Acta Universitatis Sapientiae

Informatica

Volume 2, Number 1, 2010

Sapientia Hungarian University of Transylvania
Scientia Publishing House

Contents

Acta Univ. Sapientiae, Informatica, **2,** 1 (2010) 5–9

On k-hypertournament losing scores

Shariefuddin Pirzada
Department of Mathematics
University of Kashmir, India
King Fahd University of Petroleum and Minerals, Saudi Arabia
email: sdpirzada@yahoo.co.in

Guofei Zhou
Department of Mathematics
Nanjing University, Nanjing, China
email: gfzhou@nju.edu.cn

Abstract. We give a new and short proof of a theorem on k-hypertournament losing scores due to Zhou et al. [8].

1 Introduction

An edge of a graph is a pair of vertices and an edge of a hypergraph is a subset of the vertex set, consisting of at least two vertices. An edge in a hypergraph consisting of k vertices is called a k-edge, and a hypergraph all of whose edges are k-edges is called a k-hypergraph.

A k-hypertournament is a complete k-hypergraph with each k-edge endowed with an orientation, that is, a linear arrangement of the vertices contained in the hyperedge. In other words, given two non-negative integers n and k with $n \geq k > 1$, a k-hypertournament on n vertices is a pair (V, A), where V is a set of vertices with $|V| = n$ and A is a set of k-tuples of vertices, called arcs, such that any k-subset S of V, A contains exactly one of the $k!$ k-tuples whose entries belong to S. If $n < k$, $A = \phi$ and this type of hypertournament is called a null-hypertournament. Clearly, a 2-hypertournament is simply a tournament. Let $e = (v_1, v_2, \ldots, v_k)$ be an arc in a k-hypertournament H. Then $e(v_i, v_j)$ represents the arc obtained from e by interchanging v_i and v_j.

The following result due to Landau [5] characterises the score sequences in tournaments.

Computing Classification System 1998: G.2.2, F.2.m
Mathematics Subject Classification 2010: 05C20
Key words and phrases: tournament, hypertournament, losing score

Theorem 1 *A sequence of non-negative integers* $[s_1, s_2, \ldots, s_n]$ *in non-decreasing order is a score sequence of some tournament if and only if for* $1 \leq j \leq n$

$$\sum_{i=1}^{j} s_i \geq \binom{j}{2},$$

with equality when $j = n$.

Now, there exist several proofs of Landau's theorem and a survey of these can be found in Reid [6]. Brualdi and Shen [1] obtained inequalities on the scores in tournaments which are individually stronger than that of Landau, but collectively the two are equivalent. Although tournament theory has attracted many graph theorists and much work has been reported in various journals, the latest can be seen in Iványi [2].

Instead of scores of vertices in a tournament, Zhou et al. [8] considered scores and losing scores of vertices in a k-hypertournament, and derived a result analogous to Landau's theorem [5]. The score $s(v_i)$ or s_i of a vertex v_i is the number of arcs containing v_i and in which v_i is not the last element, and the losing score $r(v_i)$ or r_i of a vertex v_i is the number of arcs containing v_i and in which v_i is the last element. The score sequence (losing score sequence) is formed by listing the scores (losing scores) in non-decreasing order.

For two integers p and q,

$$\binom{p}{q} = \frac{p!}{q!(p-q)!}$$

if $p \geq q$ and

$$\binom{p}{q} = 0$$

if $p < q$.

The following characterisation of losing score sequences in k-hypertournaments is due to Zhou et al. [8].

Theorem 2 *Given two non-negative integers* n *and* k *with* $n \geq k > 1$*, a non-decreasing sequence* $R = [r_1, r_2, \ldots, r_n]$ *of non-negative integers is a losing score sequence of some* k*-hypertournament if and only if for each* j*,*

$$\sum_{i=1}^{j} r_i \geq \binom{j}{k}, \tag{1}$$

with equality when $j = n$.

2 New proof

Koh and Ree [4] have given a different proof of Theorem 2. Some more results on scores of k-hypertournaments can be found in [3, 7]. The following is the new and short proof of Theorem 2.

Proof. The necessity part is obvious.

We prove sufficiency by contradiction. Assume all sequences of non-negative integers in non-decreasing order of length fewer than n, satisfying conditions (1) are losing score sequences. Let n be the smallest length and r_1 be the smallest possible with that choice of n such that $R = [r_1, r_2, \dots, r_n]$ is not a losing score sequence.

Consider two cases, (a) equality in (1) holds for some $j < n$, and (b) each inequality in (1) is strict for all $j < n$.

Case (a). Assume j $(j < n)$ is the smallest such that

$$\sum_{i=1}^{j} r_i = \binom{j}{k}.$$

By the minimality of n, the sequence $[r_1, r_2, \dots, r_j]$ is the losing score sequence of some k-hypertournament H_1. Also

$$\begin{aligned}\sum_{i=1}^{m}[r_{j+i} - \frac{1}{m}\sum_{i=1}^{k-1}\binom{j}{i}\binom{n-j}{k-i}] &= \sum_{i=1}^{m+j} r_i - \binom{j}{k} - \sum_{i=1}^{k-1}\binom{j}{i}\binom{n-j}{k-i}\\ &\geq \binom{m+j}{k} - \binom{j}{k} - \sum_{i=1}^{k-1}\binom{j}{i}\binom{n-j}{k-i}\\ &= \binom{m}{k},\end{aligned}$$

for each m, $1 \leq m \leq n-j$, with equality when $m = n-j$.

Let

$$\frac{1}{m}\sum_{i=1}^{k-1}\binom{j}{i}\binom{n-j}{k-i} = \alpha.$$

Therefore, by the minimality of n, the sequence

$$[r_{k+1} - \alpha, r_{k+2} - \alpha, \dots, r_n - \alpha]$$

is the losing score sequence of some k-hypertournament H_2. Taking disjoint union of H_1 and H_2, and adding all $m\alpha$ arcs between H_1 and H_2 such that

each arc among $m\alpha$ has the last entry in H_2 and each vertex of H_2 gets equal shares from these $m\alpha$ last entries, we obtain a k-hypertournament with losing score sequence R, which is a contradiction.
Case (b). Let each inequality in (1) is strict when $j < n$, and in particular $r_1 > 0$. Then the sequence $[r_1 - 1, r_2, \dots, r_n + 1]$ satisfies (1), and therefore by minimality of r_1, is the losing score sequence of some k-hypertournament H, a contradiction. Let x and y be the vertices respectively with losing scores $r_n + 1$ and $r_1 - 1$. If there is an arc e containing both x and y with y as the last element in e, let $e' = (x, y)$. Clearly, $(H - e) \cup e'$ is the k-hypertournament with losing score sequence R, again a contradiction. If not, since $r(x) > r(y)$ there exist two arcs of the form

$$e_1 = (w_1, w_2, \dots, w_{l-1}, u, w_l, \dots, w_{k-1})$$

and

$$e_2 = (w'_1, w'_2, \dots, w'_{k-1}, v),$$

where $(w'_1, w'_2, \dots, w'_{k-1})$ is a permutation of $(w_1, w_2, \dots, w_{k-1})$,
$x \notin \{w_1, w_2, \dots, w_{k-1}\}$ and $y \notin \{w_1, w_2, \dots, w_{k-1}\}$. Then, clearly R is the losing score sequence of the k-hypertournament $(H - (e_1 \cup e_2)) \cup (e'_1 \cup e'_2)$, where $e'_1 = (u, w_{k-1})$, $e'_2 = (w'_t, v)$ and t is the integer with $w'_t = w_{k-1}$. This again contradicts the hypothesis. Hence, the result follows. □

References

[1] R. A. Brualdi, J. Shen, Landau's inequalities for tournament scores and a short proof of a theorem on transitive sub-tournaments, *J. Graph Theory*, **38** (2001) 244–254. ⇒6

[2] A. Iványi, Reconstruction of complete tournaments, *Acta Univ. Sapientiae, Informatica*, **1,** 1 (2009) 71–88. ⇒6

[3] Y. Koh, S. Ree, Score sequences of hypertournament matrices, *J. Korea Soc. Math. Educ. Ser. B Pure Appl. Math.*, **38** (2001) 183–190. ⇒7

[4] Y. Koh, S. Ree, Score sequences of hypertournament matrices, On k-hypertournament matrices, *Linear Algebra Appl.*, **373** (2003) 183–195. ⇒ 7

[5] H. H. Landau, On dominance relations and the structure of animal societies, *Bull. Math. Biophys.*, **15** (1953) 143–158. ⇒5, 6

[6] K. B. Reid, Tournaments: scores, kings, generalisations and special topics, *Congr. Numer.*, **115** (1996) 171–211. ⇒6

[7] C. Wang, G. Zhou, Note on degree sequences of k-hypertournaments, *Discrete Math.* **308,** 11 (2008) 2292–2296. ⇒7

[8] G. Zhou, T. Yao, K. Zhang, On score sequences of k-tournaments, *European J. Comb.*, **21,** 8 (2000) 993–1000. ⇒5, 6

Received: December 6, 2009 • Revised: January 26, 2010

Acta Univ. Sapientiae, Informatica, **2,** 1 (2010) 10–27

Start-phase control of distributed systems written in Erlang/OTP

Péter Burcsi
Eötvös Loránd University
Faculty of Informatics
Department of Computer Algebra
email:
`peter.burcsi@compalg.inf.elte.hu`

Attila Kovács
Eötvös Loránd University
Faculty of Informatics
Department of Computer Algebra
email:
`attila.kovacs@compalg.inf.elte.hu`

Antal Tátrai
Eötvös Loránd University
Faculty of Informatics
Department of Computer Algebra
email:
`antal.tatrai@compalg.inf.elte.hu`

Abstract. This paper presents a realization for the *reliable* and *fast* startup of distributed systems written in Erlang. The traditional startup provided by the Erlang/OTP library is sequential, parallelization usually requires unsafe and ad-hoc solutions. The proposed method calls only for slight modifications in the Erlang/OTP `stdlib` by applying a system dependency graph. It makes the startup safe, quick, and it is equally easy to use in newly developed and legacy systems.

