G. Weikum. *Transaktionsverwaltung in Datenbanksystemen mit Schichtenarchitektur*. Ph.D. Thesis. Darmstadt Technical University, 1986.

[24] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1):132–180, 1991.

Received February, 1999

A Note on Decidability of Reachability for Conditional Petri Nets

Ferucio Laurenţiu ŢIPLEA [*] Cristina BĂDĂRĂU [*]

Abstract

The aim of this note is to prove that the reachability problem for Petri nets controlled by finite automata, in the sense of [5], is decidable.

1 Introduction and preliminaries

In [5] a new restriction on the transition rule of Petri nets has been introduced by associating to each transition t a language L_t from a family $\mathcal{L}$ of languages. Petri nets obtained in this way have been called *$\mathcal{L}$-conditional Petri Nets* (*$\mathcal{L}$-cPN*, for short). In an $\mathcal{L}$-*cPN* γ, a sequence w of transitions is a transition sequence of γ if it is a transition sequence in the classical sense and additionally $w_1 \in L_t$ for any decomposition $w = w_1tw_2$. In other words, the transition t is conditioned by the transition sequence previously applied.

It has been proved in [6] that the reachability problem for $\mathcal{L}$-*cPN* in the case that $\mathcal{L}$ contains the Dyck language and is closed under inverse homomorphisms and letter-disjoint shuffle product, is undecidable. The families of context-free, context-sensitive, recursive, recursively enumerable languages, and all the families of L-type Petri net languages satisfy the conditions above, but this is not the case of the family of regular languages; the reachability problem for $\mathcal{L}_3$-*cPN*, where $\mathcal{L}_3$ is the family of regular languages, remained open. In this paper we give a positive answer to this problem.

The set of non-negative integers is denoted by $\mathbf{N}$. For an alphabet V (that is, a nonempty finite set), V^* denotes the free monoid generated by V under the operation of concatenation and λ denotes the unity of V^*. The elements of V^* are called *words* over V. A *language* over V is any subset of V^*. Given a word $w \in V^*$, $|w|$ denotes the length of w.

A *finite deterministic automaton* is a 5-tuple $A = (Q, V, \delta, q_0, Q_f)$, where Q is the set of *states*, V is the set of *input symbols*, $q_0 \in Q$ is the *initial state*, $Q_f \subseteq Q$ is the set of *final states* and δ is a function from $Q \times V$ into Q. The *language accepted* by A is defined by $L(A) = \{w \in V^* | \delta(q_0, w) \in Q_f\}$ (the extension of δ to

[*]Faculty of Computer Science,"Al. I. Cuza" University of Iaşi, 6600 Iaşi, Romania, e-mail: `fltiplea@infoiasi.ro`

$V^* \times Q$ is defined as usual). The family of languages accepted by finite deterministic automata, called *regular languages*, is denoted by $\mathcal{L}_3$.

A (finite) *Petri net* (with infinite capacities), abbreviated *PN*, is a 4-tuple $\Sigma = (S, T, F, W)$, where S and T are two finite non-empty sets (of *places* and *transitions*, respectively), $S \cap T = \emptyset$, $F \subseteq (S \times T) \cup (T \times S)$ is the *flow relation* and $W : (S \times T) \cup (T \times S) \rightarrow \mathbf{N}$ is the *weight function* of Σ verifying $W(x, y) = 0$ iff $(x, y) \notin F$. A *marking* of a *PN* Σ is a function $M : S \rightarrow \mathbf{N}$. A *marked PN*, abbreviated *mPN*, is a pair $\gamma = (\Sigma, M_0)$, where Σ is a *PN* and M_0, the *initial marking* of γ, is a marking of Σ.

The behaviour of the net γ is given by the so-called *transition rule*, which consists of:

(a) the *enabling rule*: a transition t is *enabled* at a marking M (in γ), abbreviated $M[t\rangle_\gamma$, iff $W(s,t) \leq M(s)$, for any place s;

(b) the *computing rule*: if $M[t\rangle_\gamma$ then t may *occur* yielding a new marking M', abbreviated $M[t\rangle_\gamma M'$, defined by $M'(s) = M(s) - W(s,t) + W(t,s)$, for any place s.

The transition rule is extended usually to sequences of transitions by $M[\lambda\rangle_\gamma M$, and $M[wt\rangle_\gamma M'$ whenever there is a marking M'' such that $M[w\rangle_\gamma M''$ and $M''[t\rangle_\gamma M'$, where M and M' are markings of γ, $w \in T^*$ and $t \in T$.

Let $\gamma = (\Sigma, M_0)$ be a marked Petri net. A word $w \in T^*$ is called a *transition sequence* of γ if there exists a marking M of γ such that $M_0[w\rangle_\gamma M$. Moreover, the marking M is called *reachable* in γ.

Let $\mathcal{L}$ be an arbitrary family of languages. An *$\mathcal{L}$-conditional Petri net*, abbreviated *$\mathcal{L}$-cPN*, is a pair $\gamma = (\Sigma, \varphi)$ where Σ is a *PN* and φ, the *$\mathcal{L}$-conditioning function* of γ, is a function from T into $\mathcal{P}(T^*) \cap \mathcal{L}$. *Marked conditional Petri nets* are defined as marked Petri nets by changing "Σ" into "Σ, φ".

The *c-transition rule* of a conditional net γ consists of:

(c) the *c-enabling rule*: let M be a marking of γ and $u \in T^*$; the transition t is enabled at (M, u) (in γ), abbreviated $(M, u)[t\rangle_{\gamma,c}$, iff $W(s,t) \leq M(s)$ for any place s, and $u \in \varphi(t)$;

(d) the *c-computing rule*: if $(M, u)[t\rangle_{\gamma,c}$, then t may *occur* yielding a pair (M', v), abbreviated $(M, u)[t\rangle_{\gamma,c}(M', v)$, where $M[t\rangle_\Sigma M'$ and $v = ut$.

As for Petri nets, it can be extended to sequences of transitions.

Let $\gamma = (\Sigma, \varphi, M_0)$ be a marked conditional Petri net. A word $w \in T^*$ is called a *transition c-sequence* of γ if there exists a marking M of γ such that $(M_0, \lambda)[w\rangle_{\gamma,c}(M, w)$. Moreover, the marking M is called *c-reachable* in γ.

2 The main result

The *reachability problem* for Petri nets asks whether, given a net γ and a marking M of γ, M is reachable in γ. The *submarking reachability problem* for Petri nets asks whether, given a net γ, a subset S' of places and a marking M of γ, there exists M' reachable in γ such that $M|_{S'} = M'|_{S'}$. It is well-known that these two problems are equivalent [1] ([4]) and decidable ([3]).

The reachability problem for conditional Petri nets can be defined in a similar way: given an $\mathcal{L}$-conditional net γ and a marking M of γ, is M c-reachable in γ? As we have already mentioned in the first section, for $\mathcal{L}$ being the family of context-free languages (context-sensitive, etc.) the reachability problem is undecidable, and the question is whether this problem is decidable for the case $\mathcal{L} = \mathcal{L}_3$. In what follows we shall give a positive answer to this problem by reducing it to the submarking reachability problem for Petri nets.

Let $\gamma = (\Sigma, \varphi, M_0)$ be an $\mathcal{L}_3$-cPN. We may assume, without loss of generality, that at least a transition of γ is c-enabled at M_0 (otherwise, a marking M is c-reachable in γ iff $M = M_0$). Consider $T = \{t_1, \ldots, t_n\}$, $n \geq 1$, and let $A_i = (Q_i, T, \delta_i, q_0^i, Q_f^i)$ be a finite deterministic automaton accepting the regular language $\varphi(t_i)$, for all i, $1 \leq i \leq n$. We may assume that

- $Q_i \cap Q_j = \emptyset$, for all $i \neq j$, and
- $(S \cup T) \cap \bigcup_{i=1}^{n} Q_i = \emptyset$,

and let $S_i = \{s_q | q \in Q_i\}$, for all i.

We transform now the net Σ into a new net Σ' by adding to the set S all the sets S_i and replacing each transition t_i by some "labelled copies" as follows:

- for each sequence of states $q_1, q_1' \in Q_1, \ldots, q_n, q_n' \in Q_n$ such that $q_i \in Q_f^i$ and $\delta_1(q_1, t_i) = q_1', \ldots, \delta_n(q_n, t_i) = q_n'$, consider a transition $t^i_{q_1, q_1', \ldots, q_n, q_n'}$ which will be connected to places as follows:
  - $t^i_{q_1, q_1', \ldots, q_n, q_n'}$ is connected to places in S as t_i is;
  - for any $1 \leq j \leq n$,

$$W(s_{q_j}, t^i_{q_1, q_1', \ldots, q_n, q_n'}) = W(t^i_{q_1, q_1', \ldots, q_n, q_n'}, s_{q_j'}) = 1.$$

Let M_0' be the marking given by

- $M_0'(s) = M_0(s)$, for all $s \in S$;
- $M_0'(s_{q_0^i}) = 1$, for all $1 \leq i \leq n$;
- $M_0'(s_q) = 0$, for all states $q \in \bigcup_{i=1}^{n} S_i - \{q_0^i | 1 \leq i \leq n\}$,

and let $\gamma' = (\Sigma', M_0')$ be the mPN such obtained (we have to remark that the set T' is non-empty because of the hypothesis). Consider next the homomorphism $h : (T')^* \rightarrow T^*$ given by

$$h(t^i_{q_1, q_1', \ldots, q_n, q_n'}) = t_i,$$

[1] A *decision problem* is a function $A : \mathcal{I} \rightarrow \{0, 1\}$, where $\mathcal{I}$ is a countable set whose elements are called *instances* of A. A decision problem A is *reducible* to a decision problem B if any instance i of A can be transformed into an instance j of B such that $A(i) = 1$ iff $B(j) = 1$. The problems A and B are *equivalent* if each of them can be reduce to the other one.

for any transition $t^i_{q_1,q'_1,\ldots,q_n,q'_n}$ defined as above (the net Σ' together with the homomorphism h is pictorially represented in Figure 2.1: the places are represented by circles, transitions by boxes, the flow relation by arcs, and the numbers $W(f)$ will label the arcs f whenever $W(f) > 1$. The values of h are inserted into the boxes representing transitions).

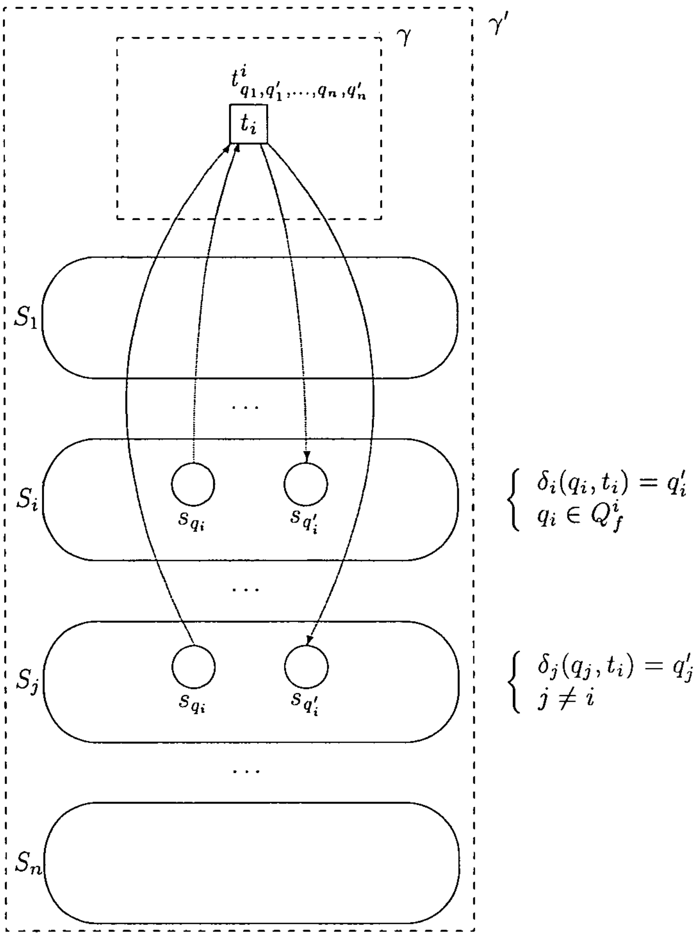

γ'
γ
$t^i_{q_1,q'_1,\ldots,q_n,q'_n}$
t_i
S_1
S_i
S_j
S_n
s_{q_i}
$s_{q'_i}$
s_{q_i}
$s_{q'_i}$
$\delta_i(q_i, t_i) = q'_i$, $q_i \in Q^i_f$
$\delta_j(q_j, t_i) = q'_j$, $j \neq i$

Figure 2.1

It is clear that for any $w \in T^*$ and marking M of γ, $(M_0, \lambda)[w\rangle_\gamma(M, w)$ iff there is $w' \in (T')^*$ and a marking M' of γ' such that $h(w') = w$, $M'_0[w'\rangle_{\gamma'}M'$, and $M = M'|_S$. This shows us that a marking M is reachable in γ iff there is a marking M' reachable in γ' such that $M'|_S = M$. That is, the reachability problem for $\mathcal{L}_3$-*cPN* can be reduced to the submarking reachability problem for Petri nets, and because this problem in decidable for Petri nets we obtain the next result.

Theorem 2.1 *The reachable problem for $\mathcal{L}_3$-cPN is decidable.*

We close this note by the remark that the reachability problem for Petri nets controlled by finite automata, in the sense of Burkhard ([1], [2]), is undecidable. Our approach to control Petri nets by finite automata ([5]) seams to be more adequate because the reachability problem is decidable and, on the other hand, the power of Petri nets is subtle increased (see [6]).

References

[1] H.D. Burkhard. *Ordered Firing in Petri Nets*, Journal of Information Processing and Cybernetics EIK 17, 1981, 71 - 86.

[2] H.D. Burkhard. *What Gives Petri Nets More Computational Power*, Preprint 45, Sektion Mathematik, Humboldt-Universität zu Berlin, 1982.

[3] E.W. Mayr. *An Algorithm for the General Petri Net Reachability Problem*, Proceedings of the 13rd Annual ACM STOC, 1981, 238–246.

[4] J.L. Peterson. *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981.

[5] F.L. Ţiplea, T. Jucan, C. Masalagiu. *Conditional Petri net languages*, Journal of Information Processing and Cybernetics EIK 27, 1991, 55 - 66.

[6] F.L. Ţiplea. *On Conditional Grammars and Conditional Petri Nets*, in Mathematical Aspects of Natural and Formal Languages (Gh. Păun, ed.), World Scientific Series in Computer Science vol. 43, Singapore, 1994, 431–456.

Received May, 1999

Unusual Algorithms for Lexicographical Enumeration*

Pál Dömösi [†]

Abstract

Using well-known results, we consider algorithms for finding minimal words of given length n in regular and context-free languages. We also show algorithms enumerating the words of given length n of regular and context-free languages in lexicographical order.

1 Introduction

E. Mäkinen [8] described algorithms to find the lexicographically minimal words for regular and context-free grammars. Using well-known recent results in [1, 2, 3, 6, 7, 9], we show similar algorithms. E. Mäkinen [8] presents also an algorithm to enumerate the words of a regular language in lexicographical order. We give another algorithm for lexicographical enumeration of regular languages. In addition, using an extension of the well-known CYK-algorithm, we show an algorithm to enumerate the words of length n of a context-free language in lexicographical order. Using the well-known Valiant algorithm, see [11, 5], a little refinement of our solution is attainable.

2 Preliminaries

A *word* (over Σ) is a finite sequence of elements of some finite non-empty set Σ. We call the set Σ an *alphabet,* the elements of Σ *letters.* If u and v are words over an alphabet Σ, then their *catenation* uv is also a word over Σ. Then we also say that u is a *prefix* of uv. In particular, for every word u over Σ, $u\lambda = \lambda u = u$, where λ denotes the *empty word.* Given a word u, we define $u^0 = \lambda$, $u^n = u^{n-1}u$, $n > 0$, $u^* = \{u^n \mid n \geq 0\}$ and $u^+ = u^* \setminus \{\lambda\}$. In addition, we put $\Sigma^n = \{w \in \Sigma \mid |w| = n\}$.

The *length* $|w|$ of a word w is the number of letters in w, where each letter is counted as many times as it occurs. Thus $|\lambda| = 0$. By the *free monoid* Σ^* *generated*

*The research has been supported by the Academy of Finland grant 137358 and the Hungarian National Foundation for Scientific Research grants OTKA-T019392 and OTKA-T030140.

†L. Kossuth University, Institute of Mathematics and Informatics, 4032 Debrecen, Egyetem tér 1, Hungary, domosi@math.klte.hu

by Σ we mean the set of all words (including the *empty word* λ) having catenation as multiplication. We set $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$, where the subsemigroup Σ^+ of Σ^* is said to be *free semigroup generated by* Σ. Subsets of Σ^* are referred to as *languages* over Σ. Given a set Σ, let $card(\Sigma)$ denote the cardinality of Σ. A language $L \subseteq \Sigma^*$ is said to be k*-slender* if $card\{w \in L \mid |w| = n\} \leq k$, for every $n \geq 0$. A language is *slender* if it is k-slender for some positive integer k. A 1-slender language is called *thin* language. A language L is said to be a *union of single loops* (or, in short, USL) if for some positive integer k and words $u_i, v_i, w_i, 1 \leq i \leq k$,

$$(*) \quad L = \bigcup_{i=1}^{k} u_i v_i^* w_i.$$

L is called a *union of paired loops* (or UPL, in short) if for some positive k and words $u_i, v_i, w_i, x_i, y_i, 1 \leq i \leq k$,

$$(**) \quad L = \bigcup_{i=1}^{k} \{u_i v_i^n w_i x_i^n y_i \mid n \geq 0\}.$$

For a USL (or UPL) language L the smallest k such that $(*)$ (or $(**)$) holds is referred to as the USL-index (or UPL-index) of L. A USL language L is said to be a *disjoint union of single loops* (DUSL, in short) if the sets in the union $(*)$ are pairwise disjoint. In this case the smallest k such that $(*)$ holds and the k sets are pairwise disjoint is referred to as the DUSL-index of L. The notions of a *disjoint union of paired loops* (DUPL) and DUPL-index are defined analogously considering $(**)$. We shall use the following well-known results.

Theorem 2.1 *[9] The next conditions, (i)-(iii), are equivalent for a language L.*
(i) L is regular and slender.
(ii) L is USL.
(iii) L is DUSL.

$\square$

Theorem 2.2 *[9] Every UPL language is DUPL, slender, linear and unambiguous.*

$\square$

Theorem 2.3 *[6, 10] Every slender context-free language is UPL.* $\square$

We will use the following extension of Theorem 2.3.

Theorem 2.4 *[7] A given slender context-free language can be effectively written as a disjoint union of (finitely many) paired loops.* $\square$

The next statement is a direct consequence of the constructive proof of Theorem 2.1 in [9].

Theorem 2.5 *A given slender regular language can be effectively written as a disjoint union of (finite many) single loops.* □

3 Finding minimal words of given length and enumeration of regular languages

Given a total order $\prec$ on Σ, a *lexicographical order* on Σ^* is defined as an extension of $\prec$ to Σ^* such that for any pair $u, v \in \Sigma^*$, $u \prec v$ if and only if either $v = uu', u' \in \Sigma^+$ or $u = wxu'$, $v = wyv'$, $x \prec y$ for some $w, u', v' \in \Sigma^*$ and $x, y \in \Sigma$. We will denote by $first(\Sigma)$ the first element of Σ under $\prec$. Moreover, for every $u \in \Sigma^*$ we put $next(u) = \min\{v \mid v \in \Sigma^*, u \prec v\}$. In addition, we put $Pref(L) = \{v \mid \exists u \in L,\ v' \in \Sigma^*, u = vv'\}$. Thus $Pref(L)$ denotes the set of all prefixes of words in L.

Given a language L, the language L_{min} is defined by taking from all words of L of the same length only the first one in lexicographic order. Of course, L_{min} is a thin language. We shall use the following results.