1 Introduction

A distributed system is usually a collection of processors that may not share memory or a clock. Each processor has its own local memory. The processors

Computing Classification System 1998: C.4
Mathematics Subject Classification 2010: 68M20
Key words and phrases: Erlang, parallel start-up

in the system are connected through a communication network. Communication takes place via messages [11]. Forms of messages include function invocation, signals, and data packets. Computation based models on message passing include the actor model and process algebras [4]. Several aspects of concurrent systems written in message passing languages have been studied including garbage collection [2], heap architectures [7], or memory management [8]. Startup concurrency is an area not fully covered yet.

Why is the investigation of the startup phase important?

- During system and performance testing, when the system is frequently started and stopped, fast startup might be beneficial.
- Critical distributed systems often have the maintainability requirement of 99.999 availability, also known as the "five nines". In order to comply with the "five nines" requirement over the course of a year, the total boot time could not take more than 5.25 minutes. In practice, due to the maintenance process, every system has a planned down time. In this case a fast and reliable startup is a must.
- The startup time is not the only reason to study the startup phase. In most product lines the requirements (and therefore the code) alter continuously. The changes may influence the code structure, which may affect the execution order of the parts. Although the code can often be reloaded without stopping the system, the changes may influence the startup. The challenge is to give a generic solution which supports reliable, robust and fast startup even when some software and/or hardware parts of the system had been changed.

In this paper we focus on the distributed programming language Erlang. Erlang was designed by the telecommunication company Ericsson to support fault-tolerant systems running in soft real-time mode. Programs in Erlang consist of functions stored in modules. Functions can be executed concurrently in lightweight processes, and communicate with each other through asynchronous message passing. The creation and deletion of processes require little memory and computation time. Erlang is an open source development system having a distributed kernel [6].

The Erlang system has a set of libraries that provide building primitives for larger systems. They include routines for I/O, file management, and list handling. In practice Erlang is most often used together with the library called the Open Telecom Platform (OTP). OTP consists of a development system platform for building, and a control system platform for running telecommunication applications. It has a set of design principles (behaviours), which together with middleware applications yield building blocks for scalable robust real time systems. Supervision, restart, and configuration mechanisms are provided. Various mechanisms, like an ORB, facilitate the development of CORBA based management systems. Interfaces towards other languages include a Java interface, an interface allowing Erlang programs to call C modules, etc. These interfaces are complemented with the possibility of defining IDL interfaces, through which code can be generated. The number of Erlang/OTP applications and libraries is continuously increasing. There are for example SNMP agents, a fault tolerant HTTP server, a distributed relational database called Mnesia, etc. One of the largest industrial applications developed in Erlang/OTP is the AXD 301 carrier-class multi-service (ATM, IP, Frame-relay, etc.) switching system of Ericsson. It is a robust and flexible system that can be used in several places of networks. It has been developed for more than 10 years (for an early announcement of the product, see [5]), resulting in a long product line. It contains several thousand Erlang modules and more than a million lines of code.

How is the startup of an Erlang application performed? The traditional startup provided by the Erlang/OTP library is sequential. It was not designed to start as quickly as possible, no special attention was paid to the possibility of parallelizing the different operations performed during startup. The only order imposed is due to the explicit dependencies described in the application configuration files. Technically, the reason of the sequential startup is that each process performing an OTP behaviour sends an ACK (acknowledge) signal to its parent only after the whole initialization process is finished. It means that each process has implicit preconditions. In the concurrent case, maintaining these preconditions is a fundamental problem. The proposed solution enables the concurrent startup and provides an Erlang/OTP extension for describing and realizing preconditions between behaviour processes. Hence the startup will not only be fast but remains reliable as well. The use of conditions to construct dependency graphs to manage the order of startup bears a resemblance to the mechanism used by Apple's MacOSX StartupItems. Each StartupItem includes a properties list of items that provides/requires/uses other items, which are used by the SystemStarter to build a soft dependency graph controlling the order of starting items [10]. There does not exist any such mechanism in Erlang/OTP.

Of course, the startup times do not only depend upon the dependencies among the applications and the degree to which these startup activities can be parallelized. The startup times are affected by several other factors, probably the most significant being disk I/O times and latencies, the time spent unnecessarily searching for hardware elements, disks, appropriate files to load, etc. In a particular system measurements are needed to find where the time goes on for startup. In this paper we do not focus on a particular system, we give instead a general solution for performing fast and reliable startup in any Erlang/OTP systems. It means that dependencies among the applications must be given in advance. These can be determined by the system designers.

The paper is structured as follows. For completeness, Section 2 contains the basic description of Erlang/OTP features and concepts. In Section 3 the basic idea of the concurrent startup of Erlang applications is presented. Section 4 deals with the details of the proposed solution presenting a prototype. The measurements of the performance of our prototypes are written in Section 5 and finally the authors write a show conclusion in Section 6.

2 Erlang/OTP

In this section a short description of Erlang/OTP concepts is given. The overview begins with a few Erlang language features, then OTP design principles and the startup mechanism are discussed. For a full description of Erlang with many examples the authors refer to the books [1, 3] and to the on-line documentation [6].

2.1 Code structure and execution

The code written in Erlang is structured as follows:

- *Functions* are grouped together in source files called *modules.* Functions that are used by other modules are exported, modules that use them must import them. Or alternatively, have to use the `apply(Mod, Fun, Arg)` built-in function, or the `module_name:function_name (args)` form.
- Modules that together implement some specific functionality, form an *application.* Applications can be started and stopped separately, and can be reused as parts of other systems. Applications do not only provide program or process structure but usually a directory structure as well. There is a descriptor file for each application containing the module names, starting parameters and many other data belonging to the application.
- A *release,* which is the highest layer, may contain several applications. It is a complete system which contains a subset of Erlang/OTP applications and a set of user-specific applications. A release is described

by a specific file, called release resource file. The release resource file can be used for generating `boot_scripts` for the system, and creating a *package* from it. After creating a `boot_script`, the system is able to start. First, the Erlang kernel is loaded. Then, a specific `gen_server` module (`application_controller`) is started. This module reads the application descriptor files sequentially, and creates a process called `application_master` for each application. The `application_master` starts the corresponding application, and sends an ACK signal back when the start is finished. Thus, as it was mentioned earlier, the Erlang/OTP startup is sequential.

The central concept of the execution is the process. As Erlang is message-oriented, executing Erlang code means creating strongly isolated processes that can only interact through message passing. Process creation, which is a lightweight operation, can be performed using the `spawn` family of functions. These functions create a parallel process and return immediately with the process ID (Pid). When a process is created in this way, we say that it is spawned. Erlang messages are sent in the form `Pid!Msg` and are received using `receive`.

2.2 Design principles

One of the most useful features in OTP is to have a pre-defined set of design patterns, called *behaviours.* These patterns were designed to provide an easy-to-use application interface for typical telecommunication applications such as client-server connections or finite state machines. In order to realize highly available and fault-tolerant systems, OTP offers a possibility to structure the processes into *supervision trees.*

2.2.1 Supervision trees

A principal OTP concept is to organize program execution into trees of processes, called supervision trees. Supervision trees have nodes that are either *workers* (leaves of the tree) or *supervisors* (internal nodes). The workers are Erlang processes which perform the functionality of the system, while supervisors start, stop, and monitor their child processes. Supervisor nodes can make decisions on what to do if an error occurs. Supervision tasks have a generic and a specific part. The generic part is responsible e.g. for the contact with the children, while the specific part defines (among other things) the restarting strategy. It is desirable that workers have a uniform interface, therefore OTP defines several behaviours with the same communication interface.

2.2.2 Behaviours

Behaviours, like every design pattern, provide a repeatable solution to commonly occurring problems. For example, a large number of simple server applications share common parts. Behaviours implement these common parts. A server code is then divided into a generic and a specific part. The generic part might contain the main loop of the server that is waiting for messages, and the specific part of the code contains what the server should do if a particular message arrives. In practice, only a *callback module* has to be implemented. OTP expects the existence of some functions (e.g. `handle_call`) in this module. As an example, several callback functions can be implemented for the complete functionality of an application, but the most important ones are: `start/2, stop/1`.

Let us summarize the most significant OTP behaviours: `gen_server`, `gen_fsm`, `gen_event` and `supervisor`. Each of them implements a basic pattern. The `gen_server` is the generic part of a server process, the `gen_event` is the generic part of event handling, the `gen_fsm` is the generic part of finite state machines. The `supervisor` behaviour is the generic part of the supervisor nodes of the supervision tree. Its callback module only contains the function `init(Arg)`, in which the children and the working strategy of the node can be specified. The `gen_server` behaviour also defines higher level functions for messaging, such as the synchronous (*call*) or asynchronous (*cast*) messages.

3 The basic idea of the solution

Let us suppose that we have an Erlang system and we plan to make the startup concurrent. If we use the `spawn` function instead of the built-in methods of supervisor child-starting then the spawned processes run parallelly, but the Erlang/OTP supervisor monitoring mechanism – one of the strongest Erlang/OTP features – is lost. Omitting the ACK mechanism from the built-in child-starting process would mean a deep redesign and reimplementation of the OTP (the ACK mechanism corresponds to sequential child-starting). Also, sequential start determines an order between processes, which would vanish using a too naive way of parallelization. Therefore an alternative parallel ordering is required to avoid dead-locks and startup crashes.

In the light of the previously mentioned properties we define our guidelines:

- The supervision tree structure, as well as other functionalities, must be preserved.
- The startup must be reliable and fast (faster than sequential).
- Only "small" modifications are permitted in the Erlang/OTP `stdlib`.

3.1 The dependency graph

In this subsection we consider dependence relations between modules and introduce the notion of dynamic dependency graphs.

In order to preserve the supervisor tree structure, we define *condition*s. Conditions represent the startup state of modules. A condition related to a module is false while the module's startup is being processed (or has yet to begin) and set to true when the corresponding startup has been finished. At the beginning of the startup all conditions are false. Conditions that the startup of another module depends on are called the preconditions of that module. A process can only start if all its preconditions are true. We can represent these relations in a dependency graph. Modules (or corresponding conditions) are the vertices, dependence between modules (or preconditions) are the directed edges in this graph.

When a behaviour module starts instruction defined by the first user (which is also the first that can imply preconditions) is the first instruction of the module's `init` function. Therefore the verification of the preconditions and setting up the completed conditions to true have to insert immediately before and after executing the `init` function.[1]

Dependency graphs are widely used in computer science. For example, dependency graphs are applied in the startup of the Mac OSX operating system [10] or in compiler optimization [9]. Moreover, a dependency graph is created when the Erlang boot script is generated from a given application.

However, there is a significant difference between the graphs above and our graph. The same Erlang software start up in different ways in different environments, therefore module parameters, execution and dependencies can vary and should be handled dynamically for full performance. In order to keep Erlang's robustness, we add one more guideline to the above:

- The dependency graph should be dynamic.

[1]We remark that during the sequential startup there exist implicit preconditions which are described in the hierarchy of the supervisor trees.

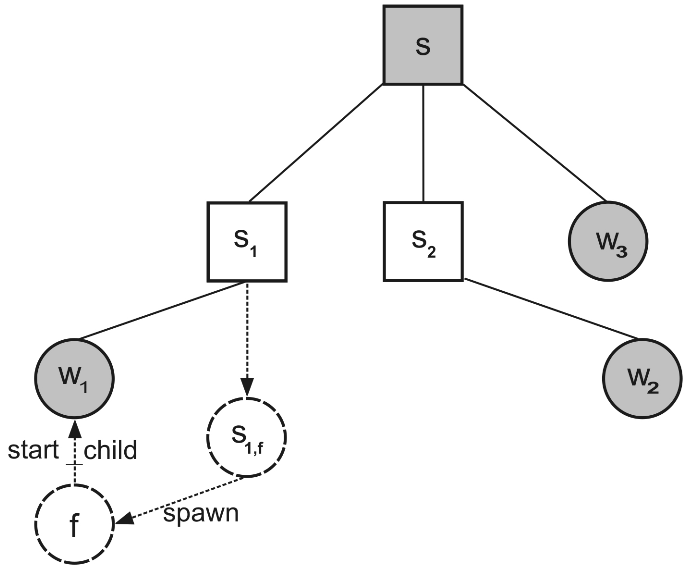


Figure 1: Inserting a *dummy* supervisor node for (1) preserving the supervisor's restarting behaviour, and (2) enabling fast parallel start-up. Supervisors are denoted by squares, permanent processes by continuous border, and temporary processes by dashed border. In the middle of the tree one can see the living processes (s_2, w_2) after termination of the dummy functions.

3.2 Concurrent startup of a supervisor's children

We also propose an Erlang trick that enables starting processes in a concurrent way. Here we can set which nodes should start concurrently. When a supervisor process s starts a child process w_1, the system starts a *dummy* (or *wrapper*) node s_1 instead. Then, the dummy process s_1 starts a simple function $\mathsf{s}_{1,f}$ (which just calls a `spawn` function) and sends an ACK message back immediately to its parent s. Consequently, the next child w_2 of the supervisor node s can start. So far function $\mathsf{s}_{1,f}$ has spawned function f. The spawned function f starts process w_1 and attaches it into the dummy process s_1 using the `supervisor::start_child` function[2]. The already started dummy process s_1 runs independently (parallel) from the other parts of the system. The ad-

[2]The `supervisor::start_child` function takes effect just after the ACK message has been sent back.

vantage of this method is that if process w_1 has a blocking precondition then only w_1 is waiting instead of the whole system. Figure 1 shows the described supervision hierarchy after start. The dummy supervisor node's restart strategy can be set in such a way that a crashing child results in the termination of the dummy supervisor. Thus the connection between s and w_1 is preserved.

The following code fragment shows the concurrent startup of a supervisor's children.

```
-module(dummy_sup_tree).

dummy_child({Tree_id, Child_spec}) ->
  spawn(dummy_sup_tree, child_starter, [{Tree_id, Child_spec}]),
  {ok, self()}.

child_starter({Tree_id, Child_spec}) ->
  supervisor:start_child(Tree_id, Child_spec),
  ok.

start_link({Child_spec}) ->
  supervisor:start_link(dummy_sup_tree, [{Child_spec}]).

init([{Child_spec}]) ->
  Sup_flags = {one_for_one, 0, 1},
  {ok,
   {Sup_flags,
    [
     {dummy_child_id, {dummy_sup_tree, dummy_child,
         [{self(), Child_spec}]}, temporary, brutal_kill,
         worker, [dummy_sup_tree, generic_server]}
    ]
   }
  }.
```

4 The solution's prototype

In this section we give the details of our solution by describing the skeleton of a prototype. We discuss the implementation of the dependency graph and how the Erlang boot script, the supervisors' `init` function, `stdlib` modules, etc. should be modified.

4.1 Realization of the dependency graph

The dependency graph is implemented as a module called `release_graph`. This module implements and exports the following functions: `get_conditions`, `get_preconditions` and `get_condition_groups`.

The `get_conditions` function returns a list of pairs. Each pair consists of a module name (with parameters) and a condition name.

```
{ {Mod, Args} , condition_name } .
```

We note that the function tag of the MFA (Module-Function-Arguments triplet) may be omitted, since it is always the `init` function of the module. The function `get_conditions` corresponds to the vertices of the dependency graph. Observe that a condition corresponds to a module together with parameters rather than a module, in accordance with our dynamic dependency graph guideline. In general, the `Args` parameter can be an actual parameter value or `undefined`. In the latter case the condition describes the module's startup with arbitrary parameters.

The `get_preconditions` function also gives a list. The elements of the list have the following structure:

```
{ { Mod, Args } , [ condition_names ] }  .
```

The function corresponds to the edges of the dependency graph. When a module's `init` function is called then the validity of the conditions in the list must be tested. Once again, the `Args` parameter can be `undefined` meaning that the startup of this module with any parameters has to wait until all conditions in the list become true.

The third function facilitates the management of dependence relations. Huge systems are likely to have many conditions and these conditions can be organized into groups. The `get_condition_groups` function returns a list of pairs of the form

```
{condition_group_name , [ conditions ] }  .
```

One can use the `condition_group_name` instead of the conditions defined in the list.

We remark that the dependency graph is not necessarily connected. Some modules are not preconditions of any other modules. In this case the definitions of the corresponding conditions are superfluous. Other modules do not have any preconditions, consequently they can be omitted from the return value of the `get_precondition` function.

Let's see an example. Let two applications `app1` and `app2` be given. The first has 3 server nodes that are controlled by a supervisor node. There is another server in the second application which has to wait for the complete startup of the first application. A possible implementation of the above functions might be:

```
...
get_conditions() ->
  [
   { {app1_rootsup   , undefined       } , cond_app1_rootsup },
   { {generic_server , [{app1_server1}]} , cond_app1_server1 },
   { {generic_server , [{app1_server2}]} , cond_app1_server2 },
   { {generic_server , [{app1_server3}]} , cond_app1_server3 }
  ].

get_condition_groups() ->
  [
   { group_app1_app , [ cond_app1_server1,
                        cond_app1_server2,
                        cond_app1_server3,
                        cond_app1_rootsup ] }
  ].

get_preconditions() ->
  [
   { {generic_server  , [{app2_server1}] } , [group_app1_app]  }
  ].
```

4.2 The condition server

The startup is controlled by a special server, called `condition_server`, which is started during the Erlang main system start. It stores and handles the dependency graph of the user programs. It also finds and loads the `release_graph` module and checks the validity of the data in it (checks for mistypes, not existing condition names, etc.). Clearly, any error in the `Args` fields remains undiscovered. If the `Args` tags are all `undefined` then the dependency graph is independent from the dynamic data. In this case, an acyclic dependency graph assures dead-lock free structure if each node that has preconditions is started in a concurrent way.

The `condition_server` performs the following two tasks based on the dependency graph. (1) First, sets the conditions belonging to the {`M,A`}s to true. This is implemented in the `set_condition({M,A})` function. (2) Second, it blocks the caller process until all its preconditions are satisfied. This is implemented in the `wait_for_conditions({M,A})` function. These functions have to be called by the generic parts of the behaviours (independently of the users' programs). Consequently, the `condition_server` must be implemented without the `gen_server` behaviour.

We remark that for those modules which don't have any preconditions or don't belong to any other module's precondition, the corresponding function call has no effect.

The `condition_server` module has to be a part of the Erlang kernel modules, since during the Erlang system's startup several event handler and server modules are started, and they require access to the condition storage system.

4.3 Modification of the supervisor behaviour

During the startup of a concurrent system, execution fork points must be named. In our case, these places are in the supervisor nodes. We modified the supervisor behaviour so that it accepts extended child specifications. The extension holds an additional field which can be `sequential` or `concurrent`. In the former case, the meaning of the child specification is equivalent with to original one. In the latter case, the supervisor node starts the child concurrently (fork point). We remark that the modification of the supervisor behaviour clearly accepts the original child specifications. The following example shows an extended child specification:

```
{app2_server1, {generic_server, start_link, [app2_server1]},
    permanent, 10, worker, [generic_server], concurrent}.
```

If the generic part of supervisors interprets a concurrent child specification it starts a dummy supervisor node with the proper parameters instead of the original child.

4.4 Further modifications of the Erlang system

It is also necessary to modify each Erlang behaviour before the callback `init` function is called, and after it returns successfully. We put these modifications into the `gen_server`, `gen_event`, `gen_fsm`, `supervisor_bridge` and `supervisor` behaviour.