Theorem 3.1 *[1, 4] For every regular language L, the language L_{min} is regular, and a regular grammar for it can be effectively constructed.* □

Theorem 3.2 *[2] For every context-free language L, the language L_{min} is context-free. Moreover, given a context-free grammar generating L, a context-free grammar for L_{min} can be effectively constructed.* □

Using Theorem 3.1 and Theorem 3.2, together with Theorem 2.5 and Theorem 2.4, the following algorithms can be constructed.

Algorithm regmin
Input: A regular grammar $G = (V, \Sigma, P, S)$ and a total order $\prec$ on Σ.
Output: A finite language $L_G = \{u_1, v_1, w_1, \ldots, u_{n(G)}, v_{n(G)}, w_{n(G)}\}$ having

$$L_{min} = \bigcup_{i=1}^{n(G)} \{u_i v_i^n w_i \mid n \geq 0\}.$$

End of algorithm regmin

Algorithm cfmin
Input: A context-free grammar $G = (V, \Sigma, P, S)$ and a total order $\prec$ on Σ.

Output: A finite language $L_G = \{u_1, v_1, w_1, x_1, y_1, \dots, u_{n(G)}, v_{n(G)}, w_{n(G)}, x_{n(G)}, y_{n(G)}\}$ having

$$L_{min} = \bigcup_{i=1}^{n(G)} \{u_i v_i^n w_i x_i^n y_i \mid n \geq 0\}.$$

End of algorithm cfmin

On the basis of the above observations, we now show how to construct the following algorithms.

Algorithm REGMIN

Input: A regular grammar $G = (V, \Sigma, P, S)$, a total order $\prec$ on Σ and a positive integer n.

Output: A finite language $L_G = \{u_1, v_1, w_1, \dots, u_{n(G)}, v_{n(G)}, w_{n(G)}\}$ (having $L_{min} = \bigcup_{i=1}^{n(G)} \{u_i v_i^n w_i \mid n \geq 0\}$) and

- a pair k, ℓ of positive integers such that $1 \leq k \leq n(G)$ if the word of length n of L_{min} exists and it has the form $u_k v_k^\ell w_k$;
- an error message if L_{min} has no word of length n.

```
Method: Apply the algorithm regmin; $k, \ell \leftarrow 0$;
for $i \leftarrow 1 \dots n(G)$ do
  if the equation $|u_i w_i| + |v_i|\alpha = n$ has a positive integer solution for $\alpha$
  then $k \leftarrow i; \ell \leftarrow \alpha$;
od
```

Output:

- k, ℓ, if $k > 0$;
- an error message if $k = 0$;

End of algorithm REGMIN

Algorithm CFMIN

Input: A context-free grammar $G = (V, \Sigma, P, S)$, a total order $\prec$ on Σ and a positive integer n.

Output: A finite language $L_G = \{u_1, v_1, w_1, x_1, y_1, \dots, u_{n(G)}, v_{n(G)}, w_{n(G)}, x_{n(G)}, y_{n(G)}\}$ (having $L_{min} = \bigcup_{i=1}^{n(G)} \{u_i v_i^n w_i x_i^n y_i \mid n \geq 0\}$) and

- a pair k, ℓ of positive integers such that $1 \leq k \leq n(G)$ if the word of length n of L_{min} exists and it has the form $u_k v_k^\ell w_k x_k^\ell y_k$;
- an error message if L_{min} has no word of length n.

```
Method:
Apply the algorithm cfmin; $k, \ell \leftarrow 0$;
for $i \leftarrow 1 \dots n(G)$ do
```

```
  if the equation $|u_i w_i y_i| + |v_i x_i|\alpha = n$ has a positive integer solution for $\alpha$
  then $k \leftarrow i; \ell \leftarrow \alpha$;
od
Output:
```

- k, ℓ, if $k > 0$;
- an error message if $k = 0$;

End of algorithm CFMIN

It is well-known that for every pair of regular grammars G_1, G_2, a regular grammar G having $L(G) = L(G_1) \setminus L(G_2)$ can be effectively constructed. Therefore, by Theorem 3.1 and Theorem 2.5, we can consider the following idea for enumerating the words of length n in $L(G)$ in lexicographical order having a regular grammar G. Assume that, using **REGMIN**, we just get either the word of length n of $(L(G))_{min}$ or an error message that there exists no such a word in $(L(G))_{min}$. Having the error message, we are ready. Otherwise, construct a regular grammar G' with $L(G') = (L(G) \setminus (L(G))_{min}$, consider G' instead of G and use the above procedure again.

In more details, we consider the following algorithm.

Algorithm reg-enumerate
Input: A regular grammar $G = (V, \Sigma, P, S)$, a total order $\prec$ on Σ and a positive integer n.
Output:

- $L_{G_j} = \{u_{j,1}, v_{j,1}, w_{j,1} \ldots, u_{j,n(G_j)}, v_{j,n(G_j)}, w_{j,n(G_j)}\}$, k_j, ℓ_j, $j = 1, \ldots, m$ (having $m = card\{p \in L(G) \mid |p| = n\}$, $L_j = \bigcup_{i=1}^{n(G_j)} u_{j,i} v_{j,i}^* w_{j,i}$, $j = 1, \ldots, m$ with $L_0 = L(G)$, $L_1 = L_{min}$, $L_k = L_{k-2} \setminus L_{k-1}$, $k = 2, \ldots, m$, such that $1 \leq k_j \leq n(G_j)$, $|u_{j,k_j} v_{j,k_j}^{\ell_j} w_{j,k_j}| = n$, $j = 1, \ldots, m$, $u_{1,k_1} v_{1,k_1}^{\ell_1} w_{1,k_1} \prec u_{2,k_2} v_{2,k_2}^{\ell_2} w_{2,k_2} \prec \ldots \prec u_{m,k_m} v_{m,k_m}^{\ell_m} w_{m,k_m}$) if $L(G)$ has a word of length n;
- an error message otherwise.

```
Method:
$P =' no'$;
while REGMIN has no error message do
$P =' yes'$;
Apply the algorithm REGMIN;
  Construct a regular grammar $G'$ having $L(G') = (L(G) \setminus (L(G))_{min}$; $G \leftarrow G'$;
od
  if $P =' no'$ then Output: an error message;
  End of algorithm reg-enumerate
```

4 Enumeration of context-free languages

In [8] it is conjectured that there exists no efficient enumeration algorithm for the lexicographic enumeration of context-free languages. We can provide an algorithm for enumeration of context-free languages, running in polynomial time and space. First we consider the following modified version of CYK algorithm to decide whether a word is a prefix of a word of given length of the language.

Algorithm MCYK
Input: A context-free grammar $G = (V, \Sigma, P, S)$ given in Chomsky normal form, a word $u = b_1 \dots b_m \in \Sigma^+$ $(b_1, \dots, b_m \in \Sigma)$, and a positive integer n.
Output: a variable P having the value

- $P =$*'yes'*, if u is a prefix of an n-length word in $L(G)$;
- $P =$*'no'*, otherwise.

Method:

```
if $m > n$ then $P =$'no' else do
for $i \leftarrow 1 \dots n$ do
  if $i \leq m$
    then $V_{i,1} \leftarrow \{A \mid A \to b_i$ is a production $\}$
    else $V_{i,1} \leftarrow \{A \mid \exists a \in \Sigma$ such that $A \to a$ is a production $\}$
od
for $j \leftarrow 2 \dots n$ do
  for $i \leftarrow 1 \dots n - j + 1$ do
  $V_{i,j} \leftarrow \emptyset$;
    for $k \leftarrow 1 \dots j - 1$ do
    $V_{i,j} \leftarrow V_{i,j} \cup \{A \mid A \to BC$ is a production, $B$ is in $V_{i,k}$ and $C$ is in $V_{i+k,j-k}\}$
    od
  od
   od
if $S \in V_{1,n}$ then $P =$'yes'
else $P =$'no';
   od
Output: $P$;
```

End of algorithm MCYK

Now we construct an algorithm to enumerate the words of length n in context-free languages. We consider the following idea for such an algorithm. Assume we just output $u = a_1 a_2 \cdots a_n$ and are looking for the next word in lexicographical order of length n in $L(G)$. This word, when it exists, has the form

$$v = a_1 a_2 \cdots a_i b_{i+1} b_{i+2} \cdots b_n,$$

for some $0 \leq i \leq n - 1, a_{i+1} \prec b_{i+1}$. Clearly, when v exists, we have

$$i = \max\{j \mid 0 \leq j \leq n - 1, a_1 a_2 \cdots a_j \text{ is the prefix of a word } w \in L(G) \text{ such that } |w| = n \text{ and the (j+1)}st \text{ letter of } w \text{ is bigger than } a_{j+1}\},$$

$$b_{i+1} = \min\{b \in \Sigma \mid a_{i+1} \prec b \text{ and } a_1 a_2 \cdots a_i b \in Pref(L(G) \cap \Sigma^n)\},$$

and, for any $2 \leq j \leq n - i$,

$$b_{i+j} = \min\{b \in \Sigma \mid a_1 a_2 \cdots a_i b_{i+1} b_{i+2} \cdots b_{i+j-1} b \in Pref(L(G) \cap \Sigma^n)\}.$$

Now, the algorithm should be clear; find first i and b_{i+1} and then, in order, b_{i+2}, b_{i+3}, ..., b_n. Notice that v exists iff i exists and, when both do, we look for each b_j knowing that there must be one.

Algorithm cf-enumerate
Input: A context-free grammar $G = (V, \Sigma, P, S)$, a total order $\prec$ on Σ and a positive integer n.
Output:

- The words of length n in $L(G)$ in lexicographical order if $L(G)$ has a word of length n;
- an error message otherwise.

Method:

Determine the minimal word $p_{min(G,n)}$ of length n in $L(G)$, if such a word exists (apply either methods in [8] having $O(n^2)$ time complexity or the algorithm **CFMIN**);

```
if there exists no word of length $n$ in $L(G)$ then $P$ ='no';
Output: an error message;
else do $a_1 \ldots a_n \leftarrow p_{min(G,n)}$; $P$ ='yes' od
while $P$ ='yes' do
    Output: $a_1 \ldots a_n$;
    $P$ ='no'; $m \leftarrow n + 1$;
  while $P$ ='no' and $m > 1$ do
    $m \leftarrow m - 1$; $b \leftarrow a_m$;
     while $P$ ='no' and $next(b) \in \Sigma$ do
      $b \leftarrow next(b)$; $b_m \leftarrow b$;
if $m > 1$ then apply MCYK for the inputs $a_1 \ldots a_{m-1} b_m$ and $n$;
else apply MCYK for the inputs $b_1$ and $n$;
      od
  od
  if $P$ ='yes' then do
if $m > 1$ then $b_1 \ldots b_{m-1} \leftarrow a_1 \ldots a_{m-1}$;
     while $m < n$ do
    $m \leftarrow m + 1$
    $b \leftarrow first(\Sigma)$; $b_m \leftarrow b$;
    Apply MCYK for the inputs $b_1 \ldots b_m$ and $n$;
       while $P$ ='no' and $next(b) \in \Sigma$ do
    $b \leftarrow next(b)$; $b_m \leftarrow b$;
    Apply MCYK for the inputs $b_1 \ldots b_m$ and $n$;
```

```
      od
    od
  $a_1 \dots a_n \leftarrow b_1 \dots b_n$;
  od
od
   End of algorithm cf-enumerate
```

Acknowledgement. The author expresses his gratitude to Professor Arto Salomaa for his kind invitation and hospitality.

The author thanks the anonymous referee for pointing out the errors in a preliminary version which have improved the clarity and the quality of the paper.

References

[1] M. Andraşiu, G. Păun, J. Dassow, A. Salomaa, Language-theoretic problems arising from Richelieu cryptosystems, *Theoret. Comput. Sci.,* **116** (1993) 339-357.

[2] J. Berstel, L. Boasson, The set of minimal words of a context-free language is context-free, *J. Comput. Syst. Sci.,* **55** (1997) 477-488.

[3] J. Dassow, G. Păun, A. Salomaa, On thinness and slenderness of L languages, *Bull.* EATCS **49** (1993), 152-158.

[4] S. Eilenberg, Automata, languages and machines, Vol A, *Academic Press,* New York, 1974.

[5] K. Hermann, G. Walter, A simple proof of Valiant's lemma, *R.A.I.R.O. Inform. Theor. Appl.* **20** (1986) 183-190.

[6] L. Ilie, On a conjecture about slender context-free languages, *Theoret. Comput. Sci.,* **132** (1994) 427-434.

[7] L. Ilie, On lengths of words in context-free languages, *Theoret. Comput. Sci.,* to appear.

[8] E. Mäkinen, On lexicographic enumeration of regular and context-free languages, *Acta Cybernet.,* **13** (1997) 55-61.

[9] G. Păun, A. Salomaa, Thin and slender languages, *Discrete Appl. Math.* **61** (1995) 257-270.

[10] D. Raz, Length considerations in context-free languages, *Theoret. Comput. Sci.* **183** (1997) 21-32.

[11] L. Valiant, General context-free recognition in less than cubic time, *J. Comput. Syst. Sci.* **10** (1975) 308-315.

Received June, 1998

Acta Cybernetica 14 (2000) 469-477.

Inferring pure context-free languages from positive data

Takeshi Koshiba * Erkki Mäkinen † Yuji Takada ‡

Abstract

We study the possibilities to infer pure context-free languages from positive data. We can show that while the whole class of pure context-free languages is not inferable from positive data, it has interesting subclasses which have the desired inference property. We study uniform pure languages, i.e., languages generated by pure grammars obeying restrictions on the length of the right hand sides of their productions, and pure languages generated by deterministic pure grammars.

1 Introduction

In pure grammars, no distinction is made between terminals and nonterminals. It follows that the generative capacity of pure grammars is much weaker than that of corresponding Chomsky type grammars. It is argued [5, 14] that the custom of dividing the alphabet of a grammar originates from the linguistic background of formal language theory and in fact, it would be more natural to study rewriting systems that do not make difference between terminals and nonterminals.

In this paper we study the possibilities to infer pure languages from posivite data. The well-known negative result by Gold [9] says that regular languages cannot be inferred from positive data only. This negative result has initiated a search for language classes having the desirable inference property. The found subclasses include, among others, 1-variable pattern languages [1], paranthesis languages [6], locally testable languages [8], deterministic even linear languages [12], and k-reversible languages [3]. Even more closely related to the present paper is Yokomori's [18] result concerning the inferability of PD0L languages from positive data and especially Tanida and Yokomori's [16] results on the inferability of monogenic pure languages.

*High Performance Computing Research Center, Fujitsu Laboratories Ltd., Present address: Telecommunications Advancement Organization of Japan, 1-1-32 Shin'urashima, Kanagawa-ku, Yokohama 221-0031, Japan, e-mail: koshiba@acm.org

†Department of Computer Science, University of Tampere, P.O. Box 607, FIN-33101 Tampere, Finland, e-mail: em@cs.uta.fi

‡Personal Systems Labs., Fujitsu Laboratories Ltd., 2-2-1 Momochihama, Sawara-ku, Fukuoka 814, Japan, e-mail: yuji@flab.fujitsu.co.jp

We show here that while the whole class of pure context-free languages is not inferable from positive data, it has interesting subclasses which have the desired inference property. The subclasses are defined by restricting the length of the right hand sides in the productions (uniform pure languages) or the number of productions (deterministic pure grammars).

The fact that the whole class of pure context-free languages is not inferable from positive data only is earlier shown by Tanida and Yokomori [16].

2 Preliminaries

We assume a familiarity with the basics of formal language theory and grammatical inference as given e.g. in [11] and [4], respectively. As inference criterion we use "identification in the limit" [9, 4].

If not otherwise stated we follow the notations and definitions of [11]. The length of a string w is denoted by $lg(w)$. A production in a (Chomsky-type) context-free grammar is said to be *terminating* if the right hand side contains no nonterminals. Otherwise, a production is said to be *continuing.*

We now define pure grammars and languages. A *pure context-free grammar* is a system $G = (\Sigma, P, s)$, where Σ is a finite alphabet, P is a finite set of productions of the form $\alpha \rightarrow \beta$, where α is in Σ and β is a word over Σ. For the sake of simplicity we assume that the empty word λ is not allowed as a right hand side of any production. Contrary to most earlier articles on pure grammars (cf. e.g. [7, 14]), we suppose that the *axiom* s is a single word over Σ. Relation $\Rightarrow$ (yields directly) and its reflexive transive closure $\Rightarrow^*$ are defined in Σ^* as usual. The language generated by a system $G = (\Sigma, P, s)$ is defined as

$$L(G) = \{w \mid s \Rightarrow^* w\}.$$

A language is a *pure context-free language* if it can be generated by a pure context-free grammar. The class of pure context-free languages is denoted by $\mathcal{P}$. Note that $\mathcal{P}$ and the class of regular languages are incomparable.

We consider here pure context-free grammars and languages only. We hereafter omit the phrase "context-free", and simply talk about pure grammars and pure languages.

A pure grammar G is *monogenic* if, whenever w is in $L(G)$ and $w \Rightarrow w'$, then there are unique words w_1 and w_2 such that $w = w_1xw_2$, $w' = w_1yw_2$, and $x \rightarrow y$ is a production.

A pure grammar G is *deterministic* if, for each symbol a, there is at most one production with a on the left hand side. A pure language is deterministic if there exists a deterministic pure grammar generating it. We denote the class of deterministic pure languages by $\mathcal{D}$.

A pure grammar G is *reduced* if every symbol appear in some word of $L(G)$. If a reduced pure grammar is monogenic then it is also deterministic. On the other hand, a deterministic pure grammar is not necessarily monogenic [14].

An *indexed family of nonempty recursive languages* is an infinite sequence $L_1, L_2, L_3, \ldots$, where each L_i is a nonempty language with decidable membership problem. The following two well-known results by Angluin [2] are essential for our further discussion.

Theorem 2.1 *([2]) If an indexed family of nonempty recursive languages is inferable from positive data, then there exists, on any input $i, i \geq 1$, a finite set of strings T_i such that*

1. *$T_i \subseteq L_i$, and*
2. *for all $j \geq 1$, if $T_i \subseteq L_j$, then L_j is not a proper subset of L_i.*

Let $\mathcal{L}$ be an indexed family of nonempty recursive languages. We say that $\mathcal{L}$ has *finite thickness*, if for each nonempty finite set $S \subseteq \Sigma^*$, the set $C(S) = \{L \mid S \subseteq L$ and $L = L_i$ for some $i\}$ is of finite cardinality.

Theorem 2.2 *([2]) If an indexed family of nonempty recursive languages has finite thickness, then it is inferable from positive data only.*

Note that thickness is not defined in terms of the number of representations (generating systems), but in terms of the number of languages.

3 A negative result

As the class of languages inferable from positive data only is known to be quite restricted, it is to be expected that $\mathcal{P}$ in not inferable from positive data. To prove this we can follow Yokomori's corresponding proof [18](Thm. 3) for propagating 0L-systems. A different proof is given in [16].

Theorem 3.1 *$\mathcal{P}$ is not inferable from positive data only.*

Proof We derive a contradiction with Theorem 2.1.

Consider the language $L = \{b\} \cup \{a^n \mid n \geq 2\}$. L is in $\mathcal{P}$, since it can be generated from axiom b with productions $b \to aa$ and $a \to aa$.

Let T be any nonempty finite subset of L, and let $T' = T \setminus \{b\}$. Further, let $T' = \{a^{n_1}, \ldots, a^{n_p}\}$.

Consider a pure grammar H with axiom b and with productions

$$\{b \to a^{n_1}, \ldots, b \to a^{n_p}\}.$$

We have $T \subseteq L(H) \subset L$ contradicting Theorem 2.1. □

Remark 3.1 *The proof of Theorem 3.1 shows why we do not allow an arbitrary set of axioms but a single axiom string. If an arbitrary set of strings were possible as an axiom, then Theorem 3.1 would hold also for all reasonable defined subclasses of pure grammars. Namely, we could choose T as the axiom set, and we would not even need any productions to show that the condition of Theorem 2.1 does not hold.*

4 k-uniform pure grammars

We say that a pure grammar $G = (\Sigma, P, s)$ is *k-uniform*, $k > 1$, if each production $\alpha \to \beta$ in P has $lg(\beta) = k$. A pure language L is k-uniform if there exists a k-uniform pure grammar generating L. The class of k-uniform pure languages is denoted by $\mathcal{P}(k)$.