The built-in utilities create boot scripts which do not start the `condition_server` automatically. In order to start the server, a new line has to be inserted into the boot script. The second line of the following code segment shows this:

```
...
  {kernelProcess,heart,{heart,start,[]}},
  {kernelProcess,condition_server,{condition_server,start,[]}},
  {kernelProcess,error_logger,{error_logger,start_link,[]}},
...
```

5 Implementation and measurements

We fully implemented the prototype described in the previous sections. The implementation can be used as an OTP extension. This extension is based on Erlang/OTP R11B version and the modifications affected the `stdlib`'s (v. 1.14.1) behaviour modules (namely: `gen_server`, `gen_fsm`, `gen_event`, `supervisor_bridge`, `supervisor`). You can download the prototype from the following url: `http://compalg.inf.elte.hu/projects/startup` .

Up to now, we described a parallel and reliable solution of the concurrent startup. Our solution gives a well-defined interface for handling the dependency problems among the concurrent starting modules. Therefore it preserves the reliability. It means that reliability of the concurrent startup is based on the dependency graph description of the users' programs. In the following we focus on the running time of the start-up.

We lack access to large industrial applications therefore we created programs for measuring the start-up time in several cases. For simplicity, no dependence conditions were defined, but concurrent supervisor child starting was performed. The measured programs use our modified Erlang/OTP libraries for making fast startup. The tested systems have some `gen_server` and some `supervisor` nodes. The `gen_server` nodes perform time-consuming, resource-intensive computations in their `init` functions. Each measured system has been started both sequentially and concurrently, the time is given in seconds. Each measurement has been performed five times and the figures show the average measured values. The measurements were performed on an SMP 4 machine with 2 AMD Dual Core Opteron, 2GHz, 16 GB RAM, Linux, Erlang 5.5, OTP R11B.

Three different system topologies were measured, a system with (1) deep process tree, (2) wide process tree, and (3) random process tree. The deep process tree was a 3-regular tree of depth 6, the wide process tree was a 10-regular tree of depth 2, and the random process tree was generated using uniform distribution from the range $[1, 5]$ for the number of children of a node, then truncating the tree at level 5.

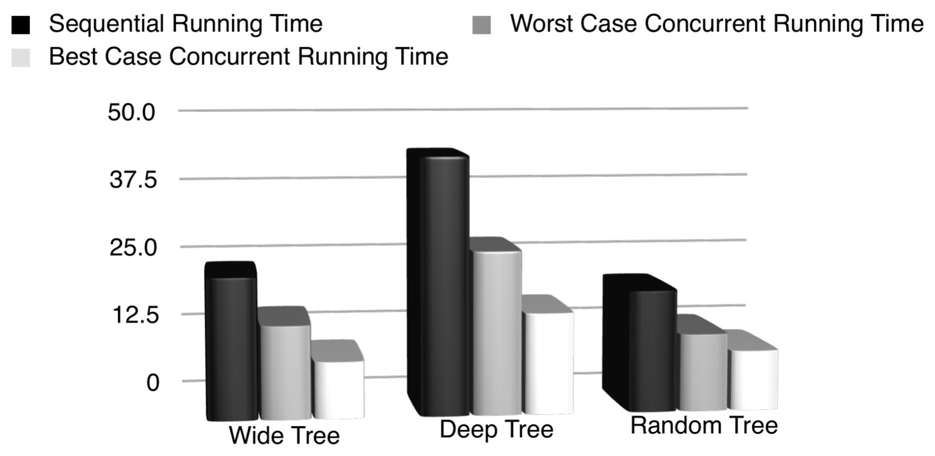


Figure 2: Start-up speed of the sequential and concurrent versions (in seconds). In concurrent case we can put different number of fork points into different places in the process tree. The authors created several concurrent cases for each kind of trees. The worst and best case values represent the slowest and the fastest concurrent start-up time in the proper kind of trees.

We measured the time that is needed for the system start-up as all servers and supervisors were started. The timer started when the `erl` shell was called and stopped when the last server or supervisor started. For this purpose we created a special application which starts just after all other servers or supervisors, and then immediately performs an illegal statement. Since this node crashes at once, consequently `erl` terminates. In other words, we measured the time between the starting and crashing of the Erlang shell.

There are several ways to make a system's process tree concurrent. We tagged the modules which have to start parallel. The speed of the startup depends on the number of the concurrent processes. The deeper the position of the fork point in the tree, the more parallel threads are created (more dummy supervisors). Therefore we show the running times as a function of the number of concurrent threads and as a function of the depth of fork points.

Figure 2 shows that the concurrent versions (not surprisingly) are always faster than the sequential ones. In some cases however, the concurrent start-up was two times faster than the sequential one.

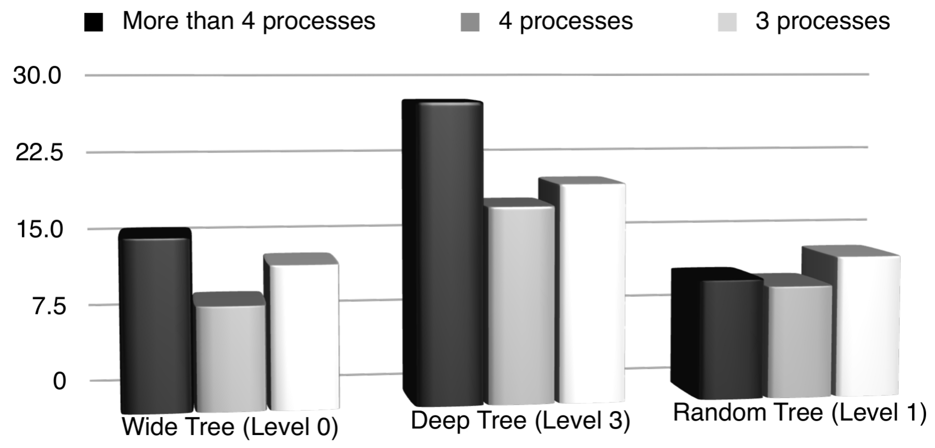


Figure 3: Start-up speed (in seconds) depending on the number of running processes, which can be set by the fork points. The levels show the depth of the fork points.

Figure 3 shows the startup speed as a function of the number of fork points. Since there were 4 processors (2 dual core) in the testbed, it is not surprising that 4-fold parallelism yields the best results. When only 3 parallel processes were started, one processor did not work, and 3 processors performed the whole startup. In case of more than 4 active processes, the processors had to switch between the active processes resulting in a serious overhead. Note however, that the most significant overhead in our measurements comes from the time consuming part of the servers' `init` functions.

Figure 4 shows how the results depend on the depth of the fork points. We measured a fall back performance when all nodes in a given level were started parallel. In this case the system had more concurrent processes in the deeper levels. One can also observe that the version of 4 active process forking is the most resistant to the depth. In this case the only overhead comes from the number of dummy supervisor trees. The measurement suggests that the system should be forked as close to the root as possible.

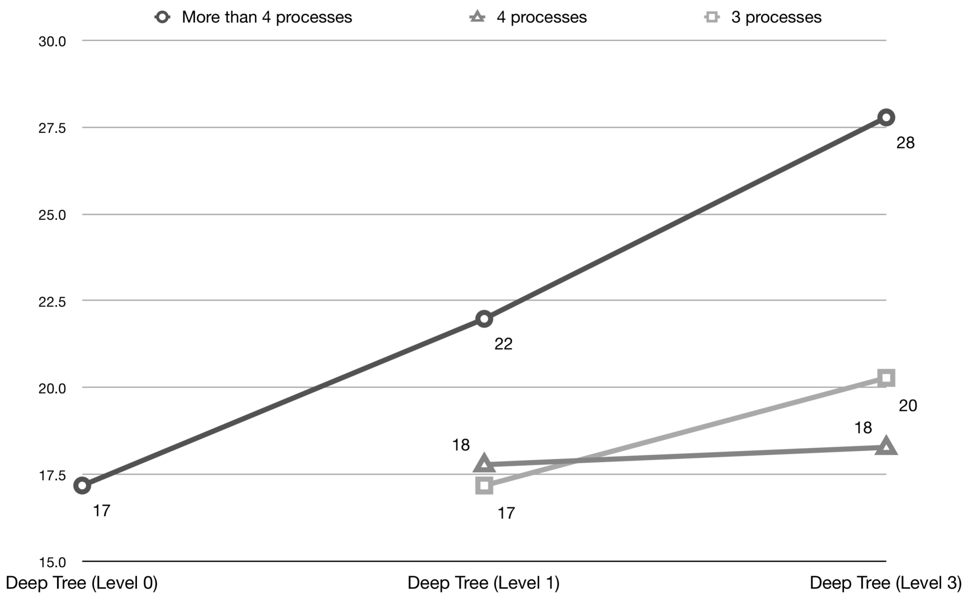


Figure 4: Start-up speed (in seconds) depending on the depth of the fork points.

6 Conclusions

In this paper we presented a solution for the parallel start-up of Erlang systems. We gave a general description of the solution and we measured the start-up time in several cases. Our measurements show that the parallel start-up can be much faster than the sequential. On the other hand our solution provides a well-defined mechanism for controlling the dependency relations among processes resulting reliable systems. The main advantages of our solution are:

- Precise and concise dependency handling.
- Preserving the supervision tree structures.
- The dependency graph is an Erlang module.

- The dependency graph is dynamic.
- Less than 150 lines modification in the `stdlib`.

Disadvantage of our solution is that bad dependency graph could result deadlock or system crash. We conclude that our solution is highly capable for the parallelization of Erlang systems' startup in case of legacy systems and new developments as well.

7 Acknowledgements

The authors would thank to the Lemon project (`https://lemon.cs.elte.hu`) that allowed us to use the project's main computer for measuring. We would also thank to Péter Nagy at Ericsson Telecommunications Hungary for answering our questions about Erlang and OTP.

The research was supported by the project TÁMOP-4.2.1/B-09/1/KMR-2010-003 of Eötvös Loránd University.

References

[1] J. Armstrong, *Making reliable distributed systems in the presence of software errors*, PhD Thesis, Stockholm, 2003. ⇒13

[2] J. Armstrong, R. Virding, One pass real-time generational mark-sweep garbage collection, *Proc. IWMM'95: International Workshop on Memory Management,* Lecture Notes in Computer Science, **986** (1995) 313–322. ⇒11

[3] J. Armstrong, R. Virding, C. Vikström, M. Williams *Concurrent Programming in Erlang,* Second edition, Prentice Hall, 1996. ⇒13

[4] M. Bell, *Service-oriented modelling: service analysis, design, and architecture*, Wiley, 2008. ⇒11

[5] S. Blau, J. Rooth, J. Axell, F. Hellstrand, M. Burgard, T. Westin, G. Wicklund, AXD 301 – a new generation ATM switching system, *Comput. Networks*, **31,** 6 (1999) 559–582. ⇒12

[6] Open Source Erlang, http://www.erlang.org ⇒11, 13

[7] E. Johansson, K. Sagonas, J. Wilhelmsson, Heap architectures for concurrent languages using message passing, *Proc. ISMM'2002: ACM SIGPLAN International Symposium on Memory Management*, 2002, pp. 88–99. ⇒ 11

[8] K. Sagonas, J. Wilhelmsson, Efficient memory management for concurrent programs that use message passing, *Sci. Comput. Programming*, **62** (2006) 98–121. ⇒11

[9] S. Horwitz, T. Reps, The Use of Program Dependence Graphs in Software Engineering, *Proc. 14th International Conference on Software Engineering*, 1992, pp. 392–411. ⇒16

[10] Mac OS X Startup Items,
http://developer.apple.com/documentation/MacOSX/ ⇒12, 16

[11] N. A. Lynch, *Distributed Algorithms,* First edition, Morgan Kaufmann, 1996 ⇒11

Received: August 7, 2009 • Revised March 5, 2010

Acta Univ. Sapientiae, Informatica, **2,** 1 (2010) 28–39

Using Coloured Petri Nets for design of parallel raytracing environment

Štefan Korečko
Department of Computers and Informatics, FEEI, Technical University of Košice, Letná 9, 041 20 Košice, Slovakia
email: `stefan.korecko@tuke.sk`

Branislav Sobota
Department of Computers and Informatics, FEEI, Technical University of Košice, Letná 9, 041 20 Košice, Slovakia
email: `branislav.sobota@tuke.sk`

Abstract. This paper deals with the parallel raytracing part of virtual-reality system PROLAND, developed at the home institution of authors. It describes an actual implementation of the raytracing part and introduces a Coloured Petri Nets model of the implementation. The model is used for an evaluation of the implementation by means of simulation-based performance analysis and also forms the basis for future improvements of its parallelization strategy.

1 Introduction

During the past several years, high-performance and feature-rich PC graphics interfaces have become available at low cost. This development enables us to build clusters of high-performance graphics PCs at reasonable cost. Then photorealistic rendering methods like raytracing or radiosity can be computed faster and inexpensively. Raytracing is one of computer graphics techniques used to produce accurate images of photorealistic quality from complex three-dimensional scenes described and stored in some computer-readable form [1, 2, 9]. It is based on a simulation of real-world optical processes. One great disadvantage of such techniques is that they are computationally very expensive and require massive amounts of floating point operations [4, 6]. Parallel

Computing Classification System 1998: I.6.3
Mathematics Subject Classification 2010: 68U20, 68U05
Key words and phrases: simulation, Coloured Petri Nets, parallel computing, raytracing

raytracing takes advantage of parallel computing, cluster computing in particular, to speed up image rendering, since this technique is inherently parallel. The use of clusters [8] for computationally intensive simulations and applications has lead to the development of interface standards such as the MPI and OpenPBS.

This paper provides insight into various means of decomposing the raytracing process (based on the free raytracer Pov-ray [12]) and describes a parallel raytracing process management simulation. We decided to use Coloured Petri Nets (CPNs) and CPN tools software for the simulation and a performance analysis based on it. The CPNs and CPN tools were chosen because of good simulation-based performance analysis support, familiar formalism and the fact that the tools are available for free. The other reason was that in the case of some more complicated raytracing process management design we can specify its analytical model using low-level Petri nets and use analytical "tools" of Petri nets, such as invariants, and others, including our own results in the field of formal methods, for a verification of the model. After that, the analytical model can be transformed to the CPN model suitable for a performance analysis.

2 Raytracing and its computation model

In nature, light sources emit rays of light, which travel through space and interact with objects and environment, by which they are absorbed, reflected, or refracted. These rays are then received by our eyes and form a picture.

Raytracing produces images by simulating these processes, with one significant modification. Emitting rays from light sources and tracking them would be very time-consuming and inefficient, because only a small fraction ends up in the eye/camera, the rest is irrelevant. So instead of this, raytracing works by casting rays from camera through image plane (for each pixel of final image) into the scene and tracking these rays. It computes the intersection of the ray with the first surface it collides with, examines the material properties (casting additional rays for refraction/reflection if necessary) and incoming light from light sources in the scene (by casting additional rays from intersection to each source) and then computes the colour of the pixel in the final image [12]. Raytracing belongs to a set of problems that utilize parallel computing very well, since it is computationally expensive and can be easily decomposed. The two main factors influencing the design and performance of parallel raytracing systems, are the computation model and the load-balancing mechanism [4].

There are two principal methods of decomposing a raytracing computation: demand-driven and data-driven (or data-parallel), and there are research activities focused on developing a hybrid model trying to combine the best features of the two models [6]. The final product of raytracer by demand-driven parallel raytracing is an image of $m \times n$ pixels, and since each pixel is computed independently, the most obvious way of decomposition is to divide the image into p parts, where p is a number of processors available and each processor would compute $m \times n/p$ pixels and ideally, the computation would be p times faster. This approach is called demand-driven parallel raytracing. A number of jobs are created each containing different subset of image pixels and these jobs are assigned to processors. Input scene is copied to local memory of each processor. Processors render their parts, return computed pixels, get another job if there is any, and in the end the final image is composed from these parts [1].

Main benefits of this approach are easy decomposition and implementation, simple job distribution and control and the fact that a general raytracing algorithm remains unchanged and scales well. The main disadvantage is that the input scene has to be copied to local memory of each processor, which poses a problem if the scene is very large.

Data-driven parallel raytracing approach, also called data-parallel raytracing, splits the input scene into a number of sections (tiles) and assigns these sections to processors [1, 2]. Each processor is responsible for all computations associated with objects in this particular section, no matter where the ray comes from. Only rays passing through the processor's section are traced. If a ray spawned at one processor needs data from another processor, it is transferred to that processor. The way the scene is divided into section determines the efficiency of parallel computation. Determining the number of rays that will pass through a section of the scene in order to estimate the sections requiring the most processing is one of the hardest problems to overcome. Using the cost function can be helpful. Main benefit of this approach is that the input scene doesn't have to be copied entirely to each processor, but it is split into sections, so even very large scenes can be processed relatively easy. Main disadvantage is that this approach doesn't scale very well with growing scene complexity and cluster size, because of task communication overhead and ray transfers [6].

2.1 Parallelization implementation

For parallel implementation a cluster-based computing system is used. Cluster-based rendering [8] in general can be described as the use of a set of computers

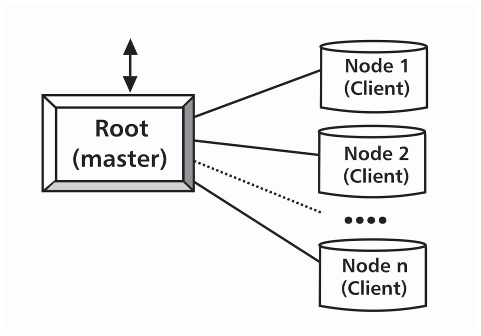


Figure 1: Basic structure for parallel raytracing implementation

connected via a network for rendering purposes, ranging from distributed non-photorealistic volume rendering over raytracing and radiosity-based rendering to interactive rendering using application programming interfaces like OpenGL or DirectX.

For the raytracing itself, a freeware program Pov-ray is used [7], and atop of Pov-ray, a front-end performing parallel decomposition and job control is built. Pov-ray is able to render only a selected portion of the picture, so it's very convenient for naive parallelization. Implementation is limited by Pov-ray's capabilities:

- only contiguous rectangular section of image can be rendered in one job,
- each job requires parsing the scene and initial computations all again,
- each Pov-ray job requires whole scene and
- program should be able to handle failures of individual nodes.

Because of these facts, the program implements demand-driven computation model. For a load balancing, static or dynamic load balancing by tiling decomposition seems to be the best choice. Implementation uses the Message Passing Interface and SPMD program model.

Fig. 1 shows the basic used structure. It isn't a typical master/slave scenario, here all nodes are equal, with the exception of the root node, which also controls the whole operation, allocates jobs and interacts with the user. That allows us to utilize massive parallelism. User puts in the scene to be rendered and additional control information. Root (master) node partitions the final image plane into sections (tiles) and allocates them to nodes. On each node, the process forks and executes Pov-ray to render its part of the image. When finished, it returns the rendered pixels and waits for another job, if required. At the end, the root node puts the whole image together and returns it to the user. It is a simple algorithm. We need to develop a better strategy for distribution of a scene section of rendered image. Better node, time and memory management is necessary. Because the development of an improved strategy using "real" hardware and software is expensive and very time-consuming, we decided to use formal CPN models and an appropriate simulation on them instead.