The property of a pure grammar being k-uniform has its implications to the length set of the language generated. (The *length set* of a language L is defined by $LS(L) = \{lg(w) \mid w \in L\}$.) Namely, the length of the axiom and the constant k together uniquely defines the length set.

It also follows directly that $\mathcal{P}(i)$ and $\mathcal{P}(j)$, $i \neq j$, cannot have any infinite language in common. Moreover, the union $\bigcup_{i>1} \mathcal{P}(i)$ of k-uniform pure languages is clearly a proper subset of $\mathcal{P}$. These remarks show that the classes of k-uniform pure languages are quite restricted. On the other hand, each of the classes $\mathcal{P}(i)$, $i \geq 2$, contains non-regular languages. A simple example in the case $k = 3$, is

$$G_1 = (\{a, b, c\}, \{c \to acb\}, abc)$$

with $L(G_1) = \{a^n cb^n \mid n \geq 1\}$.

Hagauer [10] has shown that also $\mathcal{P}(2)$ contains non-regular languages. Namely, he has shown that

$$G_2 = (\{a, b, c\}, \{a \to ab, b \to bc, c \to ca\}, a)$$

produces a non-regular language.

Theorem 4.1 *$\mathcal{P}(k)$, $k \geq 2$, is inferable from positive data only.*

Proof We show that $\mathcal{P}(k)$ has finite thickness, and hence, by Theorem 2.2 is inferable from positive data only.

Given any set S, the length of the shortest word in S gives an upper bound to the length of the axiom. Similarly, the cardinality of Σ (the alphabet considered) gives an upper bound for the number of productions having exactly k symbols in their right hand sides. Thus, $\mathcal{P}(k)$ has finite thickness. □

By letting $\mathcal{Q}(n) = \mathcal{P}(2) \cup \mathcal{P}(3) \cup \ldots \cup \mathcal{P}(n)$, where n is any natural number, we can clearly prove also the following

Theorem 4.2 *$\mathcal{Q}(n)$ is inferable from positive data only.*

We can continue further to this direction, and define a pure grammar G to be *length-bounded* if there exists a natural number k such that the length of any right hand side in G's productions is at most k. A pure language L is length-bounded if there exists a length-bounded pure grammar G such that $L(G) = L$.

Theorem 4.3 *Length-bounded pure languages are inferable from positive data only.*

Proof Analogously to the proof of Theorem 4.1. □

The class $\mathcal{P}(2)$ is somewhat related to the class of uniquely terminating regular languages which is known to be inferable from positive data [13].

A (Chomsky type) regular grammar $G = (V, S, P, S)$ is *uniquely terminating* if the productions in P fulfil the following conditions for each nonterminal A in G:

1. $A \to aB$ and $A \to aC$ imply $B = C$;

2. A has a unique terminating production; i.e. each nonterminal has exactly one terminating production. The terminals appearing in the right hand sides of terminating productions are all different.

A regular language L is uniquely terminating if there exists a uniquely terminating regular grammar generating L. Uniquely terminating languages are inferable from positive data [13].

Each uniquely terminating regular language is a member of $\mathcal{P}(2)$ provided that there are no terminals appearing both in terminating and in continuing productions. Let $G = (V, S, P, S)$ be a uniquely terminating regular grammar. The corresponding 2-uniform pure grammar H can be generated as follows. If $S \to a$ is the unique terminating production for the start symbol S of G, then a is the axiom of H. If $A \to bB$ is a continuing productions in G and the unique terminating productions for A and B are $A \to c$ and $B \to d$. Then H has the production $c \to bd$. Other productions are not needed.

The additional requirement that no terminal can appear in productions of both type characterizes well the difference between Chomsky type grammars and pure grammars. If the requirement does not hold, then the above construction ends up with a pure 2-uniform grammar which may produce words not in the original Chomsky language.

5 Inferring deterministic pure languages

Tanida and Yokomori [16] have shown that monogenic pure languages are inferable from positive data only. Their inference algorithm updates its conjectures in time $O(N^3)$ where N is the total length of the positive samples presented.

We shall now study the inferability of deterministic pure languages. Recall that reduced monogenic pure grammars are always deterministic, but deterministic pure grammars are not necessarily monogenic.

In order to prove that deterministic pure languages are inferable from positive data, we need the concept of finite elasticity from [17, 15].

A class $\mathcal{C}$ of languages has *infinite elasticity* if and only if there is an infinite sequence $w_0, w_1, w_2, \ldots$ of strings and an infinite sequence $L_1, L_2, L_3, \ldots$ of languages from $\mathcal{C}$ such that for all $n \geq 1$, $\{w_0, w_1, \ldots, w_{n-1}\} \subseteq L_n$ but $w_n \notin L_n$. If a class $\mathcal{C}$ does not have infinite elasticity, then it has *finite elasticity*.

Notice that in the above definition both the languages $L_1, L_2, L_3, \ldots$ and the strings $w_0, w_1, w_2, \ldots$ are pairwise disjoint, i.e. each language (resp. string) appears at most once in the sequence $L_1, L_2, L_3, \ldots$ (resp. $w_0, w_1, w_2, \ldots$).

Theorem 5.1 *[17, 15] If a class C of languages has finite elasticity, then C is inferable from positive data only.*

We can now show that $\mathcal{D}$ has finite elasticity, and hence, it is inferable from positive data only.

Theorem 5.2 *$\mathcal{D}$ is inferable from positive data only.*

Proof To derive a contradiction suppose that $\mathcal{D}$ has infinite elasticity. Let $w_0, w_1, w_2, \ldots$ be a sequence of strings required in the definition of finite elasticity, and let $L_1, L_2, L_3, \ldots$ be the corresponding sequence of deterministic pure languages.

Consider, for some $n > 1$, the subset $W_{n-1} = \{w_0, w_1, \ldots, w_{n-1}\}$ and the language L_n such that $W_{n-1} \subseteq L_n$ and $w_n \notin L_n$. Let $G_n = (\Sigma, P_n, s_n)$ be a deterministic pure grammar generating L_n. Since we do not allow productions of the form $a \to \lambda$, the length of s_n is bounded by the minimum length of strings in W_{n-1}. Hence, there are only a finite number of possible axioms in grammars $G_1, G_2, G_3, \ldots$.

For at least one axiom s there exist an infinite number of grammars using this axiom. These grammars have a (growing) subset of common strings. On the other hand, the number of productions in each G_i is bounded by the cardinality of Σ since we consider deterministic pure languages. Clearly, such an infinite sequence of deterministic pure grammars (and languages) with a bounded number of productions cannot exist. Thus, $\mathcal{D}$ cannot have infinite elasticity, and it is inferable from positive data only. □

We end this section by discussing pure grammars and languages which are both deterministic and k-uniform. The class of such languages is denoted by $\mathcal{D}(k)$, $k \geq 2$.

Given k and the alphabet Σ, there are only a finite number of possible production sets for a k-uniform, deterministic pure grammar. Let $\mid \Sigma \mid$ stand for the cardinality of Σ. For each a in Σ, there is at most one production with a in the left hand side. The number of possible right hand sides is $\mid \Sigma \mid^k$. Hence, there are only $(\mid \Sigma \mid^k + 1)^{|\Sigma|} - 1$ possible sets of productions. Here k and $\mid \Sigma \mid$ can be considered as constants. This leaves us with the problem of finding the proper axiom.

The "proper" axiom is, of course, the longest word over Σ having the property that all sample words so far received can be generated from it by using the production set in question. Since the number of possible production sets is indeed a constant, we can suppose that we know the correct production set. Repeating the procedure of searching the axiom for each possible production set naturally increases the constant coefficient of the time complexity, but it does not effect to the asymptotic growth rate.

Suppose now that the sample contains two words $a_1 a_2 \ldots a_m$ and $b_1 b_2 \ldots b_n$. Given the set of productions, what is the longest axiom from which the two words can be generated? A straightforward approach is to step backwards from the words according to the given productions until a common predecessor is found. Hence, we find out all the matches of the right hand sides of the given productions in $a_1a_2\dots a_m$ and $b_1b_2\dots b_n$, replace the occurrences of the right hand sides with the corresponding left hand sides, and store the words so obtained in data structures T_a and T_b, respectively. This is repeated until T_a and T_b contain a common word, the longest possible axiom.

A concise data structure for representing T_a is an automaton which accepts the possible axioms. Such an automaton A can be defined as $A = (Q, (0,0), F, \delta)$, where $Q = \{(i,j) \mid i = 0,\dots,n, j = 0,\dots,n-1\}$ is the set of states, $(0,0)$ is the initial state, $F = \{(n,j) \mid j = 0,\dots,n-1\}$ is the set of final states, and the transition relation δ is recursively defined as follows:

1. for each $i = 0,\dots,n-1$ and for each $j = 0,\dots,n-1$, $\delta((i,j), a_{j+1}) \ni (i, j+1)$
2. if $\delta((i,j),a) \ni (i',j')$, $\delta((i',j'),b) \ni (i'',j'')$, $j'' < n-1$ and $c \to ab$ is a production, then $\delta((i,j),c) \ni (i'', j''+1)$.

Note that the time needed for constructing this automaton representation is bounded by a polynomial in n.

The longest possible axiom is not necessarily unique. When a new sample word is received and the conjecture is to be updated, we represent the old sample words by the set of all possible axioms, and repeat the above procedure for finding the new axiom.

As an example, consider $(ab)^nc$ as the input word. Let $a \to ba$, $b \to ab$, $c \to ab$ be productions. It is easy to see that each word in $\{b,c\}^nc$ is a possible axiom. Hence, the number of possible axioms can be exponential in n.

We pose it as an open problem whether or not there exists a polynomial time inference algorithm for $\mathcal{D}(k)$ using positive data only. On the spirit of the previous discussion, the polynomial time inference algorithm would need an efficient method for constructing the intersection of two languages acceptable by automata of the type defined above.

However, we have an affirmative answer in a special case. Namely, if the length of the axiom is bounded by a constant, then deterministic, k-uniform pure languages are polynomial time inferable from positive data only.

Moreover, if the length of the axiom is bounded, then we even have the following stronger result. Let d be a fixed integer and $\mathcal{D}_d(k)$ be the class of languages generated by pure deterministic k-uniform grammars whose axioms are of length at most d. We set $\mathcal{D}_d = \bigcup_{i=2}^{\infty} \mathcal{D}_d(i)$.

Theorem 5.3 *$\mathcal{D}_d$ is polynomial time inferable from positive data only.*

Proof We know that $\mathcal{D}_d(k)$ is inferable from positive data only for any fixed k. We need only to infer the value of k. For each L in $\mathcal{D}_d$, there exist integers c_1 and $c_2 \leq d$ such that $LS(L) = \{c_1 \cdot n + c_2 \mid n \geq 0\}$. To infer the value of k, we need only to calculate the minimum absolute value of $lg(w_1) - lg(w_2)$ over any two words of different length presented so far. Moreover, k is at most $O(\log N)$, where N is the total length of the positive samples presented. □

6 Concluding remarks

Pure (context-free) languages are not inferable from positive data. However, natural subclasses of pure languages obtained by restricting the length of the right hand sides in the productions or the number of productions are inferable from positive data or the number of productions. We have shown the existence of such inference algorithms for k-uniform pure languages and for deterministic pure languages. Moreover, we have posed open whether there exists a polynomial time inference algorithm for deterministic, k-uniform pure languages using positive data only.

References

[1] D. Angluin, Finding patterns common to a string, *J. Comput. Syst. Sci.* **21** (1980), 46–62.

[2] D. Angluin, Inductive inference of formal languages from positive data, *Inform. Contr.* **45** (1980), 117–135.

[3] D. Angluin, Inference of reversible languages, *J. ACM* **29** (1982), 741–765.

[4] D. Angluin and C.H. Smith, Inductive inference: theory and methods, *ACM Comput. Surv.* **15** (1983), 237–269.

[5] W. Bucher and J. Hagauer, It is decidable whether a regular language is pure context-free. *Theoret. Comput. Sci.* **26** (1983), 233–241.

[6] S. Crespi-Reghizzi, G. Guida, and D. Mandrioli, Noncounting context-free languages, *J. ACM* **25** (1978), 571-580.

[7] A. Gabrelian, Pure grammars and pure languages, *Intern. J. Comput. Math.* **9** (1981), 3–16.

[8] P. Garcia, E. Vidal and J. Oncina, Learning locally testable languages in the strict sense, in: *Proceedings of the First International Workshop on Algorithmic Learning Theory* (1990), 325–338.

[9] E.M. Gold, Language identification in the limit, *Inform. Contr.* **10** (1967), 447–474.

[10] J. Hagauer, A simple variable-free CF grammar generating a non regular language. *Bull. EATCS* **6** (1978), 28–33.

[11] M.A. Harrison, *Introduction to Formal Language Theory* (Addison-Wesley, 1978).

[12] T. Koshiba, E. Mäkinen, and Y. Takada, Learning deterministic even linear languages from positive data, *Theoret. Comput. Sci.* **185** (1997), 63–79.

[13] E. Mäkinen, Inferring uniquely terminating regular languages from positive data, *Inf. Process. Lett.* **62** (1997), 57–60.

[14] H.A. Maurer, A. Salomaa, and D. Wood, Pure grammars, *Inform. Contr.* **44** (1980), 47–72.

[15] T. Motoki, T. Shinohara, and K. Wright, The correct definition of finite elasticity: Corrigendum to identification of unions, in: *Proceedings of 4th Workshop on Computational Learning Theory* (1991), 375.

[16] N. Tanida and T. Yokomori, Inductive inference of monogenic pure contex-free languages, *Lecture Notes in Computer Science* **872** (1994), 560–573.

[17] K. Wright, Identification of unions of languages drawn from an identifiable class, in: *Proceedings of 2nd Workshop on Computational Learning Theory* (1989), 328–333.

[18] T. Yokomori, Inductive inference of 0L languages, in: G. Rozenberg and A. Salomaa (eds.), *Lindenmayer Systems: Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology*, Springer, 1992, 115–132.

Received April, 1998

On inferring zero-reversible languages

Erkki Mäkinen *

Abstract

We use a language-theoretic result for zero-reversible languages to show that there exists a linear time inference method for this class of languages using positive data only.

1 Introduction

Regular languages cannot be inferred from positive data only [3]. This negative result has initiated a search for subclasses of regular languages having the desirable inference property. Several subclasses of regular languages allow inference algorithms based on merging nonterminals (or states in finite automata); such algorithms are surveyed in [5]. In this paper we consider zero-reversible languages, a well-known subclass of regular languages inferable from positive data only by using a merging algorithm.

We assume a familiarity with the basics of formal language theory and grammatical inference as given e.g. in [4] and [2], respectively. As inference criterion we use "identification in the limit" [3, 2].

If not otherwise stated we follow the notations and definitions of [4]. The empty word is denoted by λ, the reverse of a string $w = w_1w_2\dots w_n$ by w^R ($= w_nw_{n-1}\dots w_1$), and the left-quotient of L and w by $T_L(w) = \{v \mid wv \in L\}$.

We consider here regular languages and grammars only. We also suppose that grammars are reduced [4], i.e. each terminal and nonterminal appears at least in one derivation from the start symbol to a terminal word. A production of the form $A \to b$, where b is a terminal, is said to be *terminating*. A *continuing* production has the form $A \to bB$, where b is a terminal and B is a nonterminal. Other forms of productions are not allowed (except $S \to \lambda$, where S is the start symbol). A production with a nonteminal A on the left hand side is said to be an *A-production.*

2 Zero-reversible languages

Recall that a finite automaton A is zero-reversible if the following conditions hold [1]:

*Department of Computer Science, University of Tampere, P.O. Box 607, FIN-33101 Tampere, Finland, e-mail: em@cs.uta.fi

1. A is deterministic.

2. A is reset-free, i.e. for no two distinct states q_1 and q_2 do there exist an input symbol b and a state q_3 such that $\delta(q_1, b) = q_3 = \delta(q_2, b)$, where δ is the transition function of A.

3. A has at most one final state.

A regular language L is zero-reversible if there exists a zero-reversible finite automaton accepting L. We denote the class of zero-reversible languages as $\mathcal{R}(0)$.

Angluin's inference algorithm [1] for $\mathcal{R}(0)$ starts with a prefix tree automaton and proceeds by merging states as long as the conditions (i) - (iii) are not satisfied. It follows that the time complexity for outputting the next conjecture is not linear, but it has a small nonlinear factor.

The following purely language-theoretic characterization is also useful.

Proposition 2.1 *[1] A regular language L is zero-reversible if and only if whenever u_1v and u_2v are in L, then $T_L(u_1) = T_L(u_2)$.*

A regular grammar $G = (V, \Sigma, P, S)$ is said to be *deterministic* if, for each nonterminal A, the right hand sides of A-productions start with unique terminals. Given a nonterminal A and a sequence w of terminal symbols in a deterministic grammar, $A \Rightarrow^+ wB$ is possible for at most one symbol B in $(V \setminus \Sigma) \cup \{\lambda\}$. The concept of *backward determinism* is related to a somewhat opposite situation.

G is said to be backward deterministic, if $A \Rightarrow^+ w$ and $B \Rightarrow^+ w$, where $w \in \Sigma^+$, always implies $A = B$. Hence, in a backward deterministic grammar each terminal string is possible to generate from at most one nonterminal. Notice that a backward deterministic grammar is not necessarily deterministic.

A language L is backward deterministic if there exists a backward deterministic grammar generating L. The class of backward deterministic languages is denoted by $\mathcal{B}$.

Notice that in backward deterministic grammars terminating productions have unique right hand sides. Namely, if we have $A \Rightarrow a$ and $B \Rightarrow a$, where $a \in \Sigma$, then we have $A = B$. Similarly, if we have

$$A = A_0 \Rightarrow a_1 A_1 \Rightarrow \ldots \Rightarrow a_1 \ldots a_{n-1} A_{n-1} \Rightarrow a_1 \ldots a_{n-1} a_n$$

and

$$B = B_0 \Rightarrow a_1 B_1 \Rightarrow \ldots \Rightarrow a_1 \ldots a_{n-1} B_{n-1} \Rightarrow a_1 \ldots a_{n-1} a_n,$$

then we have $A_i = B_i$, for $i = 0, \ldots, n-1$.

Theorem 2.1 $\mathcal{R}(0) \subseteq \mathcal{B}$.

Proof Let L be zero-reversible. Suppose that $A \Rightarrow^+ w$ and $B \Rightarrow^+ w$ are possible in a regular grammar G generating L. Let $S \Rightarrow^+ u_1A$ and $S \Rightarrow^+ u_2B$ be derivations in G. We have u_1w and u_2w in L, and since L is zero-reversible, $T_L(u_1) = T_L(u_2)$. In other words, everything derivable from A is also derivable from B, and vice versa. Thus, we can replace all appearances of B in G by A without changing the language generated. This can be repeated with all pairs of nonterminals generating a common terminal string. Hence, there is a backward deterministic grammar for L. □

The inclusion in Theorem 2.1 is proper since there are nondeterministic languages in $\mathcal{B}$. However, all deterministic languages in $\mathcal{B}$ are zero-reversible. Namely, if we have u_1v and u_2v in a deterministic language L in $\mathcal{B}$, in the corresponding backward deterministic grammar we have

$$S \Rightarrow^+ u_1A \Rightarrow^+ u_1v$$

and

$$S \Rightarrow^+ u_2B \Rightarrow^+ u_2v,$$

for some nonterminals A and B. Now $A \Rightarrow^+ v$ and $B \Rightarrow^+ v$ must imply $A = B$. And further, since L is deterministic, $T_L(u_1) = T_L(u_2)$. Hence, L is zero-reversible by Proposition 2.1. We have proved the following theorem.

Theorem 2.2 *If a deterministic language L is in $\mathcal{B}$, then L is zero-reversible.*

3 The new algorithm

Our new algorithm is based on Theorem 2.1. Contrary to Angluin's algorithm [1], we start with a suffix automaton (a trie containing the suffixes), since we consider terminal strings derivable from nonterminals. In a reduced regular grammar such strings are always suffixes.

As an example, consider a sample { 0, 00, 11, 1100 } (cf. [1, Example 29]). We have the following derivations:

$$S \Rightarrow 0$$

$$S \Rightarrow 0A_1 \Rightarrow 00$$

$$S \Rightarrow 1A_2 \Rightarrow 11$$

$$S \Rightarrow 1A_3 \Rightarrow 11A_4 \Rightarrow 110A_5 \Rightarrow 1100.$$

The corresponding trie is shown in Figure 1. Nodes with at least one ending word are drawn as squares. Each node (except the root) has a set of nonterminals associated with it.

The nonterminals associated with the same node are merged. The nonterminal with the smallest subscript is chosen to be the canonical element, i.e. the one used as the representative of the merged nonterminals. We assume that $S = A_0$.

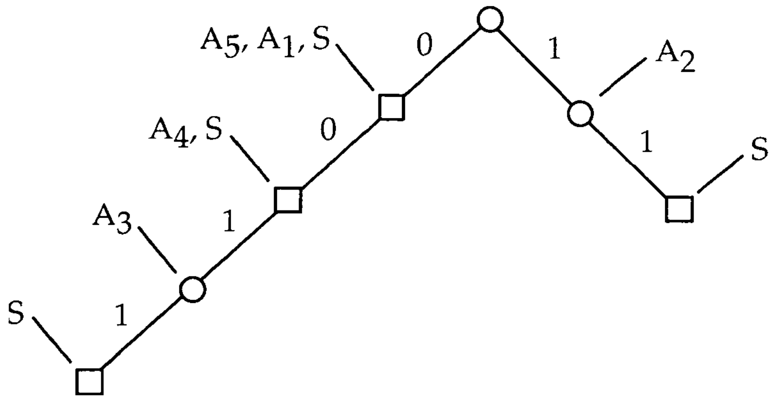


Figure 1: The trie for the sample { 0, 00, 11, 1100 }.

The productions of the resulting grammar can be read by traversing the edges from the leaves to the root. We obtain the productions

$$S \to 1A_3, A_3 \to 1S, S \to 0S, S \to 0, S \to 1A_2, A_2 \to 1.$$

Notice that we do not merge nonterminals A_2 and A_3, although we have productions $S \to 1A_2$ and $S \to 1A_3$. The corresponding states in the resulting finite automaton are merged in Angluin's algorithm [1].

Figure 2 shows the trie after reading the next sample 101. The corresponding derivation is

$$S \Rightarrow 1A_6 \Rightarrow 10A_7 \Rightarrow 101.$$

We merge A_2 and A_7. Notice that merging A_2, A_3, and A_7 would imply a further merge (S and A_2), a finally, a grammar equivalent with the finite automaton shown in Figure 5(d) of [1].

We can formulate our algorithm as follows.

Algorithm 3.1 (BZR) ***Input:*** *A non-empty sample* $T = \{w_1, w_2, \ldots, w_n\}$. ***Output:*** *A backward deterministic grammar G.*

1. *Insert the strings* $w_1^R, w_2^R, \ldots, w_n^R$ *to an initially empty string.*

2. *Associate the nonterminals from the derivations corresponding to the sample words to the nodes of the trie.*

3. *Merge the nonterminals appearing in each node. Choose the nonterminal with the smallest subscript as the the canoninal nonterminal (where* $S = A_0$*).*

4. *Read the resulting productions from the trie by traversing the edges from the leaves to the root. If a node is associated with* A_i*, its parent is associated with* A_j*, and the edge between the two nodes is labelled with a, the production*

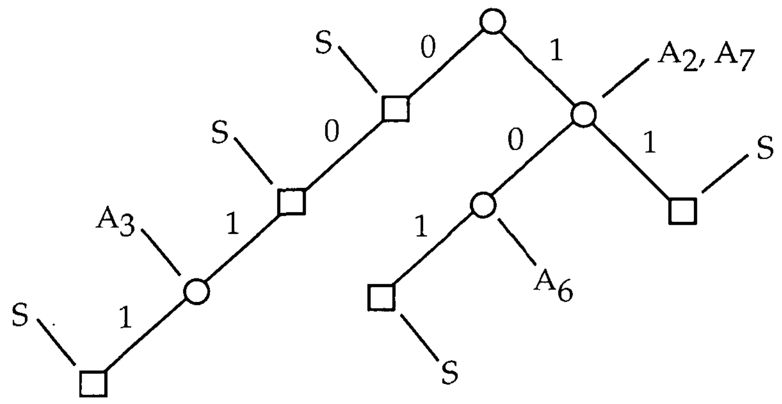


Figure 2: The trie after reading the next input word 101.

obtained is $A_i \to aA_j$. If a child of the root is associated with A and the edge between the nodes is labelled with a, we obtain a terminating production $A \to a$.

5. *If λ is in T, then insert the production $S \to \lambda$ to G.*

If the input sample contains words u_1v and u_2v, the algorithm guarantees that in the output grammar G, v is derivable from a single nonterminal only. Hence, G is backward deterministic. Moreover, by Proposition 2.1 and by the fact that sample words are from a zero-reversible language, $L(G)$ is zero-reversible. It is clear that $L(G)$ is the smallest zero-reversible language containing the sample. Hence, $L(G)$ coincides with the language produced by Angluin's inference algorithm [1] for zero-reversible languages.

Notice that the grammar outputted by BZR is not necessarily deterministic. However, a corresponding deterministic grammar must exist since the language generated is in $\mathcal{R}(0)$. We have simply left some of the merges of Angluin's algorithm undone.

BZR runs in time $O(n)$, where n is the sum of the lengths of the input words. Hence, we have the following theorem.

Theorem 3.1 *$\mathcal{R}(0)$ is inferable in linear time from positive data only.*

The space complexity of BZR is also linear. The trie contains less than n nodes, and it is sufficient to maintain one nonterminal (the canonical one) associated with a node.

Grammars obtained by BZR are bigger (have more productions) than those corresponding the finite automata produced by Angluin's algorithm. The bigger size of the resulting grammar seems to be the cost of dropping the nonlinear factor from the time complexity.

4 k-reversible languages

Proposition 2.1 has the following generalization in the case $k \geq 0$:

Proposition 4.1 *[1] A regular language L is k-reversible if and only if whenever u_1vw and u_2vw are in L and $lg(v) = k$, then $T_L(u_1v) = T_L(u_2v)$.*

It is possible to apply the approach of the previous section also to the case $k > 0$. However, the simplicity of the algorithm is lost in this case. Namely, we should maintain links between the derivations corresponding to the sample words and the nonterminals associated with the nodes in the trie, since merging is possible only when the condition of Proposition 4.1 is fulfilled.

Acknowledgements. This work was supported by the Academy of Finland (project 35025).

References

[1] D. Angluin, Inference of reversible languages, *J. ACM* **29** (1982), 741–765.

[2] D. Angluin and C.H. Smith, Inductive inference: theory and methods, *ACM Comput. Surv.* **15** (1983), 237–269.

[3] E.M. Gold, Language identification in the limit, *Inform. Contr.* **10** (1967), 447–474.

[4] M.A. Harrison, *Introduction to Formal Language Theory* (Addison-Wesley, 1978).

[5] E. Mäkinen, Inferring regular languages by merging nonterminals, *Intern. J. Computer Math.* **70** (1999), 601–616.

Received October, 1998

F codes

Marek Michalik [*]

Abstract

The notion of an F code is introduced as a generalization of the notion of an L code. All interrelations between ordinary codes of bounded delay, L codes of bounded delay and F codes are established. Attention is also focused on unary morphisms. Many of them are F codes.

1 Introduction

Consider a nonerasing morphism $h : A^* \to A^*$, where A is a finite alphabet. We emphasise that all morphisms discussed in this paper are nonerasing, that is, $h(a) \neq 1$ (the empty word) for every $a \in A$. A morphism h is a code if h is injective. We will denote by C the class of all codes. A morphism h is an L code if the function $\bar{h}$ given by

$$\bar{h}(a_1 a_2 \ldots a_n) = h(a_1)h^2(a_2)\ldots h^n(a_n)$$

($a_i \in A$ and $h^i(a_i)$ is the i-th iterate of the morphism h) is injective. For a positive integer k and a word w, we denote by $\mathrm{pref}_k(w)$ the prefix (initial subword) of w of length k. If a word w is shorter then k, then $\mathrm{pref}_k(w) = w$. The first letter of the word w, we denote by $\mathrm{first}(w)$. A morphism h is of *bounded delay* k if, for all words u and w, the equation

$$\mathrm{pref}_k(h(u)) = \mathrm{pref}_k(h(w))$$

implies the equation $\mathrm{first}(u) = \mathrm{first}(w)$. A morphism h is of *bounded delay* if it is of bounded delay k, for some k. A morphism h is of *weakly bounded delay* k if, for all words u and w, the equation

$$\mathrm{pref}_k(\bar{h}(u)) = \mathrm{pref}_k(\bar{h}(w))$$

implies the equation $\mathrm{first}(u) = \mathrm{first}(w)$. If for all $i \geq 0$, the equation

$$\mathrm{pref}_k(h^i\bar{h}(u)) = \mathrm{pref}_k(h^i\bar{h}(w))$$

implies the equation $\mathrm{first}(u) = \mathrm{first}(w)$, then h is of *strongly bounded delay* k. In general, h is of weakly or strongly bounded delay if it is so for some k. A morphism

[*]Faculty of Applied Mathematics, University of Mining and Metallurgy, Al. Mickiwicza 30, 30-059 Cracow, Poland. e-mail: grmichal@kinga.cyf-kr.edu.pl

h is of *medium bounded delay* if, for some recursive function f and all $i \geq 0$, u and w, the equation

$$\text{pref}_{f(i)}(h^i\bar{h}(u)) = \text{pref}_{f(i)}(h^i\bar{h}(w))$$

implies the equation $\text{first}(u) = \text{first}(w)$. Morphism h is a prefix code if for every different words u, w $h(u)$ is not a prefix of $h(w)$. We will denote by L, B, W, S, M, P the corresponding classes of the morphisms.

The next diagram due to [1] shows all inclusion between the classes introduced. The arrow stands for strict inclusion.

$$\begin{tikzcd}
 & & & C \arrow[r] & L \\
S \arrow[r] & P \arrow[r] & B \arrow[ur] \arrow[dr] & & \\
 & & & M \arrow[r] \arrow[uur] & W
\end{tikzcd}$$

2 F codes

From now on, f, g denote functions $\mathbb{N} \to \mathbb{N}$. We say that $f \prec g$ if there exists $n_0 \in \mathbb{N}$ such that $f(n) = g(n)$ for $n < n_0$ and $f(n_0) < g(n_0)$.

We will use the symbol $\hat{h}_f$ to denote function $\hat{h}_f : A^* \to A^*$ given by

$$\hat{h}_f(a_1a_2...a_n) = h^{f(1)}(a_1)h^{f(2)}(a_2)...h^{f(n)}(a_n)$$

($a_i \in A, h^{f(i)}(a_i)$ is the $f(i)$-th iterate of the morphism h) We call the morphism h an F code if there exists a minimal function $f : \mathbb{N} \to \mathbb{N}$ such that $\hat{h}_f$ is injective. We will denote by F the class of all F codes. It is easy to see that every L code is an F code.

If function $\bar{h}$ is injective then there exists a minimal function f such that $\hat{h} := \hat{h}_f$ is injective. Thus we conclude that every L code is an F code. We show that $F - L \neq \emptyset$.

Lemma 2.1 *Let* $A = \{a_1, \ldots, a_n\}, h : A^* \to A^*$,
$\text{min}_k := \min\{||h^k(a_i)| - |h^k(a_j)||, |h^k(a_i)| : i \neq j; i,j \in \{1, \ldots, n\}\}$,
$\text{max}_k := \max\{|h^k(a_i)| : i \in \{1, \ldots, n\}\}$.
If for each $n \in N$ *there exists* k*, such that* $\text{min}_k > n$ *then we can define the function* f *as follows*
$f(1) = \min\{k : \text{min}_k > 0\}$, $d_1 := \text{max}_{f(1)}$,
$\forall i \in N$ $f(i+1) := \min\{k : \text{min}_k > d_i\}$, $d_{i+1} := \text{max}_{f(i+1)} + d_i$
The function $\hat{h}_f$ *is injective.*

Proof. It suffices to prove that different words have different length. The proof of this is by induction on word length. By the definition of $f(1)$ we have $||\hat{h}_f(a_i)| -$

$|\hat{h}_f(a_j)|| > 0$ for all $a_i, a_j \in A, i \neq j$. Let $w = a_1 \ldots a_k, u = a'_1 \ldots a'_k a'_{k+1}$. It is clear that $|\hat{h}_f(u)| > |h^{f(k+1)}(a'_{k+1})| > d_k \geq |\hat{h}_f(w)|$. The proof is completed by showing that for all $u, w \in A, |u| = |w| = k+1$, it holds that $||\hat{h}_f(u)| - |\hat{h}_f(w)|| > 0$. Consider $w = a_1 \ldots a_{k+1}, u = a'_1 \ldots a'_{k+1}$ and $a_{k+1} \neq a'_{k+1}$. We see at once that $||h^{f(k+1)}(a_{k+1})| - |h^{f(k+1)}(a'_{k+1})|| > d_k, |\hat{h}_f(a_1 \ldots a_k)| \leq d_k, |\hat{h}_f(a'_1 \ldots a'_k)| \leq d_k$, thus $||\hat{h}_f(a_1 \ldots a_{k+1})| - |\hat{h}_f(a'_1 \ldots a'_{k+1})|| > 0$ and finally $||\hat{h}_f(u)| - |\hat{h}_f(w)|| > 0$. □

Lemma 2.2 *Let* $A = \{a_1, \ldots, a_n\}, h : A^* \to A^*$,
$\min_k := \min\{||h^k(a_i)| - |h^k(a_j)||, |h^k(a_i)| : i \neq j; i, j \in \{1, \ldots, n\}\}$.
If the sequence $\min_k$ *is not bounded then morphism* h *is an* F *code.*

Proof. Let f be defined as in lemma 2.1. There exists a minimal function g such that $\hat{h}_g$ is injective, thus $h \in F$. □

Theorem 2.3 *The class* L *is strictly included in the class* F.

Proof. Every L code is an F code. Let $h: \{a, b\}^* \to \{a, b\}^*$ be given by $h(a) = a^2$, $h(b) = a^6$. Morphism h is not an L code ($\bar{h}(aa) = \bar{h}(b)$). We have $\min_k = 2^k$, hence by lemma 2.2, $h \in F$. □

Remark 2.4 *For the morphism* $h: \{a, b\}^* \to \{a, b\}^*$ *given by* $h(a) = a^2$, $h(b) = a^6$ *the function* $f(i) = 2i - 1$ *is a minimal function such that* $\hat{h}_f$ *is injective.*

Proof. To prove this, we observe that for every function $p(i) = n_i$ such that $\hat{h}_p$ is an injective function we have $\forall i \neq j\ (n_i \neq n_j) \wedge (|n_i - n_j| \neq 1)$ (as a consequence of $h^n(ab) = h^n(ba)$ and $h^n(a)h^{n+1}(a) = h^n(b)$), hence p is minimal. □

It is easy to check that if $h : A^* \to A^*$ is an F code then $h_{|A} : A \to A^*$ is injective. The reverse implication is not true. Let $A = \{a, b, c, d\}$ and h be given by $h(a) = b, h(b) = a, h(c) = bd, h(d) = d$. Function $h_{|A}$ is injective. For every $f : N \to N$, $\hat{h}_f(ad) = \hat{h}_f(c)$, hence $h \notin F$.
Similarly to L codes we have

Remark 2.5 *The composition of two* F *codes is not necessarily an* F *code.*

Proof. Consider the morphisms g and h defined by $g(a) = ab, g(b) = ba, h(a) = a^2, h(b) = a$. Clearly g is a code and, hence, an F code. h is an L code. However, the composition $h \circ g \equiv a^3$ is not an F code. □

Theorem 2.6 *Classes* F *and* W *are incomparable.*

Proof. It suffices to show that $W \not\subseteq F$. Consider a morphism h given by $h(a) = edb, h(b) = b^2, h(c) = deb, h(d) = a, h(e) = a^3$, then $h \in W$ (see [1]). For all $i \geq 2$, $h^i(a) = h^i(c)$ and $h(ddd) = h(e)$, hence $h \notin F$. From this we conclude that W and F are incomparable. □

Now we can redraw the diagram as follows:

$$\begin{tikzcd}
 & & & C \arrow[r] & L \arrow[r] & F \\
S \arrow[r] & P \arrow[r] & B \arrow[ur] \arrow[dr] & & & \\
 & & & M \arrow[uur] \arrow[r] & W &
\end{tikzcd}$$

If we restrict our considerations to the class of morphisms $h : \{a, b\}^* \to \{a, b\}^*$ we obtain

Theorem 2.7 *$B = C = M = W$ and*

$$\begin{tikzcd}
S \arrow[r] & P \arrow[r] & C \arrow[r] & L \arrow[r] & F
\end{tikzcd}$$

Proof. Let $h(a) = ba$, $h(b) = b^2$. $h \in P \setminus S$ (see [1]). Morphism $h(a) = a$, $h(b) = ab$ is a code but not a prefix code. Morphism $h(a) = a^2$, $h(b) = a$ is an element of $L - C$. From theorem 2.3 we obtain $F \setminus L \neq \emptyset$. To complete the proof it suffices to show that $C \subset B$ and $W \subset C$.
($C \subset B$) Every prefix code is of bounded delay. Let $h(a) = x$, $h(b) = xy$, $x, y \in A^+, x \neq y$. Morphism h is of bounded delay $k = 2|x| + |y|$.
($W \subset C$) Let $h \notin C$, then there exist $n, m \in \mathbb{N}$ such that $h(a) = x^m$, $h(b) = x^n$ thus $h \notin W$. □

3 F codes and the unary morphism

Theorem 3.1 *Let $A = \{a_1, \ldots, a_n\}$, $a \in A$, $h : A^* \to \{a\}^*$,*
$\min_k := \min\{||h^k(a_i)| - |h^k(a_j)||, |h^k(a_i)| : i \neq j; i, j \in \{1, \ldots, n\}\}$.
The unary morphism h is an F code if and only if the sequence $\min_k$ is not bounded.

Proof. If sequence $\min_k$ is not bounded then from lemma 2.1 $h \in F$.
Consider $h \in F$. Suppose that there exists $M \in \mathbb{N}$ such that for all $k \in \mathbb{N}$ $\min_k \leq M$. The morphism h is a nonerasing morphism, thus for all $a_i \in A$ and $n \in \mathbb{N}$ we have $|h^n(a_i)| \leq |h^{n+1}(a_i)|$. If $\min_k \leq M$ then $\exists n_0 \in \mathbb{N} \forall t \geq n_0 \forall i \in \{1, \ldots, n\}$ $|h^t(a_i)| = |h^{t+1}(a_i)|$. We have $a^p = h^t(a_i) = h^{t+1}(a_i) = h(h^t(a_i)) = h(a^p)$ for some $p \in \mathbb{N}$. This implies $h(a) = a$ and finally $\hat{h}(a_i) = \hat{h}(a^{|\hat{h}(a_i)|})$ which contradicts $h \in F$. □

Remark 3.2 *The last theorem is not true for arbitrary morphism.*

Proof. Let $h(a) = a$, $h(b) = ab$, $h(c) = b$. For every $w, u_1, u_2, u_3 \in \{a, b, c\}^*$ $\bar{h}(wbu_1) \neq \bar{h}(wcu_2)$, $\bar{h}(wbu_1) \neq \bar{h}(wau_3)$, $\bar{h}(wcu_2) \neq \bar{h}(wau_3)$. To show this we observe that $\forall u_3, v, w \in A^*$ $\forall k \in \mathbb{N}$
$\text{pref}_k\{ah^{|w|+1}(\bar{h}(u_3))\} \notin \{a^{|w|+1}bv,\ a^{|w|}bv\}$. Thus h is an L code, but $\min_1 = 0$ and $\forall k > 1$ $\min_k = 1$. □

Corollary 3.3 *It is decidable whether the unary morphism is an F code.*

The morphism h is an almost L code if and only if h is not an L code and $\exists t \in \mathbb{N} \forall w, u \in A^*, \mathrm{first}(w) \neq \mathrm{first}(u)$ $\left(\bar{h}(w) = \bar{h}(u) \Rightarrow (|w| = t \vee |u| = t)\right)$

Remark 3.4 $F \setminus \mathrm{almost\,L} \neq \emptyset$.