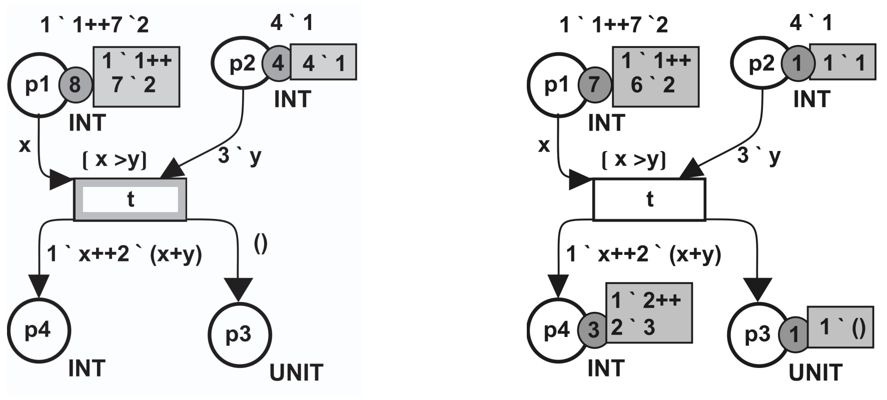
3 Coloured Petri Nets

Coloured Petri Nets (CPNs) [3] is a discrete-event formal modelling language, able to express properties such as non-determinism and concurrency. It combines a well-known Petri nets formalism with an individuality of tokens to enhance its modelling power and the CPN ML functional programming language to handle data manipulation and decision procedures.

A CPN model has a form of digraph with two types of vertices: places (ellipses) and transitions (rectangles).

Each place holds tokens of some type. In CPNs types are called colour sets. Colour sets range from simple ones as UNIT (with the only value "()"), INT, BOOL, to compound sets such as List, Record or Product. An example of user-defined colour sets (record and timed list) can be seen in Fig. 5. Tokens in places define state of CPN, which is called marking. Markings are represented as multisets, i.e. marking "$1`1 + +7`2$" of the place $\mathtt{p1}$ from Fig. 2 means that $\mathtt{p1}$ holds one token of value 1 and seven tokens of value 2. If there is only one token in a place, we can omit a number of tokens (for example we can write "4" instead of "$1`4$").

Transitions of CPN represent events that change the state (marking) of the net. A transition $\mathtt{t}$ can be executed, or fired, when there are enough tokens of corresponding value in places from which there is an arc to $\mathtt{t}$. These tokens are removed when $\mathtt{t}$ is fired and new tokens are generated in places to which there is an arc from $\mathtt{t}$. A number and values of removed and created tokens are determined by corresponding guarding predicates (guards), associated with transitions, and arc expressions. A small example in Fig. 2 illustrates the behaviour of CPN. The net in Fig. 2 has 3 places ($\mathtt{p1}$, $\mathtt{p2}$, $\mathtt{p4}$) of colour set INT and one place of colour set UNIT. Initially the net is in the (initial) marking with "$1`1 + +7`2$" in $\mathtt{p1}$ and 4 tokens of value 1 in $\mathtt{p2}$ (Fig. 2(a)). An actual marking is shown in boxes left to the places. The transition $\mathtt{t}$ with the guard "$x > y$" can be fired only for $x = 2$ and $y = 1$ now. The net after the firing is shown in Fig. 2(b).



(a) (b)

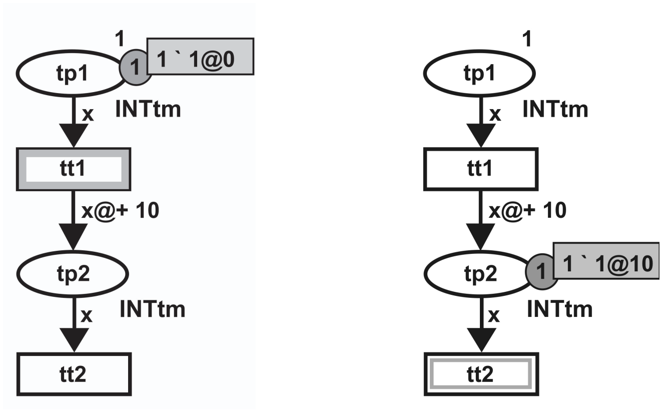
Figure 2: CPN fragment before (a) and after (b) the firing of the transition t

3.1 Performance analysis with CPNs

To broaden the scope of CPNs usage, facilities allowing simulation-based performance analysis have been added to both the CPNs language and its supporting tool, called CPN tools [11]. These facilities include time concept for CPNs (timed CPNs), random distribution functions for CPN ML and data collecting and simulation control monitors for CPN tools.

A (model) time in CPNs and CPN tools is represented as an integer value. There are also values, called time stamps, associated with tokens, representing a minimal time when the tokens are ready for firing. Colour sets of such tokens must be timed. In our models we distinguish timed colour sets by a postfix “tm” or “Tm”. The model time doesn't change while there is some transition that can be fired. When there is no transition to fire, the time advances to the nearest time value with some transitions to fire. All time-related information in markings, expressions and guards is prefixed by “@”. A small example of timed CPN can be seen in Fig. 3(a). Both places can hold timed integers. In the initial marking we have one token of value 1 and timestamp 0 in tp1. So, tt1 can fire in time = 0. After firing of tt1 one token of value 1 and time stamp = firing time +10 appears in tp2 (Fig. 3(b)). In addition, the model time advances to 10, because there is nothing to be fired in time = 0 and the token in tp2 will not be available (ready) before time = 10.

Because of space limitations we described CPNs very briefly here. An interested reader can find more information in [3, 10] or at [11].



(a) (b)

Figure 3: Timed CPN fragment before (a) and after (b) the firing of $\mathsf{tt1}$

4 CPN model for parallel raytracing

A timed CPN specification of our current implementation of distributed raytracing, as described in section 2.1, can be seen in Fig. 4. The time in our model is measured in milliseconds.

In the initial marking there are tokens in places $\mathsf{newScene}$, $\mathsf{nodesNo}$, $\mathsf{freeNodes}$, $\mathsf{scStartTime}$ and $\mathsf{preparedTiles}$. The place $\mathsf{newScene}$ holds one token with randomly chosen value from interval 10000 to 70000 (computed by the function *discrete*). This value characterizes a complexity of a scene to be raytraced and its range is based on our practical experience. In general the scene complexity depends on its size, number of objects, objects complexity (number of polygons), objects material (opacity, mirrors, ...), illumination model and camera parameters. The place $\mathsf{nodesNo}$ holds a token with number of computers in our cluster (8 computers) and $\mathsf{freeNodes}$ has one token for each node, where its value designates a type of the node. Albeit all the nodes are equal we have to distinguish between the client nodes (*type 2*) and the master node (*type 1*) that also manages the whole process. So, only about 70% of master performance is used for raytracing. The $\mathsf{preparedTiles}$ holds one token with empty list of tiles, because the scene is not divided yet.

Only the transition $\mathsf{sendScene}$ can be fired in the initial marking, in $\mathsf{time} = 0$. Its firing represents sending of the whole scene to each client node. Sending of the scene is a sequential process and its duration is computed by an expression

$$(\mathsf{nNo} - 1) * \mathsf{rnNormal\_int}(20000, 10000)$$

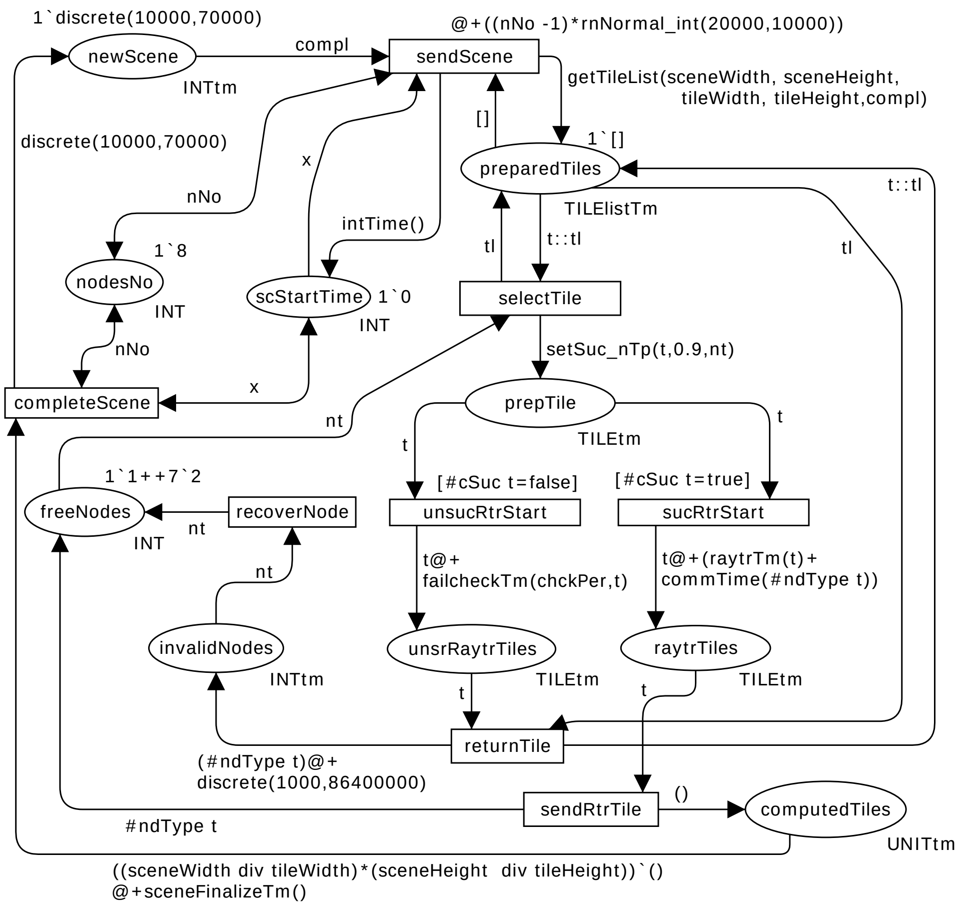


Figure 4: Timed CPN model of distributed raytracing in 8 computers cluster

where nNo is a number of all nodes and the function *rnNormal_int(m,v)* returns a value from the exponential random distribution with mean *m* and variance *v*. The firing also divides the scene into the list of tiles with (almost) constant width and height and saves the starting time point of scene raytracing as a token in scStartTime. To store information about a tile the colour set TILE is used (Fig. 5), where fields *wdt* and *hgt* store tile dimensions, *complxt* stores tile complexity, *cSuc* determines whether tile raytracing will be successful and *ndType* is a type of node where the tile will be raytraced. The list is generated by the function *getTileList* and is stored as a single token in

```
colset TILE = record wdt:INT *  hgt:INT * complxt:INT
                 * cSuc: BOOL*  ndType:INT;
colset TILElistTm = list TILE timed;
```

Figure 5: Declarations of some colour sets

```
fun tileCopml(0, remCmpl) = 0  |
    tileCopml(1, remCmpl) = remCmpl |
    tileCopml(remTiles, 0) = 0 |
    tileCopml(remTiles, remCmpl) =
let
  val tCmp = (Real.fromInt remCmpl /
                Real.fromInt remTiles)
  val cmpl=rnNormalr_int(tCmp*0.8,tCmp *0.7)
in
  if (remCmpl>cmpl) then cmpl else remCmpl
end;
```

Figure 6: Definition of `tileCopml` function

the place preparedTiles. The function also distributes the scene complexity randomly among the tiles. This random distribution is computed by the function *tileCopml* (Fig. 6), that is called within *getTileList*. Its first argument, *remTiles*, is a number of remaining tites to be added to the generated list and *remCmpl* is a complexity to be distributed among remaining tiles.