Proof. Let $h(a) = a^3$, $h(b) = a^6$, $h(c) = a^2$, then $h^n(a) = a^{3^n}$, $h^n(b) = a^{6\cdot 3^{n-1}}$, $h^n(c) = a^{2\cdot 3^{n-1}}$, $\min_n = 3^{n-1}$. From lemma 2.2 we obtain $h \in F$. For every $w \in A^*$ $\bar{h}(aaw) = \bar{h}(bcw)$, thus $h \notin \mathrm{almost}L$. $\square$

Let U be the class of all unary morphisms $h : \{a, b\} \to \{a\}^*$ such that $h(a) = a^n$, $h(b) = a^r$, $n \neq r$, $n \geq 2$.

Theorem 3.5 $F \cap U = (L \cup \mathrm{almost\,L}) \cap \mathrm{U}$

Proof. It is clear that if $n = r$ or $n = 1$ then h is neither an F code nor an almost L code. If $n \neq r$ and $n \geq 2$ then the sequence $\min_k = \min\{|n^k - r\cdot n^{k-1}|, n^k, r\cdot n^{k-1}\}$ is not bounded. This implies $h \in F$. Every unary morphism such that $h(a) = a^n$, $h(b) = a^r$, $n \neq r$, $n \geq 2$ is either an L code or an almost L code (see [3]). $\square$

References

[1] H.A. Maurer, A. Salomaa, D. Wood, *Bounded delay L codes* (Theoretical Computer Science 84 (1991) 265-279)

[2] J. Honkala and A. Salomaa, *Characterization results about L codes* (Informatique theorique et Applications vol. 26, $n°3$, (1992) 287-301)

[3] H.A. Maurer, A. Salomaa, D. Wood, *L codes and number systems* (Theoretical Computer Science 22 (1983) 331-346)

Received January, 1999

Acta Cybernetica 14 (1999) 491-495.

Note on the Cardinality of some Sets of Clones

Jovanka Pantović [*] Dušan Vojvodić [†]

Abstract

All minimal clones containing a three-element grupoid have been determined in [3]. In this paper we solve the problem of the cardinality of the set of clones which contain some of these clones.

1 Notation and Preliminaries

Denote by $\mathbf{N}$ the set $\{1,2,\ldots\}$ of positive integers and for $k,n \in \mathbf{N}$, set $E_k = \{0,1,\ldots,k-1\}$. We say that f is an *i-th projection of arity* n $(1 \leq i \leq n)$ if $f \in P_k^{(n)}$ and f satisfies the identity $f(x_1,\ldots,x_n) \approx x_i$.

For $n,m \geq 1, f \in P_k^{(n)}$ and $g_1,\ldots,g_n \in P_k^{(m)}$, *the superposition of* f and $g_1,\ldots,g_n$, denoted by $f(g_1,\ldots,g_n)$, is defined by $f(g_1,\ldots,g_n)(a_1,\ldots,a_m) = f(g_1(a_1,\ldots,a_m),\ldots,g_n(a_1,\ldots,a_m))$ for all $(a_1,\ldots,a_m) \in E_k^m$. A set

A set C of operations on E_k is called a *clone* if it contains all the projections and is closed under superposition.

For an arbitrary set F of operations on E_k there exists the least clone containing F. This clone is called *the clone generated by* F, and will be denoted by $\langle F\rangle_{\mathrm{CL}}$. Instead of $\langle\{f\}\rangle_{\mathrm{CL}}$ we will write simply $\langle f\rangle_{\mathrm{CL}}$. For a clone C and $n \geq 1$ we denote by $C^{(n)}$ the set of n-ary operations from C.

The clones on E_k form an algebraic lattice $Lat(E_k)$ whose least element is the clone of all projections and whose greatest element is the clone of all operations on E_k. The atoms (dual atoms) of $Lat(E_k)$ are called *minimal (maximal) clones.*

A full description of all clones for $k=2$ was given by Post, for $k=3$ a complete list of all maximal clones was found by Iablonskiĭ and all minimal clones were determined by Csákány.

Let h be a positive integer. A subset ρ of E_k^h (i.e. a set of h-tuples over E_k) is an h-ary *relation on* E_k. An n-ary operation f on E_k *preserves* ρ if for every $h \times n$ matrix $X = [x_{ij}]$ over E_k whose columns are all h-tuples from ρ we have $(f(x_{00},\ldots,x_{0(n-1)}),\ldots,f(x_{(h-1)0},\ldots,x_{(h-1)(n-1)})) \in \rho$. The set of all operations on E_k preserving a given relation ρ is denoted Polρ.

[*]Faculty of Engineering, University of Novi Sad, Trg Dositeja, Obradovića 3, 21000 Novi Sad, Yugoslavia, e-mail:pantovic@uns.ns.ac.yu

[†]Faculty of Science and Mathematics, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Yugoslavia, e-mail:vojvod@eunet.yu

Let $k = 3$ and let ϕ be a permutation of E_3. To each n-ary function f we assign f^ϕ, called *conjugate* of f, defined by $f^\phi(x_0, \ldots, x_{n-1}) = \phi(f(\phi^{-1}(x_0), \ldots, \phi^{-1}(x_{n-1}))$. The map $f \to f^\phi$ carries each clone C onto the clone C^ϕ; in particular $\langle f\rangle^\phi_{\mathrm{CL}} = \langle f^\phi\rangle_{\mathrm{CL}}$, and $g \in \langle f\rangle_{\mathrm{CL}}$ implies $g^\phi \in \langle f^\phi\rangle_{\mathrm{CL}}$. We can permute the variables of f as well: for a permutation ψ of E_n put $f_\psi(x_0, \ldots, x_{n-1}) = f(x_{\psi(0)}, \ldots, x_{\psi(n-1)})$. Remark that always $(f^\phi)_\psi = (f_\psi)^\phi$. Note also that $\langle f_\psi\rangle_{\mathrm{CL}} = \langle f\rangle_{\mathrm{CL}}$ for any ψ. The conjugations and permutations of variables generate a permutation group T_n of order $3n!$ on the set of all n-ary functions on E_3.

A binary idempotent function with Cayley table

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & n_5 & n_4 \\ 1 & n_3 & 1 & n_2 \\ 2 & n_1 & n_0 & 2 \end{array}$$

is denoted by b_n, where $n = n_0 + 3n_1 + 3^2n_2 + 3^3n_3 + 3^4n_4 + 3^5n_5$.

It is proved in [3] that every minimal clone on E_3 containing an essential binary operation is a conjugate of exactly one of the following twelve clones: $\langle b_i\rangle_{\mathrm{CL}}$ with $i \in \{0, 8, 10, 11, 16, 17, 26, 33, 35, 68, 178, 624\}$. The following table shows the binary functions on E_3 which generate minimal clones.

$xy \to$	00	01	02	10	11	12	20	21	22
b_0	0	0	0	0	1	0	0	0	2
b_8	0	0	0	0	1	0	2	2	2
b_{10}	0	0	0	0	1	1	0	1	2
b_{11}	0	0	0	0	1	1	0	2	2
b_{16}	0	0	0	0	1	1	2	1	2
b_{17}	0	0	0	0	1	1	2	2	2
b_{26}	0	0	0	0	1	2	2	2	2
b_{33}	0	0	0	1	1	0	2	0	2
b_{35}	0	0	0	1	1	0	2	2	2
b_{68}	0	0	0	2	1	1	1	2	2
b_{178}	0	0	2	0	1	1	2	1	2
b_{624}	0	2	1	2	1	0	1	0	2

2 Results

Theorem 2.1 *The cardinality of the set of clones on E_3 containing a conjugate of $\langle b_j\rangle_{\mathrm{CL}}, j \in \{0, 8, 11, 17, 33, 35\}$ is continuum.*

Proof. The proof is based on the operations of Janov–Mučnik.

We shall define a countable set of operations F and an operation g so that for all $f \in F$, $f \notin \langle (F \setminus \{f\}) \cup \{g\}\rangle_{\mathrm{CL}}$. This implies that for each $G, H \subseteq F$, from $G \neq H$ it follows $\langle G \cup \{g\}\rangle_{\mathrm{CL}} \neq \langle H \cup \{g\}\rangle_{\mathrm{CL}}$. In this way we get a set of distinct clones of a continuum cardinality.

For $i = 1, \dots, m$ denote by $\mathbf{e}_i$ the m-tuples $(1, \dots, 1, 2, 1, \dots, 1)$ with 2, at the i-th place. Let $A_m = \{\mathbf{e}_1, \dots, \mathbf{e}_m\}$.

For $m > 2$, consider the m-ary operation f_m (Janov–Mučnik,[5]) which takes the value 1 on A_m and 0 otherwise.

Modifying an idea which is attributed to Rónyai in [1], we define the relations $\rho_m \subseteq E_3^m$ on E_3 for $m > 2 : \rho_m = A_m \cup B_m$, where $B_m = \{(b_1, \dots, b_m) | b_j = 0$ for some $j, 1 \leq j \leq n\}$.

In what follows we prove that for each $i \neq m$ and $j \in \{0, 8, 11, 17, 33, 35\}$, f_i and b_j preserve ρ_m while f_m does not.

Let $X = [x_{ij}]$ be the $m \times m$ matrix with $x_{11} = \dots x_{mm} = 2$ and $x_{ij} = 1$ otherwise.The i-th column of X is $\mathbf{e}_i \in \rho_m (i = 1, \dots, m)$ while the values of f_m on the rows of X form $(f_m(\mathbf{e}_1), \dots, f_m(\mathbf{e}_m))^T = (1, \dots, 1)^T \notin \rho$. Hence, $f_m \notin Pol\rho_m$.

Suppose to the contrary that f_i doesn't preserve ρ_m for some $i \neq m$. Then there is an $m \times i$ matrix X with all columns in ρ_m and with rows $\mathbf{a}_1, \dots, \mathbf{a}_m$ such that $\mathbf{b} := (f_i(\mathbf{a}_1), \dots, f_i(\mathbf{a}_m))^T \notin \rho$. Since $im f_i = \{0, 1\}$ and $B_m \subset \rho_m$ clearly $\mathbf{b} = (1, \dots, 1)^T$. By the definition of f_i there exist $1 \leq j_1, \dots, j_m \leq i$ such that $\mathbf{a}_k = \mathbf{e}_{j_k}$ for all $k = 1, \dots, m$. If $j_k = j_l$ for some $1 \leq k < l \leq m$ then the j_k-th column of X contains at least two 2s and so does not belong to ρ_m. As $i \neq m$ we can choose $k \in \{1, \dots, i\} \setminus \{j_1, \dots, j_m\}$. Clearly, the k-th column of X is $(1, \dots, 1)^T \notin \rho_m$.

If $b_j, j \in \{0, 8, 11, 17, 33, 35\}$ does not preserve ρ then there exist $\mathbf{a}, \mathbf{b} \in \rho$ such that $(b_j(a_1, b_1), \dots, b_j(a_m, b_m)) \notin \rho$, i.e. $(b_j(a_1, b_1), \dots, b_j(a_m, b_m)) \in \{1, 2\}^m \setminus A_m$. It follows that $((b_j(a_1, b_1), \dots, b_j(a_m, b_m)) = \mathbf{a}$ since $b_j(a_l, b_l) = 1$ implies $a_l = 1$ and $b_j(a_l, b_l) = 2$ implies $a_l = 2$. So, we get a contradiction.

The set of clones of the form $\langle G \cup \{b_0, b_8, b_{11}, b_{17}, b_{33}, b_{35}\} \rangle_{\text{CL}}$, $G \subseteq \{f_2, f_3, \dots\}$ has a continuum cardinality. □

Theorem 2.2 *The cardinality of the set of clones on E_3 containing a conjugate of $\langle b_j \rangle_{\text{CL}}, j \in \{10, 16, 26, 68\}$ is at least $\aleph_0$.*

Proof.

Let $\{0, 1, 2\} = \{p, q, r\}$, and for $i = 1, \dots, m$ denote by $\mathbf{e}_i$ the m-tuples $(p, \dots, p, r, p, \dots, p)$ with r at the i-th place. Let $A_m = \{\mathbf{e}_1, \dots, \mathbf{e}_m\}$.

For $m > 2$, consider the m-ary operation f_m (similar to the Janov–Mučnik operations :

$$f_m(\mathbf{x}) = \begin{cases} p & if \;\; x \in A_m, \\ q & otherwise \end{cases} .$$

Define the following relations $\rho_m \subseteq E_3^m$ on E_3 for $m > 2 : \rho_m = E_3^m \setminus \{(p, \dots, p)\}$.

In what follows we prove that f_i preserves ρ_m if and only if $i > m$.

Suppose to the contrary that f_i doesn't preserve ρ_m for some $i > m$. Then there is an $m \times i$ matrix X with all columns in ρ_m and with rows $\mathbf{a}_1, \dots, \mathbf{a}_m$ such that $\mathbf{b} := (f_i(\mathbf{a}_1), \dots, f_i(\mathbf{a}_m))^T \notin \rho$, i.e. $\mathbf{b} = (p, \dots, p)^T$. By the definition of f_i, $\mathbf{a}_k = \mathbf{e}_{j_k}, 1 \leq j_k \leq i$, for all $k = 1, \dots, m$. Since $i > m$, $i - m + 1$ column has to be equal $(p, \dots, p)$, which gives a contradiction.

Let $i \leq m$ and $X = [x_{ij}]$ be the $m \times i$ matrix with $x_{lj} = p$ if $l \neq j$ and $x_{jj} = r$ for $j \in \{1, \dots, i-1\}, l \in \{1, \dots, m\}, x_{1i} = \dots = x_{(i-1)i} = p$ and $x_{ii} = \dots = x_{mi} = r$. The values of f_i on the rows of X form $(p, \dots, p) \notin \rho$.

We shall prove that b_{10} and b_{16} preserve ρ_m with $r = 1, p = 2$ and $q = 0$; b_{26} preserves ρ_m with $p = 1, q = 0$, and $r = 2$; and b_{68} preserves ρ_m with $p = 0, q = 2$, and $r = 1$.

Suppose to the contrary that $b_j, j \in \{10, 16, 26, 68\}$ does not preserve ρ_m. Then, there is an $m \times 2$ matrix with both columns in ρ_m such that $(b_j(x_1, y_1), \dots, b_j(x_m, y_m)) = (p, \dots, p)$. Therefore by the definition of b_j clearly $x_l = p, l \in \{1, \dots, m\}$ for each $j \in \{10, 16, 26, 68\}$. Thus, the first column of X is $(p, \dots, p)^T \notin \rho$, a contradiction.

So, we proved that for each $j \in \{10, 16, 26, 68\}$, the set $\{\bigcup_{m>2} f_m\}$ satisfies $\langle \bigcup_{i>m} \{f_m\} \cup \{b_j\} \rangle_{CL} \supset \langle \bigcup_{i>m+1} \{f_m\} \cup \{b_j\} \rangle_{CL} \supset \langle \bigcup_{i>m+2} \{f_m\} \cup \{b_j\} \rangle_{CL} \dots$, proving that there are at least $\aleph_0$ clones containing $\langle b_j \rangle_{CL}$. $\square$

It is still an open problem to determine a cardinality of the set of clones that contain a clone generated by b_{178} and b_{624}.

References

[1] Ágoston I., Demetrovics J., Hannák L., *The Number of Clones Containing all Constants (A Problem of R.McKenzie)* , Colloquia Mathematica Societatis Janos Bolyai, 43, Lectures in Universal Algebra, Szeged, Hungary, (1983), 21-25.

[2] Burris S., Sankappanavar H.P., *A Course in Universal Algebra*(Hungarian), Akademiai Kiadó, Budapest, 1989. (Springer-Verlag,GTM78,1981.)

[3] Csákány B., *Three-element grupoids with minimal clones* , Acta Sci. Math. (Szeged), v.45, 1983, pp. 111-117.

[4] Demetrovics J., Hannák L., *On the Number of Functionally Complete Algebras*, Proc. of the 12th Int. Symp. on Multiple-Valued Logic, 1982, 329-330.

[5] Iablonskiĭ S.V., *Introduction to Discrete Mathematics (Russian)*, Nauka, Moscow, 1979.

[6] Janov Ju. I., Mucnik A. A., *On the existence of k-valued closed classes that have no bases*, (Russian), Dokl. Akad. Nasuk SSSR 127 (1959), 44-46.

[7] Pantović J., Tošić R., Vojvodić G., *The Cardinality of Functionally Complete Algebras on a Three Element Set* , Algebra Univers. 38, 136-140.

[8] Rosenberg I.G., *Uber die funktionale Vollständigkeit in den mehrwertigen Logiken (Struktur der Funktionen von mehreren Veränderlichen auf endlichen Mengen)* , Rozpravy Československe Akad. Věd. Řada Mat. Přírod. Věd., V. 80, 1970, 3-93.

[9] Szabo L., *On Minimal and Maximal Clones* , Acta Cybernetica 10(1992) No.4, 323-327.

[10] Szendrei Á., *Simple surjective algebras having no proper subalgebras*, J. Austral. Math. Soc. Ser A, 48 (1990), 434-454.

Received April, 1998

Acta Cybernetica **14** (2000) 497-501.

A Note on the Equivalence of the Set Covering and Process Network Synthesis Problems*

B. Imreh† J. Fülöp‡ F. Friedler§

Abstract

In this paper, combining and completing some earlier results presented in this journal, it is proved that the Process Network Synthesis problem (PNS problem for short) is equivalent to the set covering problem.

Keywords: combinatorial optimization, network design, complexity.

1 Introduction

The foundations of PNS and the background of the combinatorial model studied here can be found in [6], [8], and [9]. Since the present work is based on the results of the papers [3], [7] published in this journal, we recall here only the main definitions briefly.

By *structural model* of PNS we mean a triplet (P, R, O) where P, R, O are finite sets, $\emptyset \neq P$ is the set of *desired products*, R is the set of *raw materials*, $\emptyset \neq O$ is the set of *available operating units*, furthermore $O \subseteq \wp'(M) \times \wp'(M)$ where M is the set of materials involved in the investigation and $\wp'(M)$ denotes the set of all nonempty subsets of M. It is assumed that $P \cap R = \emptyset$ and $M \cap O = \emptyset$. An operating unit, $u = (\alpha, \beta) \in O$, can be interpreted such that α and β contain the *input* and *output* materials of u, respectively. Pair (M, O) determines a directed graph called *process graph*. The set of vertices of this graph is $M \cup O$, and the set of arcs is $A = A_1 \cup A_2$ where $A_1 = \{(X, Y) : Y = (\alpha, \beta) \in O \text{ and } X \in \alpha\}$ and $A_2 = \{(Y, X) : Y = (\alpha, \beta) \in O \text{ and } X \in \beta\}$. If there exist vertices $X_1, X_2, ..., X_n$, such that $(X_1, X_2), (X_2, X_3), \ldots, (X_{n-1}, X_n)$ are arcs of process graph (M, O), then

*This work has been supported by the Ministry of Education of Hungary, Grant FKFP 0008/1999. The research of the second author was supported in part by Hungarian National Research Foundation, Grant OTKA T016413.

†Department of Informatics, József Attila University, Árpád tér 2, H-6720 Szeged, Hungary

‡Laboratory of Operations Research and Decision Systems, Computer and Automation Institute, Hungarian Academy of Sciences, H-1518 Budapest, P.O.Box 63, Hungary

§Department of Computer Science, University of Veszprém, Egyetem u. 10, H-8200 Veszprém, Hungary the path determined by these arcs is denoted by $[X_1, X_n]$. Obviously, for any suitable pair (m, o), one can consider the corresponding process graph.

Let process graphs (m, o) and (M, O) be given. (m, o) is defined to be a *subgraph* of (M, O), if $m \subseteq M$ and $o \subseteq O$.

For structural model (P, R, O), process graph (M, O) with $M \supseteq \cup\{\alpha \cup \beta : (\alpha, \beta) \in O\}$ presents the interconnections among the operating units of O. Furthermore, every feasible process network, producing the given set P of products from the given set R of raw materials using operating units from O, corresponds to a subgraph of (M, O). Therefore, examining the corresponding subgraphs of (M, O), we can determine the feasible process networks. If there is no further constraints such as material balance, then the subgraphs of (M, O) which can be assigned to the feasible process networks have the following common combinatorial properties (*cf.* [8]).

Subgraph (m, o) of (M, O) is called a *solution-structure* of (P, R, O) if:

$(S1)$ $P \subseteq m$,

$(S2)$ $\forall X \in m$, $X \in R \Leftrightarrow X$ is a source in (m, o),

$(S3)$ $\forall Y_0 \in o$, $\exists$ path $[Y_0, Y_n]$ with $Y_n \in P$,

$(S4)$ $\forall X \in m$, $\exists(\alpha, \beta) \in o$ such that $X \in \alpha \bigcup \beta$.

Let $S(P, R, O)$ denote the set of solution-structures of (P, R, O). It can be easily seen that any solution-structure (m, o) is uniquely determined by set o. By this observation, $S(P, R, O)$ can be considered as a family of subsets of O.

PNS problem with weigths

Let a PNS problems be given. Let us suppose that its every operating unit has a positive weight. Then, the following optimization problem can be studied:

We are to find a feasible process network with the minimal weight where by weight of a process network we mean the sum of the weights of the operating units belonging to the process network under consideration.

Each feasible process network in such a PNS problem is determined uniquely from the corresponding solution-structure and vice versa. Thus, the problem can be formalized as follows:

Let a structural model of PNS problem (P, R, O) be given. Moreover, let w be a positive real-valued function defined on O, the *weight function*. Then, the basic model is

$$(1) \qquad \min\{\sum_{u \in o} w(u) : (m, o) \in S(P, R, O)\}.$$

In what follows, the elements of $S(P, R, O)$ are called *feasible solutions* and by PNS problem we mean a PNS problem with weights.

2 Relationship between the PNS and the set covering problems

We recall (*cf.* [3]) that any set covering problem can be transformed into an equivalent PNS problem. On the other hand, it is proved in [7] that every cycle free PNS problem can be transformed into a suitable set covering problem. Now, using a similar argument as in [7], we extend this statement to an arbitrary PNS problem which results in the equivalence of the two problems considered.

This extension is proved in more steps.

First, let us observe that $S(P,R,O)$ is closed under the finite union. This results in the existence of a greatest feasible solution, $\cup\{(m,o) : (m,o) \in S(P,R,O)\}$, which is called *maximal structure* of the problem provided that $S(P,R,O) \neq \emptyset$. The existence of the maximal structure makes possible a reduction of PNS problem under consideration in the following way. Let a PNS problem be given by (P,R,O) and let us denote (M,O) the process graph belonging to it. Furthermore, let us suppose that $S(P,R,O) \neq \emptyset$, and let denote $(\bar{M},\bar{O})$ the corresponding maximal structure. Then, we can construct a new model given by $(P, R \cap \bar{M}, \bar{O})$. Since $(\bar{M},\bar{O})$ contains every feasible solution from $S(P,R,O)$, $S(P,R,O) = S(P,R \cap \bar{M},\bar{O})$. This new model is called the *reduced model* of problem considered. By the basis of the common set of feasible solutions, we obtain that problem

$$(2) \qquad \min\{\sum_{u \in o} w(u) : (m,o) \in S(P, R \cap \bar{M}, \bar{O})\}.$$

is equivalent to problem (1) provided that $S(P,R,O) \neq \emptyset$.

Now, if $S(P,R,O) = \emptyset$, then the set covering problem consisting of set P and a nonempty proper subset of P with an arbitrary weight is equivalent to the PNS problem under consideration, since both problems have no feasible solution. On the other hand, $S(P,R,O) = \emptyset$ can be decided in polynomial time by using the algorithm presented in [9] for generating the maximal structure of (P,R,O).

If $S(P,R,O) \neq \emptyset$, then instead of (1) we can consider (2) where the process graph of the problem is its maximal structure $(\bar{M},\bar{O})$. In this case, a conjuctive normal form (CNF) exists for describing the feasible solutions. Description of the feasible solutions by (CNF) was originally initiated in [4] and this description is also used in [7]. Here, taking the maximal structure into account, a simplest and more precise description is given. The basis of this approach is the observation that a faesible solution is determined by o uniquely. This observation makes possible to the reformulation of properties ($S1$) through ($S4$).

To do this, let $\bar{O} = \{(\alpha_1,\beta_1),\ldots,(\alpha_l,\beta_l)\}$, and $J = \{1,\ldots,l\}$. Then, for any subgraph (m,o) of $(\bar{M},\bar{O})$, an l-vector of logical values y_i, $i \in J$, can be associated with such that y_i is true if and only if $(\alpha_i,\beta_i) \in o$. It is easy to see that this is a one-to-one mapping between the subgraphs of $(\bar{M},\bar{O})$ fulfilling ($S4$) and the l-vectors of logical values. For logical l-vector $\mathbf{y}$, subgraph (m,o) associated with $\mathbf{y}$ is determined by $m = \bigcup_{i \in T(\mathbf{y})} \alpha_i \cup \beta_i$ and $o = \{(\alpha_i,\beta_i) : i \in T(\mathbf{y})\}$ where

$T(\mathbf{y}) = \{i : i \in J \ \& \ y_i \text{ is true}\}$. Obviously, for an arbitrary logical l-vector, $\mathbf{y}$, the subgraph associated with $\mathbf{y}$ is not necessarily a feasible solution. Procedure below provides such a CNF, Φ, that a logical l-vector, $\mathbf{y}$, satisfies Φ if and only if the subgraph associated with $\mathbf{y}$ is a feasible solution.

Algorithm for CNF Generation

- *Step 1.* Set $\Phi_0 = \bigwedge_{X \in P} \bigvee_{\substack{i \in J \\ X \in \beta_i}} y_i$.
- *Step 2.* Let $\Phi_1 = \bigwedge_{\substack{i \in J \\ X \in \alpha_i \setminus R}} (\neg y_i \vee \bigvee_{\substack{h \in J \\ X \in \beta_h}} y_h)$.
- *Step 3.* Set $\Phi_2 = \bigwedge_{\substack{i \in J \\ P \cap \beta_i = \emptyset}} (\neg y_i \vee \bigvee_{\substack{h \in J \\ \beta_i \cap \alpha_h \neq \emptyset}} y_h)$.
- *Step 4.* Let $\Phi = \Phi_0 \wedge \Phi_1 \wedge \Phi_2$.

Now, we show that an l-vector $\bar{\mathbf{y}}$ of logical values satisfies Φ if and only if process graph associated with $\bar{\mathbf{y}}$ is a feasible solution. To do this, let (m, o) be an arbitrary feasible solution in $(\bar{M}, \bar{O})$, and let us denote by $\bar{\mathbf{y}}$ the l-vector of logical values associated with (m, o). Then, $(S1)$ implies $\Phi_0(\bar{\mathbf{y}}) = \uparrow$ and $(S2)$ implies $\Phi_1(\bar{\mathbf{y}}) = \uparrow$. Regarding Φ_2, let $i \in J$ and $P \cap \beta_i = \emptyset$. Then, $\neg y_i \bigvee_{\substack{h \in J \\ \beta_i \cap \alpha_h \neq \emptyset}} y_h$ is a member of Φ_2. If $\bar{y}_i = \downarrow$, then the considered disjunction is true. If $\bar{y}_i = \uparrow$, then $u_i \in o$, but in this case, $(S3)$ implies that there is an operating unit $u_h \in o$ such that $\beta_i \cap \alpha_h \neq \emptyset$. This yields that $\bar{y}_h = \uparrow$ for some $h \in J$, and thus, the considered member is also true. Idea presented above is valid for every member of Φ_2, and hence, $\Phi_2(\bar{\mathbf{y}}) = \uparrow$. Consequently, $\bar{\mathbf{y}}$ satisfies Φ.

Conversely, let us suppose that the logical vector $\bar{\mathbf{y}}$ satisfies Φ. Let (m, o) denote the subgaph of $(\bar{M}, \bar{O})$ associated with $\bar{\mathbf{y}}$. We prove that (m, o) is a feasible solution, *i.e.*, properties $(S1)$ through $(S4)$ hold for (m, o).

Property $(S4)$ follows directly from the definition of (m, o).

Since $\Phi_0(\bar{\mathbf{y}}) = \uparrow$, there exists an operating unit in o which produces X directly, for every $X \in P$. Consequently, $(S1)$ is valid as well.

To prove $(S2)$, let $X \in m$ and $X \in R$. Since $o \subseteq \bar{O}$ and there is no operating unit in $\bar{O}$ producing raw material, there is no (Y, X) arc in (m, o). Now, let us suppose that $X \in m$ and there is no (Y, X) arc in (m, o). Then, we show that $X \in R$. Contrary, let us assume that $X \notin R$. Since $X \in m$, there exists an operating unit, $u_i = (\alpha_i, \beta_i) \in o$ such that $X \in \alpha_i$ which implies $\bar{y}_i = \uparrow$. On the other hand, since $(\bar{M}, \bar{O})$ is a feasible solution, there are operation units $u_h \in \bar{O}$ such that $X \in \beta_h$, and therefore, Φ_1 contains a member of the form $\neg y_i \bigvee_{\substack{h \in J \\ X \in \beta_h}} y_h$. Since $\Phi_1(\bar{\mathbf{y}}) = \uparrow$, this member is also true on $\bar{\mathbf{y}}$, and hence, there is an h_0 such that

$u_{h_0} \in o$ and $X \in \beta_{h_0}$. This yields that there exists (Y, X) arc in (m, o) which is a contradiction. Therefore, $X \in R$, and thus, $(S2)$ is valid.

For proving $(S3)$, let us suppose that $u_i \in o$. If $\beta_i \cap P \neq \emptyset$, then we are ready, there is a path from u_i into P. In the opposite case, using again that $(\bar{M}, \bar{O})$ is a feasible solution, there is a path from u_i into P in $(\bar{M}, \bar{O})$, *i.e.*, there is an operating unit $u_h \in \bar{O}$ such that $\beta_i \cap \alpha_h \neq \emptyset$. Then, Φ_2 contains a member of form $\neg y_i \bigvee\limits_{\substack{h \in J \\ \beta_i \cap \alpha_h \neq \emptyset}} y_h$, and in a similar way as above, we obtain that $u_h \in o$ for some h. Repeating this procedure, in a finite number of steps, we obtain a path from u_i into P in (m, o), and therefore, $(S3)$ is valid.

By the strong relationship between the PNS problem and the corresponding CNF given above, the following problem is equivalent to problem (2).

$$(3) \qquad \min\{ \sum_{j \in T(\mathrm{y})} w_j : \mathrm{y} \text{ fullfils } \Phi\}$$

where $w_j = w(u_j)$, $j = 1, \ldots, l$.

For every $i \in J$, introducing two 0-1 variables, z_i^+ and z_i^-, such that $z_i^+ = 1$ if and only if y_i is true, and $z_i^- = 1 - z_i^+$, problem (3) can be transcribed into an equivalent binary programming problem by introducing some appropriate new constraints. It is easy to see that this binary problem is a set covering/partitioning problem. On the other hand, using the well-known trick of converting set partitioning constraints into set covering ones (*cf. e.g.* [10]), we obtain an equivalent set covering problem. The corresponding binary and set covering problems can be found in [7].

Summarizing, we obtained that the solution of a PNS problem can be traced back to the solution of a suitable set covering problem. Combining this statement by the observation that any set covering problem can be given as a special PNS problem, we obtain the following statement.

Proposition 1. *The PNS problem is equivalent to the set covering problem.*

Equivalence proved above enables the sophisticated techniques developed for solving set covering problems (see, *e.g.*, [2, 5, 10] and the references therein) also to be applied for solving PNS problems. Regarding this solution technique, it is an open problem whether a more economical transformation of the PNS problem into a set covering problem exists.

It is worth noting that Proposition 1, by the well-known fact that the set covering problem is NP-complete (see [1] and [11]), implies immediately the following observation.

Corollary 1. *The PNS problem is NP-complete.*

References

[1] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading Mass, 1974.

[2] E. Balas, M. C. Carrera, A Dynamic Subgradient- Based Branch-and-Bound Procedure for Set Covering, *Operations Research* 44 (1996), 875-890.

[3] Z. Blázsik, B. Imreh, A Note on Connection between PNS and Set Covering Problems, *Acta Cybernetica* **12** (1996), 309-312.

[4] M. H. Brendel, F. Friedler and L.T. Fan, Combinatorial Foundation for Logical Formulation in Total Flowsheet Synthesis, *Computers chem. Engng.*, submitted for publication.

[5] M. L. Fisher, P. Keida, Optimal Solution of Set Covering/Partitioning Problems Using Dual Heuristics, *Management Science* **36** (1990), 674-688.

[6] F. Friedler, L. T. Fan, B. Imreh, Process Network Synthesis: Problem Definition, *Networks* **28** (1998), 119-124.

[7] J. Fülöp, B. Imreh, F. Friedler, On the reformulation of some classes of PNS problems as set covering problems, *Acta Cybernetica* **13** (1998), 329-337.

[8] F. Friedler, K. Tarján, Y. W. Huang, and L. T. Fan, Graph-Theoretic Approach to Process Synthesis: Axioms and Theorems, *Chem. Eng. Sci.* **47**(8) (1992), 1973-1988.

[9] F. Friedler, K. Tarján, Y. W. Huang, and L. T. Fan, Graph-Theoretic Approach to Process Synthesis: Polynomial Algorithm for Maximal Structure Generation, *Computers chem. Engng.* **17**(9) (1993), 929-942.

[10] R. S. Garfinkel, G. L. Nemhauser, *Integer Programming*, Wiley, New York, 1972.

[11] R. M. Karp, Reducibility among Combinatorial Problems, in Complexity of Computer Computations, R. E. Miller and T. W. Thatcher, eds., Plenum Press, New York, 1972.

Received October, 1999

Note on the Work Function Algorithm

Béla Csaba *†‡

Abstract

We prove that the work function algorithm is $(n-1)$-competitive for the k-server problem, where n is the number of points in the metric space. This gives improved upper bounds when $k+3 \leq n \leq 2k-1$; in particular, it shows that the work function algorithm is optimal for $k = n-1$. Recently this result was proved independently by Koutsoupias in [K].

1 Introduction

We give a short introduction to the deterministic k-server problem ([ST], [MMS]). There is a metric space $\mathcal{M}$ with a distance function $d(.,.)$ on it. Let us denote $|\mathcal{M}|$ by n. There are k $(1 < k < n)$ mobile servers initially residing on the pointset I, no two on the same point. Repeatedly requests are generated by an off-line adversary, and we have to satisfy them in an on-line fashion. Satisfying a request is putting a server on the requested point of the metric space. When moving a server, a cost occurs which is the distance of the previous and the present position of the server which is moved. While the adversary has the advantage that it satisfies the request sequence at the end thus it can satisfy them optimally paying the least, an on-line algorithm pays after every request. In competitivity analysis we compare the on-line cost with the optimal (off-line) cost, we are looking for an algorithm with the best competitive ratio. In [MMS] it is proved that no on-line algorithm can have competitive ratio less than k. They also gave optimal on-line algorithm for two special cases: when $k = 2$ and when $k = n-1$. They conjectured that in every case there is an optimal, k- competitive on-line algorithm. There are other cases with optimal on-line algorithms: when the metric space is tree-like ([CL]), when every distance is the same (paging problem, [ST]), or for the weighted cache problem ([CKPV]). So far the best general on-line algorithm is given by Koutsoupias and Papadimitriou ([KP1]). They proved that the work function algorithm is $(2k-1)$-competitive. In another paper ([KP2]) they also showed that the work function algorithm is k-competitive if $k+2 = n$. In this paper we give a proof that their

*Department of Computer Science; Rutgers, The State University of NJ; 110 Frelinghuysen Road; Piscataway, NJ 08854-8019 USA. Email: bcsaba@paul.rutgers.edu

†Research partially supported by a DIMACS fellowship

‡Research partially supported by OTKA T30074.

algorithm is $(n-1)$–competitive as well, hence for $k+3 \le n \le 2k-1$ this gives the best known upper bounds, and for $k = n-1$ it proves that the work function algorithm is optimal. In [K] Koutsoupias proved this result by considering the so called weak adversaries for the k–server problem. Our paper gives a simple alternative proof.

The outline of this note is as follows: in the next chapter we present a brief description of the work function algorithm, and state some relevant properties of it, without proofs. Then in the third chapter we prove our result.

2 Work Function Algorithm

Let $\mathcal{M}$, I and the request sequence ϱ is given. Then the work function w_ϱ maps configurations to nonnegative real numbers: $w_\varrho(X)$ is the optimal cost of servicing ϱ starting at I and ending at X. For a work function w_ϱ the resulting work function after the request r is $w_{\varrho r}$. The following proves to be very useful:

Fact 2.1 *For all X $w_{\varrho r}(X) \ge w_\varrho(X)$.*

Let us now define the algorithm itself.

Definition 2.2 (Work Function Algorithm) *Let ϱ be a request sequence and A be the configuration of the servers after satisfying ϱ. The work function algorithm services the new request at r by moving one of the servers to the configuration $B = A - a + r$ (i.e.,the server moves from a to r), where $w_{\varrho r}(B) + d(a,r)$ is minimal.*

The so called extended cost of a move, $\max_X\{w_{\varrho r}(X) - w_\varrho(X)\}$ in some way encapsulates the optimal and the on-line costs.

Fact 2.3 *If the sum of the extended costs is bounded above by $c+1$ times the optimal cost plus a constant, then the work function algorithm is c–competitive.*

The notion of minimizer configurations is also an important one.

Definition 2.4 *A configuration A is called a minimizer with respect to the point a with respect to w_ϱ if*

$$w_\varrho(A) - \sum_{x\in A} d(a,x) = \min_X\{w_\varrho(X) - \sum_{x\in X} d(a,x)\}. \tag{1}$$

We finish decribing the relevant definitions and facts by the following lemma, the *Duality Lemma.*

Lemma 2.5 (Duality Lemma) *Let w_ϱ be a work function, and let $w_{\varrho r}$ be the resulting work function after request r. Then any minimizer A of r with respect to w_ϱ is also a minimizer of r with respect to $w_{\varrho r}$, and the extended cost of servicing the request r is $w_{\varrho r}(A) - w_\varrho(A)$.*

3 Proof of the $(n-1)$-competitivity

Let ϱ be the request sequence, and let the next request be r. Denote A_x the minimizer configuration of x with respect to w_ϱ, and B_x the minimizer configuration of x with respect to $w_{\varrho r}$. Let us define a potential function P_ϱ:

$$P_\varrho = \sum_x \left[w_\varrho(A_x) - \sum_{a \in A_x} d(x,a) \right]. \tag{2}$$

Analoguosly, we can define $P_{\varrho r}$. Denoting the empty request sequence by $\emptyset$ one may observe, that $P_\emptyset$, the initial potential function is a constant. In the sequel by $opt(\varrho)$ we will denote the optimal cost of satisfaction of the request sequence ϱ.