A firing of the transition selectTile means an assignment of raytracing job to a free node *nt*. The selected tile *t* is removed from the list in preparedTiles and moves to prepTile. In addition, the function *setSuc_nTp* assigns a node type (field *ndType*) to *t* and randomly chooses a raytracing job success (*cSuc*) for *t*. We assume that 90% of all jobs on client nodes will be successful and that the master node never fails. If the field *cSuc* of *t* is true (i.e. "#cSuc t = true" in CPN ML), then a firing of sucRtrStart moves *t* to raytrTiles. The timestamp of `t` is also increased by raytracing time and a communication delay. The raytracing time is computed by *raytrTm* from all fields of *t* except *cSuc*. The communication delay, computed by *commTime*, is taken from an exponential random distribution and represents the time needed to contact the master node, which can be busy performing other tasks, and to send the raytraced tile to it. After raytracing sendRtrTile moves the tile into already computed ones (computedTiles) and frees the node used.

The path of a fallen one begins with a firing of unsucRtrStart, which moves t to unsrRaytrTiles. The delay computed by *failcheckTm* is a time needed to detect that a given node failed and is not responding. The response of nodes is checked regularly in our implementation, so the delay computed is a randomly chosen multiple of checking period (*chckPer*) with some upper limit. Next, a firing of returnTile moves t back to the list in preparedTiles and the failed node to invalidNodes, where it waits for recovery. We optimistically suppose that each node recovers within one day. Finally the node is returned to freeNodes by a firing of recoverNode.

After successful processing of all tiles the scene can be finalized and the transition completeScene fired. Its firing removes all tokens from computedTiles and generates a new one in newScene, so a raytracing process can start over again. There is a data collecting monitor, which saves information about raytracing duration and number of used nodes into the text file for further processing when completeScene is fired.

5 Simulation experiments

To evaluate our implementation of parallel raytracing under various conditions we carried out several simulation experiments on the CPN model created. Here we present results concerning the relation between number of nodes in the cluster and duration of scene raytracing. In these experiments we fixed the scene complexity to 36500 and used a big scene with 30000×22500 pixels and a small one with 10000×7500 pixels. Tile dimensions were 1000×750 pixels in both cases. We considered two scenarios:

- an ideal scenario, where all nodes are equal (i.e. the master can use all of its performance for raytracing) and no computation fails and
- a real scenario, with conditions as described in Section 4.

Number of nodes ranged from 2 to 25. The results obtained are depicted in Fig. 7. As a reference we also included the raytracing duration when only one node is used. The values used in graphs are averages from multiple simulation runs. Of course, in the real scenario, the raytracing time is longer and the curve is not so “smooth” as in the ideal scenario. This is because in the real scenario some nodes can be invalid and raytracing time can be equal or even longer as in the cluster with fewer nodes. The results also reveal that it is not effective to use more than ten nodes in our current implementation of parallel raytracing environment.

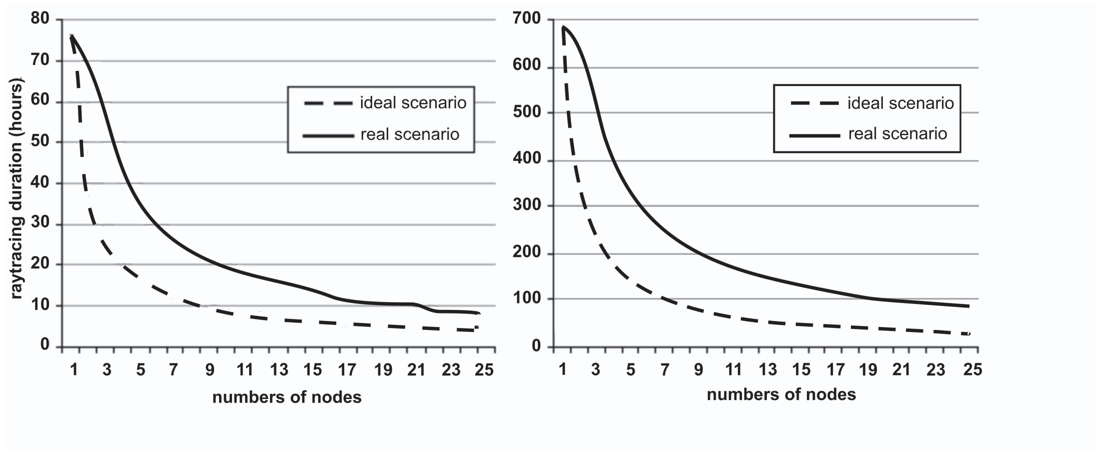


Figure 7: Number of nodes to raytracing duration ratio for 10000×7500 pixels scene (left) and 30000×22500 scene (right)

6 Conclusion

In this paper we presented our current implementation of distributed raytracing in a cluster environment. We also introduced a CPN model of the implementation, which has been used to evaluate a performance of the implementation and will be used as a basis for the development of an improved parallelization strategy. Our intention is to evaluate possible improvements on corresponding CPN models and choose the best with respect to the performance analysis. In the case of some more complicated strategy we can specify an analytical model using low-level Petri nets first and use analytical facilities of Petri nets, such as invariants and others, including our own theoretical and practical results [5], for a verification of the model.

7 Acknowledgements

The work presented has been supported by VEGA grant project No.1/0646/09: “Tasks solution for large graphical data processing in the environment of parallel, distributed and network computer systems” and by Agency of the Ministry of Education of the Slovak Republic for the Structural Funds of the EU under the project “Centre of Information and Communication Technologies for Knowledge Systems” (project number: 26220120020).

References

[1] A. Dietrich, E. Gobbetti, S.-E. Yoon, Massive-model rendering echniques: A tutorial, *IEEE Comput. Graph. Appl.*, **27,** 6 (2007) 20–34. ⇒28, 30

[2] I. Georgiev, P. Slusallek, RTfact: Generic concepts for flexible and high performance ray tracing, *Proc. IEEE/EG Symposium on Interactive Ray Tracing 2008*, Los Angeles, USA, 2008, pp. 115–122. ⇒28, 30

[3] K. Jensen, L.M. Kristensen and L. Wells, Coloured Petri Nets and CPN tools for modelling and validation of concurrent systems, *Int. J. Softw. Tools for Technol. Transfer*, **9,** 3–4 (2007) 213–254. ⇒32, 34

[4] A. Heirich, J. Arvo, A competitive analysis of load balancing strategies for parallel ray tracing, *J. of Supercomputing*, **12,** 1–2 (1998) 57–68. ⇒ 28, 29

[5] Š. Hudák, Š. Korečko, S. Šimoňák, A support tool for the reachability and other Petri nets-related problems and formal design and analysis of discrete systems, *Prob. Program.*, **20,** 2–3 (2008) 613–621. ⇒38

[6] I. Notkin, C. Gotsman, Parallel progressive ray-tracing, *Comput. Graph. Forum*, **16,** 1 (1997) 43–55. ⇒28, 30

[7] P. Rusyniak, *Photorealistic methods for large data processing*, Diploma Thesis, DCI FEEI TU Košice, 2008, 60 pag. (in Slovak). ⇒31

[8] B. Sobota, M. Straka, J. Perháč, A visualization in cluster environment, *Proc. 3rd Int. Workshop on Grid Computing for Complex Problems, GCCP'2007*, Bratislava, Slovakia, 2007, pp. 68–73. ⇒29, 30

[9] I. Wald et al., Applying ray tracing for virtual reality and industrial design, *Proc. IEEE Symposium on Interactive Ray Tracing 2006*, Salt Lake City, USA, 2006, pp. 177–185. ⇒28

[10] L. Wells, Performance analysis using CPN tools, *Proc. VALUETOOLS 2006*, Pisa, Italy, 2006. ⇒34

[11] *CPN tools homepage*, http://wiki.daimi.au.dk/cpntools. ⇒33, 34

[12] *What is ray-tracing?*, http://www.povray.org/documentation/view/3.6.0/4/. ⇒29

Received: August 30, 2009 • Revised February 28, 2010

Acta Univ. Sapientiae, Informatica, **2,** 1 (2010) 40–50

Automatic derivation of domain terms and concept location based on the analysis of the identifiers

Peter Václavík
Technical University of Košice
Faculty of Electrical Engineering and Informatics
Department of Computers and Informatics
email: `Peter.Vaclavik@tuke.sk`

Jaroslav Porubän
Technical University of Košice
Faculty of Electrical Engineering and Informatics
Department of Computers and Informatics
email: `Jaroslav.Poruban@tuke.sk`

Marek Mezei
Technical University of Košice
Faculty of Electrical Engineering and Informatics
Department of Computers and Informatics
email: `marekmezei@gmail.com`

Abstract. Developers express the meaning of the domain ideas in specifically selected identifiers and comments that form the target implemented code. Software maintenance requires knowledge and understanding of the encoded ideas. This paper presents a way how to create automatically domain vocabulary. Knowledge of domain vocabulary supports the comprehension of a specific domain for later code maintenance or evolution. We present experiments conducted in two selected domains: application servers and web frameworks. Knowledge of domain terms enables easy localization of chunks of code that belong to a certain term. We consider these chunks of code as "concepts" and their placement in the code as "concept location". Application developers may also benefit from the obtained domain terms. These terms are parts of speech that characterize a certain concept. Concepts are encoded in "classes" (OO paradigm) and

Computing Classification System 1998: D.2.8
Mathematics Subject Classification 2010: 68N99
Key words and phrases: program comprehension, domain knowledge, program quality, software measurement

the obtained vocabulary of terms supports the selection and the comprehension of the class' appropriate identifiers. We measured the following software products with our tool: JBoss, JOnAS, GlassFish, Tapestry, Google Web Toolkit and Echo2.