Lemma 3.1 *(1) $P_{\varrho r} - P_\varrho \geq$ extended cost of the move from ϱ to ϱr and (2) $P_\varrho - n \cdot opt(\varrho) \leq$ constant.*

Proof. For proving (1) observe that A_r is a minimizer of r with respect to w_ϱ and $w_{\varrho r}$. Thus, $w_{\varrho r}(A_r) - \sum_{a \in A_r} d(r,a)$ is also a minimal expression, and $w_{\varrho r}(A_r) - w_\varrho(A_r)=$ extended cost. Let now v be another point of $\mathcal{M}$, then $w_\varrho(A_v) - \sum_{a \in A_v} d(v,a) \leq w_\varrho(B_v) - \sum_{b \in B_v} d(v,b) \leq w_{\varrho r}(B_v) - \sum_{b \in B_v} d(v,b)$. This proves (1). For proving the second part of the statement, observe that P_ϱ is the sum of optimal costs plus a constant. □

Theorem 3.2 *The work function algorithm is $(n-1)$-competitive.*

Proof. Summing up the potential function values over all requests we can see from Lemma 3.1 that this sum is at least the sum of the extended costs plus a constant. By Fact 2.3 we can conclude the statement of the theorem. □

References

[CKPV] Chrobak, M., Karloff, H., Payne, T. and Vishwanathan, S., *New results on Server Problems*, SIAM Journal on Disc. Math.,4:172-181, 1991

[CL] Chrobak, M. and Larmore, L., *An optimal online algorithm for k servers on trees*, SIAM Journal on Computing, 20:144-148, 1991

[K] Koutsoupias, E., *Weak adversaries for the k-server problem*, FOCS 99, pp. 444–449

[KP1] Koutsoupias, E. and Papadimitrou, C., *On the k-Server Conjecture*, STOC 94, pp. 507-511

[KP2] Koutsoupias, E. and Papadimitrou, C., *The 2-evader problem*, Information Processing Letters, 57(5):249-252, 1996

[MMS] Manasse, M. S., McGeoch L. A. and Sleator, D. D., *Competitive Algorithms for Server Problems*, Journal of Algorithms 11 (1990), pp. 208-230

[ST] Sleator, D. D., Tarjan, R. E., *Amortized Efficiency of List Update and Paging Rules*, Comm. of the ACM, February 1985, pp. 202-208

Received October, 1999

Acta Cybernetica 14 (2000) 507-522.

Modular Reinforcement Learning: A Case Study in a Robot Domain

Zsolt Kalmár [*] Csaba Szepesvári [†] András Lőrincz [‡]

Abstract

The behaviour of reinforcement learning (RL) algorithms is best understood in completely observable, finite state- and action-space, discrete-time controlled Markov-chains. Robot-learning domains, on the other hand, are inherently infinite both in time and space, and moreover they are only partially observable. In this article we suggest a systematic design method whose motivation comes from the desire to transform the task-to-be-solved into a finite-state, discrete-time, "approximately" Markovian task, which is completely observable, too. The key idea is to break up the problem into subtasks and design controllers for each of the subtasks. Then operating conditions are attached to the controllers (together the controllers and their operating conditions which are called modules) and possible additional features are designed to facilitate observability. A new discrete time-counter is introduced at the "module-level" that clicks only when a change in the value of one of the features is observed. The approach was tried out on a real-life robot. Several RL algorithms were compared and it was found that a model-based approach worked best. The learnt switching strategy performed equally well as a handcrafted version. Moreover, the learnt strategy seemed to exploit certain properties of the environment which could not have been seen in advance, which predicted the promising possibility that a learnt controller might overperform a handcrafted switching strategy in the future.

1 Introduction

Reinforcement learning (RL) is the process of learning the coordination of concurrent behaviours and their timing. A few years ago Markovian Decision Problems (MDPs) were proposed as the model for the analysis of RL [17] and since then a mathematically well-founded theory has been constructed for a large class

[*] Department of Informatics JATE, Szeged, Aradi vértanúk tere 1, Hungary, H-6720; Present adress: Mindmaker Ltd. Budapest 1121, Konkoly Thege M. út 29-33; email: kalmar@mindmaker.hu

[†] Research Group on Art. Int., JATE, Szeged, Aradi vértanúk tere 1, Hungary, H-6720; Present adress: Mindmaker Ltd. Budapest 1121, Konkoly Thege M. út 29-33; email: szepes@mindmaker.hu

[‡] Eötvös Loránd University, Budapest, Múzeum krt. 8, Hungary H-1068; email: lorincz@iserv.iki.kfki.hu

of RL algorithms. These algorithms are based on modifications of the two basic dynamic-programming algorithms used to solve MDPs, namely the value- and policy-iteration algorithms [25, 5, 10, 23, 18]. The RL algorithms learn via experience, gradually building an estimate of the optimal value-function, which is known to encompass all the knowledge needed to behave in an optimal way according to a fixed criterion, usually the expected total discounted-cost criterion. The basic limitations of all of the early theoretical results of these algorithms was that they assumed finite state- and action-spaces, and discrete-time models in which the state information too was assumed to be available for measurement. In a real-life problem however, the state- and action-spaces are infinite, usually non-discrete, time is continuous and the system's state is not measurable (i.e., with the latter property the process is only partially observable as opposed to being completely observable). Recognizing the serious drawbacks of the simple theoretical case, researchers have begun looking at the more interesting yet theoretically more difficult cases (see e.g. [11, 16]). To date, however, no complete and theoretically sound solution has been found to deal with such involved problems. In fact the above-mentioned learning problem is indeed, intractable owing to partial-observability. This result follows from a theorem of Littman [9].

In this paper an attempt is made to show that RL can be applied to learn real-life tasks when *a priori* knowledge is combined in some suitable way. The key to our proposed method lies in the use of high-level modules along with a specification of the operating conditions for the modules and other "features", to transform the task into a finite-state and action, completely-observable task. Of course, the design of the modules and features requires a fair amount of *a priori* knowledge, but this knowledge is usually readily available. In addition to this, there may be several possible ways of breaking up the task into smaller subtasks but it may be far from trivial to identify the best decomposition scheme. If all the possible decompositions are simultaneously available then RL can be used to find the best combination. Here we propose design principles and theoretical tools for the analysis of learning and demonstrate the success of this approach via *real-life* examples. A detailed comparison of several RL methods, such as Adaptive Dynamic Programming (ADP), Adaptive Real-Time Dynamic Programming (ARTDP) and Q-learning is provided, having been combined with different exploration strategies.

The article is organized in the following way. In Section 2 we introduce our proposed method and discuss the motivations behind it. The notion of "approximately" stationary MDPs is also introduced as a useful tool for the analysis of "module-level" learning. Then, in Section 3 the outcome of certain experiments using a mobile robot are presented. The relationship of our work to that of others is contrasted in Section 4. Finally our conclusions and possible directions for further research are given in Section 5. Some details were left out from this article, but these can be found in [8].

2 Module-based Reinforcement Learning

First of all, we will briefly run through Markovian Decision Problems (MDPs), a value-function approximation-based RL algorithm to learn solutions for MDPs and their associated theory. Next, the concept of recursive-features and time discretization based on these features are elaborated upon. This is then followed by a sensible definition and principles of module-design together with a brief explanation of why the modular approach can prove successful in practice.

2.1 Markovian Decision Problems

RL is the process by which an agent improves its behaviour from observing its own interactions with the environment. One particularly well-studied RL scenario is that of a single agent minimizing the expected-discounted total cost in a discrete-time finite-state, finite-action environment, when the theory of MDPs can be used as the underlying mathematical model. A finite MDP is defined by the 4-tuple (S, A, p, c), where S is a finite set of states, A is a finite set of actions, $p : S \times A \times S \to [0, 1]$ is a transition probability function satisfying $\sum_{s' \in S} p(s, a, s') = 1$ for all $(s, a) \in S \times A$ pairs and $c : S \times A \to \Re$ is the so-called immediate cost-function. The ultimate target of learning is to identify an optimal policy. A policy is some function that tells the agent which set of actions should be chosen under which circumstances. A policy π is optimal under the *expected discounted total cost criterion* if, with respect to the space of all possible policies, π results in a minimum expected discounted total cost for all states. The optimal policy can be found by identifying the optimal value-function, defined recursively by

$$v^*(s) = \min_{a \in U(s)} \left(c(s, a) + \gamma \sum_{s'} p(s, a, s') \, v^*(s') \right)$$

for all states $s \in S$, where $c(s, a)$ is the immediate cost for taking action a from state s, γ is the discount factor, and $p(s, a, s')$ is the probability that state s' is reached from state s when action a is chosen. $U(s)$ is the set of admissible actions in state s. The policy which for each state selects the action that minimizes the right-hand-side of the above fixed-point equation constitutes an optimal policy. This yields the result that to identify an optimal policy it is sufficient just to find the optimal value-function v^*. The above simultaneous non-linear equations (non-linear because of the presence of the minimization operator), also known as the *Bellman equations* [3], can be solved by various dynamic programming methods such as the value- or policy-iteration methods [15].

RL algorithms are generalizations of the DP methods to the case when the transition probabilities and immediate costs are unknown. The class of RL algorithms of interest here can be viewed as variants of the value-iteration method: these algorithms gradually improve an estimate of the optimal value-function via learning from interactions with the environment. There are two possible ways to learn the optimal value-function. One is to estimate the model (i.e., the transition probabilities and immediate costs) while the other is to estimate the optimal action-values

```
Initialization: Let $t = 0$, and initialize the utilized model ($M_0$) and the Q-function ($Q_0$)
Repeat Forever
1. Observe the next state $s_{t+1}$ and reinforcement signal $c_t$ .
2. Incorporate the new experience $(s_t, a_t, s_{t+1}, c_t)$ into the model and into the estimate of the optimal Q-function: $(M_{t+1}, Q_{t+1}) = F_t(M_t, Q_t, (s_t, a_t, s_{t+1}, c_t))$.
3. Choose the next action to be executed based on $(M_{t+1}, Q_{t+1})$: $a_{t+1} = S_t(M_{t+1}, Q_{t+1}, s_{t+1})$ and execute the selected action.
4. $t := t + 1$.
```

Table 1: **The structure of value-function-approximation based RL algorithms.**

directly. The optimal action-value of an action a given a state s is defined as the total expected discounted cost of executing the action from the given state and proceeding in an optimal fashion afterwards:

$$Q^*(s,a) = c(s,a) + \gamma \sum_{s'} p(s,a,s') v^*(s'). \tag{1}$$

The general structure of value-function-approximation based RL algorithms is given in Table 1.

In the RL algorithms various models are utilized along with an update rule F_t and action-selection rule S_t.

In the case of the Adaptive Real-Time Dynamic Programming (ARTDP) algorithm the model consists (M_t) of the estimates of the transition probabilities and costs, the update-rule F_t being implemented, e.g., as an averaging process. Instead of the optimal Q-function, the optimal value-function is estimated and stored to spare storage space, and the Q-values are then computed by replacing the true transition probabilities, costs and the optimal value-function in Equation 1 by their estimates. An update of the estimate for the optimal value-function is implemented by an asynchronous dynamic programming algorithm using an inner-loop in Step 2 of the algorithm.

Another popular RL algorithm is Q-learning, which does not employ a model but instead the Q-values are updated directly according to the iteration procedure [25]

$$Q_{t+1}(s_t,a_t) = (1-\alpha_t(s_t,a_t))Q_t(s_t,a_t) + \\ \alpha_t(s_t,a_t)(c_t + \gamma \min_a Q_t(s_{t+1},a)),$$

where $\alpha_t(s_t,a_t) \geq 0$, and satisfies the usual Robbins-Monro type of conditions. For example, one might set $\alpha_t(s,a) = \frac{1}{n_t(s,a)+1}$ where $n_t(s,a)$ is the number of time the state-action pair (s,a) was visited before time t. But often in practice $\alpha_t(s,a) =$ const is employed which while yielding increased adaptivities no longer ensures convergence.

Both algorithms mentioned previously are guaranteed to converge to the optimal value-/Q-function if each state-action pair is updated infinitely often. The action selection procedure S_t should be carefully chosen so that it fits the dynamics of the controlled process in a way that the condition is met. For example, the execution of random actions meets this "sufficient-exploration" condition when the MDP is ergodic. However, if on-line performance is important then more sophisticated exploration is needed which, in addition to ensuring sufficient exploratory behaviour, exploits accumulated knowledge. For more details the interested reader is referred to [8].

2.2 The modular architecture

In case of a real-life robot-learning task the dynamics cannot be formulated exactly as a *finite* MDP, nor is the state information available for measurement. This latter restriction is modelled by Partially-Observable MDPs (POMDPs) where (in the simplest case) one extends an MDP with an *observation function* h which maps the set of states S into a set X, called the observation set (which is usually non-countable, just like S). The defining assumption of a POMDP is that the full state s can be observed only through the observation function, i.e., only $h(s)$ is available as input and this information alone is usually insufficient for efficient control since h is usually a non-injection (i.e., h may map different states to the same observations).

The dynamics of the controlled system is defined by $P(s_{t+1} = s'|s_t = s, a_t = a) = p(s, a, s')$ and $x_t = h(s_t)$. The first part of the controller is the feature extraction part. The designer needs to design a feature space F together with a feature-extraction function R mapping $X \times A \times F^k$ to F, where k is another design parameter. The feature extraction function R transforms observation-action pairs into features in a recursive way: the feature f_t at time t is defined by $f_t = R(x_t, a_t, f_{t-1}, \ldots, f_{t-k})$, $t \geq 0$, where $f_{t-1}, \ldots, f_{t-k}$ are other design parameters. The rest of the system is composed of a finite number of controllers, $M^{(1)}, \ldots, M^{(n)}$, where $M^{(i)} = (Z^{(i)}, \delta^{(i)})$, $\delta^{(i)} : Z^{(i)} \times F \to \{0,1\} \times A \times Z^{(i)}$. The ith controller stores an internal "state" $z_t^{(i)}$ which develops in time according to $z_{t+1}^{(i)} = \delta^{(i)}(z_t^{(i)}, f_t)$. $\delta_1^{(i)}(z_t^{(i)}, f_t)$ determines whether the ith controller is available for control in time step t: If $\delta_1^{(i)}(z_t^{(i)}, f_t) = 1$ then the controller is available, otherwise it is not available. Each controller should be thought of as a "local" controller that is able to carry out a particular subtask of the whole problem. The control problem then is to design a switching strategy that switches between the appropriate modules (local controllers) such that the controlled system will eventually show a behavior consistent with the ultimate goal of control. This is formulated as follows: we further restrict the switching function to be a memoryless mapping $S : F' \times \{0,1\}^n \to \{1, 2, \ldots, n\}$. Here F' is another design set that comes together with a mapping $\pi : F \to F'$. π maps computed features to "monitored" features. The purpose of π is to bring in some more flexibility in the design procedure. The role of π will be clear soon once the control equations are given. In order to arrive at the definition of control, let use first define the sequence $\tau_0, \tau_1, \ldots$ as follows. Let $\tau_0 = 0$ and let τ_1, $\tau_2, \ldots$ be defined by $\tau_{s+1} = \min\{\, t > \tau_s : \delta_1^{(i)}(z_t^{(i)}, f_t) \neq \delta_1^{(i)}(z_{t-1}^{(i)}, f_{t-1})$ for some i or $\pi(f_t) \neq \pi(f_{t-1}) \,\}$. A switching function $S : F' \times \{0,1\}^n \to \{1, 2, \ldots, n\}$ is called admissible if from $S(f, c_1, \ldots, c_n) = i$ it follows that $c_i = 1$ (i.e. only controllers which are available for control are chosen by the switching function). Given an admissible switching function S the control works as follows: at any given time instant t there is only one module active. The index of this module is denoted by m_t and satisfies $m_{t+1} = m_t$ if $t \notin \{\tau_0, \dots, \tau_s, \dots\}$, otherwise $m_{t+1} = S(\pi(f_t), \delta_1^{(1)}(z_t^{(1)}, f_t), \dots, \delta_1^{(n)}(z_t^{(n)}, f_t))$. The control is then given by $a_t = \delta_2^{(m_t)}(z_t^{(m_t)}, f_t)$.

Assume a goal oriented task, i.e. a POMDP where the success of a controller is measured in terms of if the system state can be driven to a particular set of goal states. Then the goal of the design procedure is to set up the modules and the additional features in such a way that there exists an admissible switching function S that for any given history results in a closed-loop behaviour which fulfills the "goal" of control. It can be extremely hard to prove even the *existence* of such a valid switching function. One approach is to use a so-called *accessibility decision problem* which is a discrete graph with its node set being $F' \times \{0,1\}^n$ and the edges connect features which can be observed in succession. Then, standard *DP* techniques can be used to decide the existence of a proper switching function [8].

Of course, since the definitions of the modules and features depend on the designer, it is reasonable to assume that by clever design a satisfactory decomposition and controllers could be found even if only qualitative properties of the controlled object were known. RL could then be used for two purposes: either to find the best switching function assuming that at least two proper switching functions exist, or to decide empirically whether a valid switching controller exists at all. The first kind of application of RL arises as result of the desire to guarantee the existence of a proper switching function through the introduction of more modules and features than is minimally needed. But then, good switching which exploits the capabilities of all the available modules could well become complicated to find manually.

If the accessibility decision problem were extendible with transition-probabilities to turn it to an MDP [1], then RL could be rightly applied to find the best switching function. For example, if one uses a fixed (maybe stochastic) stationary switching policy and provided that the system dynamics can be formulated as an MDP then there is a theoretically well-founded way of introducing transition-probabilities (see [16]). Unfortunately, the resulting probabilities may well depend on the switching policy which can prevent the convergence of the RL algorithms. However, the following "stability" theorem shows that the difference of the cost of optimal policies corresponding to different transition probabilities is proportional to the extent the transition probabilities differ, so we may expect that a slight change in the transition probabilities does not result in completely different optimal switching policies and hence, as will be explained shortly after the theorem, we may expect RL to work properly, after all.

Theorem 2.1 *Assume that two MDPs differ only in their transition-probability matrices, and let these two matrices be denoted by p_1 and p_2. Let the corresponding*

[1]Note that as the original control problem is deterministic it is not immediate when the introduction of probabilities can be justified. One idea is to refer to the ergodicity of the control problem.

optimal cost-functions be v_1^* *and* v_2^*. *Then,*

$$||v_1^* - v_2^*|| \le \gamma \frac{nC||p_1 - p_2||}{(1-\gamma)^2},$$

where $C = ||c||$ *is the maximum of the immediate costs,* $||\cdot||$ *denotes the supremum-norm and* n *is the size of the state-space.*

Proof: Let T_i be the optimal-cost operator corresponding to the transition-probability matrix p_i, i.e.,

$$(T_i v)(s) = \min_{a \in U(x)} \left(c(s,a) + \gamma \sum_{s' \in X} p_i(s,a,s') v(s') \right),$$

$$v : S \to \Re,\ i = 1,2.$$

Proceeding with standard fixpoint and contraction arguments (see e.g. [19]) we get that $||v_1^* - v_2^*|| \le ||T_1 v_1^* - T_1 v_2^*|| + ||T_1 v_2^* - T_2 v_2^*||$ and since T_1 is a contraction with index γ, and the inequality $||T_1 v - T_2 v|| \le \gamma ||p_1 - p_2|| \sum_{y \in X} |v(y)|$ we obtain $\delta = ||v_1^* - v_2^*|| \le \gamma\delta + \gamma ||p_1 - p_2|| |X| C/(1-\gamma)$, where $||v_1^*|| \le C/(1-\gamma)$ has been employed [15]. Rearranging the inequality in terms of δ then yields Theorem 2.1. Q.E.D.

Motivated by the previous theorem we define ε-stationary MDPs as the quadruple (S, A, p, c), where S, A and c are as before but p, the transition probability matrix, may vary in time but with $||p_t - p^*|| \le \varepsilon$ holding for all $t > 0$. Our expectations are that although the transitions cannot be modelled with a fixed transition probability matrix (i.e., stationary MDP), they can be modelled by an ε-stationary one even if the switching functions are arbitrarily varied and we conjecture that RL methods would then result in oscillating estimates of the optimal value-function, but with the oscillation being asymptotically proportional to ε. Note that ε-stationarity was clearly observed in our experiments which we will describe now.

3 Experiments

The validity of the proposed method was checked with actual experiments carried out using a Khepera-robot. The robot, the experimental setup, general specifications of the modules and the results are all presented in this section.

3.1 The Robot and its Environment

The mobile robot employed in the experiments is shown in Figure 1.

It is a Khepera[2] robot equipped with eight IR-sensors, six in the front and two at the back, the IR-sensors measuring the proximity of objects in the range

[2]The Khepera was designed and built at Laboratory of Microcomputing, Swiss Federal Institute of Technology, Lausanne, Switzerland.

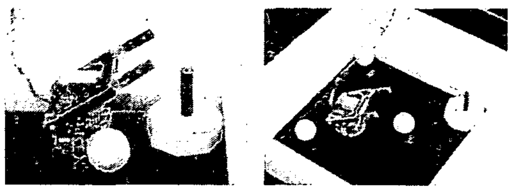


Figure 1: **The Khepera and the experimental environment.** The task was to grasp a ball and hit the stick with it.

0-5 cm. The robot has two wheels driven by two independent DC-motors and a gripper which has two degrees of freedom and is equipped with a resistivity sensor and an object-presence sensor. The vision turret is mounted on the top of the robot as shown. It is an image-sensor giving a linear-image of the horizontal view of the environment with a resolution of 64 pixels and 256 levels of grey. The horizontal viewing-angle is limited to about 36 degrees. This sensor is designed to detect objects in front of the robot situated at a distance spanning 5 to 50 cm. The image sensor has no tilt-angle, so the robot observes only those things whose height exceeds 5 cm.

The learning task was defined as follows: find a ball in an arena, bring it to one of the corners marked by a stick and hit the stick with the ball. The robot's environment is shown in Figure 1. The size of the arena was 50 cm x 50 cm with a black coloured floor and white coloured walls. The stick was black and 7 cm long, while three white-coloured balls with diameter 3.5 cm were scattered about in the arena. The task can be argued to have been biologically inspired because it can be considered as the abstraction of certain foraging tasks or a "basketball game". The environment is highly chaotic because the balls move in an unpredictable manner and so the outcome of certain actions is not completely predictable, e.g., a grasped ball may easily slip out from the gripper.

3.2 The Modules

3.2.1 Subtask decomposition

Firstly, the task was decomposed into subtasks. The following subtasks were naturally: (T1) to find a ball, (T2) grasp it, (T3) bring it to the stick, and (T4) hit the stick with the grasped ball. Subtask (T3) was further broken into two subtasks, that of (T3.1) 'safe wandering' and (T3.2) 'go to the stick', since the robot cannot see the stick from every position and direction. Similarly, because of the robot's limited sensing capabilities, subtask (T1) was replaced by safe-wandering and subtask (T2) was refined to 'when an object nearby is sensed examine it and grasp it if it is a ball'. Notice that subtask 'safe wandering' is used for two purposes (to find a ball or the stick). The operating conditions of the corresponding controllers arose naturally as (T2) – an object should be nearby, (T3.2) – the stick should be detected, (T4) – the stick should be in front of the robot, and (T1,T3.1) – no condition. Since the behaviour of the robot must differ before and after locating a ball, an additional feature indicating when a ball was held was supplied. As the robot's gripper is equipped with an 'object-presence' sensor the 'the ball is held" feature was easy to implement. If there had not been such a sensor then this feature still could have been implemented as a switching-feature: the value of the feature would be `'on'` if the robot used the grasping behaviour, and hence not the hitting behaviour. An 'unstuck' subtask and corresponding controller were also included since the robot sometimes got stuck. Of course yet another feature is included for the detection of "goal-states". The corresponding feature indicates when the stick was hit by the ball. This feature's value is `'on'` iff the gripper is half-closed but the object presence sensor does not give a signal. Because of the implementation of the grasping module (the gripper was closed only after the grasping module was executed) this implementation of the "stick has been hit by the ball" feature was satisfactory for our purposes, although sometimes the ball slipped out from the gripper in which case the feature turned 'on' even though the robot did not actually reach the goal. Fortunately this situation did not happen too often and thus did not affect learning.

The resulting list of modules and features is shown in Table 2. The controllers work as intended, some fine details are discussed here (for more complete description see [8]). For example, the observation process was switched off until the controller of `Module 3` was working so as the complexity of the module-level decision problem is reduced. The dynamics of the controller associated with `Module 1` were based on the maximization of a function which depended on the proximity of objects and the speed of both motors[3]. If there were no obstacles near the robot this module made the robot go forward. This controller could thus serve as one for exploring the environment. `Module 2` was applicable only if the stick was in the viewing-angle of the robot, which could be detected in an unambiguous way because the only black thing that could get into the view of the robot was the stick. The range of allowed behaviour associated with this module was implemented as a proportional controller which drove the robot in such a way that the angle difference between the direction of motion and line of sight to the stick was reduced. The behaviour associated with `Module 3` was applicable only if there was an object next to the robot, which was defined as a function of the immediate values of IR-sensors. The associated behaviour was the following: the robot turned to a direction which brought it to point directly at the object then the gripper was lowered. The "hit the stick" module (`Module 4`) lowers the gripper which under appropriate conditions result in that the ball jumps out of the gripper resulting in the goal state. `Module 5` was created to handle stuck situations. This module lets the robot go backward and is applicable if the robot has not been able to move the wheels into the desired position for a while. This condition is a typical time-window based feature.

Simple case-analysis shows that there is no switching controller that would reach the goal with complete certainty (in the worst-case, the robot could return accidentally to state "10000000" from any state when the goal feature was `'off'`),

[3]Modules are numbered by the identification number of their features.

FNo	'on'	Behaviour
1	always	explore while avoiding obstacles
2	if the stick is in the viewing angle	go to the stick
3	if an object is near	examine the object grasp it if it is a ball
4	if the stick is near	hit the stick
5	if the robot is stuck	go backward
6	if the ball is grasped	-
7	if the stick is hit with the ball	-

Table 2: **Description of the features and the modules.** 'FNo.' means 'Feature No.', in the column labelled by 'on' the conditions under which the respective feature's value is `'on'`are listed.

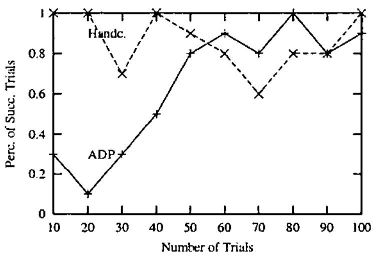
so that an almost-sure switching strategy should always exist. On the other hand, it is clear that a switching strategy which eventually attains the target does exist.

3.3 Details of learning

A dense cost-structure was applied: the cost of using each behaviour was one except when the goal was reached, whose cost was set to zero. Costs were discounted at a rate of $\gamma = 0.99$. Note that from time to time the robot by chance became stuck (the robot's 'stuck feature' was `'on'`), and the robot tried to execute a module which could not change the value of the feature-vector. This meant that the robot did not have a second option to try another module since by definition the robot could only make decisions if the feature-representation changed. As a result the robot could sometimes get stuck in a "perpetual" or so-called "jammed" state. To prevent this happening we built in an additional rule which was to stop and reinitialize the robot when it got stuck and could not unjam itself after 50 sensory measurements. A cost equivalent to the cost of never reaching the goal, i.e., a cost of $\frac{1}{1-\gamma}$ (= 100) was then communicated to the robot, which mimicked in effect that such actions virtually last forever.

Experiments were fully automated and organized in trials. Each trial run lasted until the robot reached the goal or the number of decisions exceeded 150 (a number that was determined experimentally), or until the robot became jammed. The 'stick was hit' event was registered by checking the state of the gripper (see also the description of `Feature 7`).

During learning the Boltzmann-exploration strategy was employed where the temperature was reduced by $T_{t+1} = 0.999 T_t$ uniformly for all states [2]. During the experiments the cumulative number of successful trials were measured and compared to the total number of trials done so far, together with the average number of decisions made in a trial.



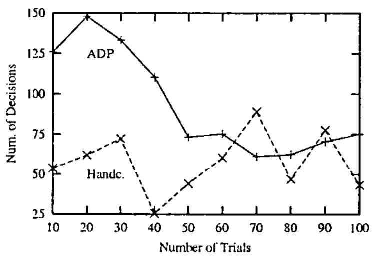


Figure 2: **Learning curves.** In the first graph the percentage of successful trials out of ten are shown as a function of the number of trials. In the second graph the number of decisions taken by the robot and averaged over ten trials are both shown, as well as a function of the number of learning trials.

3.4 Results

Two sets of experiments were conducted. The first set was performed to check the validity of the module based approach, while the second was carried out to compare different RL algorithms. In the first set the starting exploration parameter T_0 was set to 100 and the experiment lasted for 100 trials. These values were chosen in such a way that the robot could learn a good switching policy, the results of these experiments being shown in Figure 2.

One might conclude from the left subgraph which shows the percentage of task completions in different stages of learning that the robot could solve the task after 50 trials fairly well. Late fluctuations were attributable to unsuccessful ball searches: as the robot could not see the balls if they were far from it, the robot had to explore to find one and the exploration sometimes took more than 150 decisions, yielding trials which were categorized as being failures. The evaluation of behaviour-coordination is also observed in the second subgraph, which shows the number of decisions per trial as a function of time. The reason for later fluctuations is again due to a ticklish ball search. The performance of a handcrafted switching policy is shown on the graphs as well. As can be seen the differences between the respective performances of the handcrafted and learnt switching functions are eventually negligible. In order, to get a more precise evaluation of the differences the average number of steps to reach the goal were computed for both switchings over 300 trials, together with their standard deviations. The averages were 46.61 and 48.37 for the learnt and the handcrafted switching functions, respectively, with nearly equal std-s of 34.78 and 34.82, respectively.

Theoretically, the total number of states is $2^7 = 128$, but as learning concentrates on feature-configurations that really occur this number transpires to be just 25 here. It was observed that the learnt policy was always consistent with a set of handcrafted rules, but in certain cases the learnt rules are more refined than their handcrafted counterparts. For example, the robot learnt to exploit the fact that the arena was not completely level and as a result balls were biased towards the

Method	$T_0 = 100$	$T_0 = 50$	$T_0 = 25$	$T_0 = 0$
ADP	(61;19;6)	(52;20;4)	(45;12;5)	*(44:16:4)*
ARTDP	36	29	50	24
RTQL-1	53	69	47	66
RTQL-2	71	65	63	73
RTQL-3	(83;1;2)	(79;26:3)	(65;24;4)	*(61:14:2)*

Table 3: **Regret.** The table shows the number of unsuccessful trials among the first 100 trials. The entries with three number in them show cases when more than one experiment was run. In those entries the first number shows the average of the number of unsuccessful trials, the second is the standard deviation while the third is the number of experiments run.

stick and as a result if the robot did not hold a ball but could see the stick it moved towards the stick.

In the rest of the experiments we compared two versions of ARTDP and three versions of *real-time Q-learning* (RTQL). The two variants of ARTDP which we call ADP, and "ARTDP", corresponding to the cases when in the inner loop of ARTDP the optimal value function associated with the actual estimated model (transition probabilities and immediate cost) is computed and when only the estimate of the value of the actual state is updated. Note that due to the small number of states and module-based time discretization even ADP could be run in real-time. But variants of RTQL differ in the choice of the learning-rate's time-dependence. RTQL-1 refers to the choice of the so-called *search-then-converge* method, where $\alpha_k(s,a) = \frac{50}{100+n_k(s,a)}$, $n_k(s,a)$ being the number of times the event $(s,a) = (s_t, a_t)$ happened before time k plus one (the parameters 50 and 100 were determined experimentally as being the best choices). In the other two cases (the corresponding algorithms were denoted by RTQL-2 and RTQL-3 respectively) constant learning rates (0.1 and 0.25, respectively) were utilized.

The online performances of the algorithms were measured as the cumulative number of unsuccessful trials, i.e., the regret. The regret R_t at time t is the difference between the performance of an optimal agent (robot) and that of the learning agent accumulated up to trial t, i.e., it is the price of learning up to time t. A comparison of the different algorithms with different exploration ratios is given in Table 3. All algorithms were examined with all the four different exploration parameters since the same exploration rate may well result in different regrets for different algorithms, as was also confirmed in the experiments.

First note that in order to evaluate statistically the differences observed for different exploration strategies much more experiments would be needed but running these experiments would require an enormous amount of time (approximately 40 days) and have not been performed yet. Thus we performed the following procedure: Based on the first runs with every exploration-parameter and algorithm the algorithms that seemed to perform the best were selected (these were the ADP and the RTQL-3 algorithms) and some more experiments were carried out with these.

The results of these experiments (15 more for the ADP and 7 more for the RTQL-3) indicated that the difference between the performances of the RTQL-3 and ADP is significant at the level $p = 0.05$ (Student's t-test was applied for testing this).

We have also tested another exploration strategy which Thrun found the best among several undirected methods[4] [21]. These runs reinforced our previous findings that estimating a model (i.e., running ADP or ARTDP instead of Q-learning) could reduce the regret rate by as much as 40%.

4 Related Work

There are two main research-tracks that influenced our work. The first was the introduction of features in RL. Learning while using features were studied by Tsitsiklis and Van Roy to deal with large finite state spaces, and also to deal with infinite state spaces [22]. Issues of learning in partially observable environments have been discussed by Singh et al. [16].

The work of Connell and Mahadevan complements ours in that they set-up subtasks to be learned by RL and fixed the switching controller [13].

Asada et al. considered many aspects of mobile robot learning. They applied a vision-based state-estimation approach and defined "macro-actions" similar to our controllers [1]. In one of their papers they describe a goal-shooting problem in which a mobile robot shot a goal while avoiding another robot [24]. First the robot learned two behaviours separately: the "shot" and "avoid" behaviours. Then, the two behaviours were synthetised by a handcrafted rule and later this rule was refined via RL. The learnt action-values of the two behaviours were reused in the learning process while the combination of rules took place at the level of state variables.

Matarić considered a multi-robot learning task where each robot had the same set of behaviours and features [14]. Just as in our case, her goal was to learn a good switching function by RL. She considered the case when each of the robots learned separately and the ultimate goal was that learning should lead to a good collective behaviour, i.e., she concentrated mainly on the more involved multi-agent perspective of learning. In contrast to her work, we followed a more engineer-like approach when we suggested designing the modules based on well-articulated and simple principles and contrary to her findings it was discovered that RL can indeed work well at the modular level.

In the AI community there is an interesting approach to mobile robot control called Behaviour-Based Artificial Intelligence in which "competence" modules or behaviours have been proposed as the building blocks of "creatures" [12, 4]. The decision-making procedure is, on the other hand, usually quite different from ours.

The technique proposed here was also motivated by our earlier experiences with a value-estimation based algorithm given in the form of "activation spreading" [20]. In this work activation spread out along the edges of a dynamically varying graph,

[4]An exploration strategy is called undirected when the exploration does not depend on the number of visits to the state-action pairs.

where the nodes represented state transitions called triplets. Later the algorithm was extended so that useful subgoals could be found by learning [6, 7]. In the future we plan to extend the present module-based learning system with this kind of generalization capability. Such an extension may in turn allow the learning of a hierarchical organization of modules.

5 Summary and Conclusions

In this article module-based reinforcement learning was proposed to solve the coordination of multiple "behaviours" or controllers. The extended features served as the basis of time- and space discretization as well as the operating conditions of the modules. The construction principles of the modules were: decompose the problem into subtasks; for each subtask design controllers and the controllers' operating conditions; check if the problem could be solved by the controllers under the operating and observability conditions, add additional features or modules if necessary, set-up the reinforcement function and learn a switching function from experience.

The idea of our approach was that a partially observable decision problem could be usually transformed into a completely observable one if appropriate features (filters) and controllers were employed. Of course, some *a priori* knowledge of the task and robot is always required to find those features and controllers. It was argued that RL could work well even if the resulting problem was only an ϵ-stationary Markovian. The design principles were applied to a real-life robot learning problem and several RL-algorithms were compared in practice. We found that estimating the model and solving the optimality equation at each step (which could be done owing to the economic, feature-based time-discretization) yielded the best results. The robot learned the task after 700 decisions, which usually took less than 15 minutes in real-time. We conjecture that using a rough initial model good initial solutions could be computed off-line which could further decrease the time required to learn the optimal solution for the task.

The main difference between earlier works and our approach here is that we have established principles for the design modules and found that our subsequent design and simple RL worked spendidly. Plans for future research include extending the method via module learning and also the theoretical investigation of ϵ-stationary Markovian decision problems using the techniques developed in [10].

Acknowledgements

The authors would like to thank Zoltán Gábor for his efforts of building the experimental environment. This work was supported by the CSEM Switzerland, OTKA Grants No. F20132 and T017110, Soros Foundation No. 247/2/7011.

References

[1] M. Asada, S. Noda, S. Tawaratsumida, and K. Hosoda. Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Machine Learning*, 23:279–303, 1996.

[2] A. Barto, S. J. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 1(72):81–138, 1995.

[3] R. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, New Jersey, 1957.

[4] R. Brooks. Elephants don't play chess. In *Designing Autonomous Agents.* Bradford-MIT Press, 1991.

[5] T. Jaakkola, M. Jordan, and S. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6(6):1185–1201, November 1994.

[6] Z. Kalmár, C. Szepesvári, and A. Lőrincz. Generalization in an autonomous agent. In *Proc. of IEEE WCCI ICNN'94*, volume 3, pages 1815–1817, Orlando, Florida, June 1994. IEEE Inc.

[7] Z. Kalmár, C. Szepesvári, and A. Lőrincz. Generalized dynamic concept model as a route to construct adaptive autonomous agents. *Neural Network World*, 5:353–360, 1995.

[8] Z. Kalmár, C. Szepesvári, and A. Lőrincz. Module based reinforcement learning: Experiments with a real robot. *Machine Learning*, 31:55–85, 1998. joint special issue on "Learning Robots" with the J. of Autonomous Robots;.

[9] M. Littman. *Algorithms for Sequential Decision Making.* PhD thesis, Department of Computer Science, Brown University, February 1996. Also Technical Report CS-96-09.

[10] M. Littman and C. Szepesvári. A Generalized Reinforcement Learning Model: Convergence and applications. In *Int. Conf. on Machine Learning*, pages 310–318, 1996.

[11] M. L. Littman, A. Cassandra, and L. P. Kaelbling. Learning policies for partially observable environments: Scaling up. In A. Prieditis and S. Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 362–370, San Francisco, CA, 1995. Morgan Kaufmann.

[12] P. Maes. A bottom-up mechanism for behavior selection in an artificial creature. In J. Meyer and S. Wilson, editors, *Proc. of the First International Conference on Simulation of Adaptive Behavior.* MIT Press, 1991.

[13] S. Mahadevan and J. Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55:311–365, 1992.

[14] M. Matarić. Reinforcement learning in the multi-robot domain. *Autonomous Robots*, 4, 1997.

[15] S. Ross. *Applied Probability Models with Optimization Applications.* Holden Day, San Francisco, California, 1970.

[16] S. Singh, T. Jaakkola, and M. Jordan. Learning without state-estimation in partially observable Markovian decision processes. In *Proc. of the Eleventh Machine Learning Conference*, pages pp. 284–292, 1995.

[17] R. Sutton. *Temporal Credit Assignment in Reinforcement Learning.* PhD thesis, University of Massachusetts, Amherst, MA, 1984.

[18] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems*, 8, 1996.

[19] C. Szepesvári and M. Littman. A unified analysis of value-function-based reinforcement-learning algorithms. *Neural Computation*, 1997. submitted.

[20] C. Szepesvári and A. Lőrincz. Behavior of an adaptive self-organizing autonomous agent working with cues and competing concepts. *Adaptive Behavior*, 2(2):131–160, 1994.

[21] S. Thrun. *The role of exploration in learning control.* Van Nostrand Rheinhold, Florence KY, 1992.

[22] J. Tsitsiklis and B. Van Roy. An analysis of temporal difference learning with function approximation. Technical Report LIDS-P-2322, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, 1995.

[23] J. N. Tsitsiklis and B. Van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22:59–94, 1996.

[24] E. Uchibe, M. Asada, and K. Hosoda. Behavior coordination for a mobile robot using modular reinforcement learning. In *Proc. of IEEE/RSJ Int. Conf. on Intelligent Robot and Sytems*, pages 1329–1336, 1996.

[25] C. Watkins and P. Dayan. Q-learning. *Machine Learning*, 3(8):279–292, 1992.

Received May, 1999

CONTENTS

ISSN 0324—721 X

Felelős szerkesztő és kiadó: Csirik János
A kézirat a nyomdába érkezett: 2000. március