1 Introduction

Program comprehension is an essential part of software evolution and software maintenance: software that is not comprehended cannot be changed [5, 6, 7, 8].

Among the earliest results are the two classic theories of program comprehension, called *top-down* and *bottom-up* theories [9]. Bottom-up theory: Consider that understanding a program is obtained from source code reading and then mentally chunking or grouping the statements or control structures into higher abstract level, i.e. from bottom up. Such information is further aggregated until high-level abstraction of the program is obtained. Chunks are described as code fragments in programs. Available literature shows chunks to be used during the bottom-up approach of software comprehension. Chunks vary in size. Several chunks can be combined into larger chunks [1]. On the other hand, the top-down approach starts the comprehension process with a hypothesis concerning a high-level abstraction, which then will be further refined, leading to a hierarchical comprehension structure. The understanding of the program is developed from the confirmation or refutation of hypotheses.

An important task in program comprehension is to understand where and how the relevant concepts are located in the code. Concept location is the starting point for the desired program change. Concept location means a process where we assume that programmer understands the concept of the program domain, but does not know where is it located within the code. All domain concepts should map onto one or more fragments of the code. In other words, process of concept location is the process that finds that code-fragment [5].

Developers who are new to a project know little about the identifiers or comments in the source code, but it is likely that they have some knowledge about the problem domain of the software. In this paper, we present a new way of program comprehension that is based on naming of identifiers. When trying to understand the source code of a software system, developers usually start by locating familiar concepts in the source code. Keyword search is one of the most popular methods for this kind of task, but the success is strictly tied to the quality of the user queries and the words used to construct the identifiers and comments.

We present a way how to create a domain vocabulary automatically as a result of source code analysis. We classify the parts of speech and measure their occurrence in the source code.

2 Motivation

Domain level knowledge is important when programmers attempt to understand a program. Programmer inspects source code structure that is directed by identifiers. The quality and the "orthogonality" of the identifiers in the source code affects the time of program comprehension. Next kinds of quality could be measured:

1. percentage of *fullword* identifiers,
2. percentage of *abbreviations* and *unrecognized* identifiers,
3. percentage of *domain terms* identified in the application.

Percentage of full word identifiers is very important in the case of absence of documentation. The first two qualities could be derived directly from the source code toward common vocabulary. We don't need any additional domain data source to get relevant results. The third quality is not derived directly. We need to make measurements in order to obtain domain vocabulary.

Usually we don't have domain terms of the analysed software product. The question is: *How can we create the vocabulary of terms for a particular domain?* In this paper we propose a way to derive it automatically.

Nowadays, the companies are affected by employee fluctuation, especially in the IT sector. Each company has ongoing projects in the phase of developing or maintenance. New developer participating in the project has to understand project to solve the assigned task. Domain terms are usually in the specification. The transition from specification to implementation is bound usually to the transformation of terms. For example, if the specification contains word **car**, that word could be changed to word **vehicle** in the implementation phase. In spite of the fact that the word **vehicle** is a hypernym of the word **car**, we cannot find the word "vehicle" by brute force through searching by keywords. That is the reason for looking for some statistical evidence that car is a vehicle. It means that there exists "gap" in the meanings between the words used in specification and implementation. Our goal is to eliminate partly "this kind" of gap.

Developers of new software products may put another question: *What kind of parts of speech is usually used for a particular category of identifiers?* We can measure it directly from the source code. We can also find, if the rules are domain specific or generally applicable.

3 Methodology of programm inspection

Full word identifiers provide better comprehension then single letters or abbreviations [3]. It is the reason why we want to provide a tool for measurement of this aspect of program quality. We use the WordNet database of words to identify the potential domain terms.

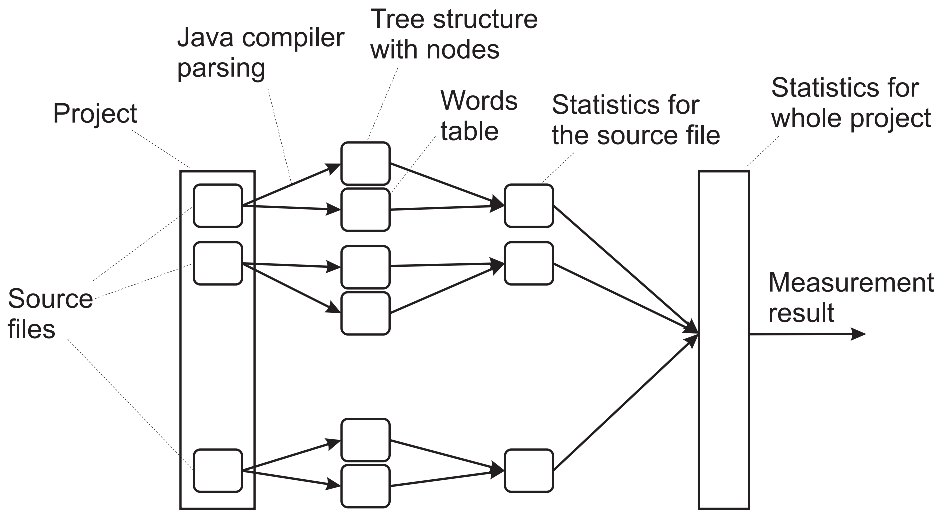

Java compiler parsing
Tree structure with nodes
Words table
Statistics for the source file
Statistics for whole project
Project
Source files
Measurement result

Figure 1: The methodology of program inspection

We apply our tool to well-known open-source projects. They belong to two domains: domain of application servers and domain of web frameworks. Each project consists of a set of source code files. We examine every source file separately. Based on information we have got by source files analysis we make measurements for the whole project. Our measurements follow the scheme shown on the Fig. 1.

1. First, we parse every source file using Java compiler. We build a tree structure of nodes. Each node belongs to one of the next types: *class*, *method*, *method parameter* or *class variable*. Then we process the names of each identified node. Name processing consists of splitting the name according to common naming conventions. For example, "setValue" is split into "set" and "value" words. After then we put all identified words into a table.
2. As a second step, we produce statistics for the source file. We examine which word belongs to class variable, method parameter, method or class

and also we try to assign part of speech to the words.

After source files analysis mentioned in previous steps we produce statistics for the whole project: we build a set of words containing all words used in the source files, and also we build a set of words used in the variables (class variables and method parameters), methods and classes. The set of words used in project will represent the software vocabulary for the particular project.

The software domain vocabulary represents the intersection of all software vocabularies of all software products of the same domain. Not all identified words are suitable candidates for the inclusion into software domain vocabulary. It is expected to apply filters in a process of source code analysis. So, the reason behind filtering is to eliminate terms that are irrelevant regarding the domain. As a final result we obtain a set of words ordered by occurrence. We obtain domain vocabulary as well as potential domain vocabulary (words are not identified in all measured software products).

As was mentioned in the previous section, the categorization in accordance to the parts of speech is expected in the experiment. It induces another problem: one word can belong to more parts of speech (e.g. "good" is adjective as well as noun). WordNet provides help in disambiguation and classification of words.

4 Experiments based on word analysis

WordNet provides a database of the most used words in the parts of speech. As was mentioned earlier, we have developed a tool to measure results in the graph, tabular and textual form. The tool's input is the project's source code. To present it we decide to inspect software products of two application domains:

- Java EE application server,
- Web framework.

We have selected next Java EE application servers:

- JOnAS 4.10.3 (http://jonas.ow2.org/),
- JBoss 5.0.1.GA (http://www.jboss.org/jbossas),
- GlassFish Server v2.1 (https://glassfish.dev.java.net/).

and web frameworks:

- Google Web Toolkit 1.5.3 (GWT) (http://code.google.com/intl/sk/webtoolkit/),

- Echo2 v2.1 (http://echo.nextapp.com/site/echo2),
- Tapestry 5.0.18 (http://tapestry.apache.org/).

	Glass.	**JOnAS**	**JBoss**	**Echo2**	**GWT**	**Tap.**
Number of source files	10553	3611	6448	402	593	1707
Number of words	10229	4502	5055	1044	2584	2154
Number of recognized words	4297 (42%)	2140 (48%)	2932 (58%)	903 (86%)	1582 (61%)	1714 (80%)
Number of not recognized words	5932 (58%)	2362 (52%)	2123 (42%)	141 (14%)	1002 (39%)	440 (20%)
Number of nouns	2361 (55%)	1311 (61%)	1687 (58%)	537 (60%)	839 (53%)	913 (54%)
Number of verbs	1259 (29%)	542 (25%)	842 (29%)	229 (25%)	484 (31%)	526 (30%)
Number of adjectives	549 (13%)	235 (11%)	330 (11%)	117 (13%)	213 (13%)	226 (13%)
Number of adverbs	128 (3%)	52 (3%)	73 (2%)	20 (2%)	46 (3%)	49 (3%)

Table 1: The number of recognized domain-terms for application servers and web frameworks

Table 1 summarizes data for the selected products. Other kind of results obtained from our tool in tabular form gives us information about identified words that are parts of software vocabulary. Now, each domain has three sets of software vocabulary. In our measurement we have selected the 50 most used words of each software vocabulary for further analysis. Their intersection is a set of terms belonging to the domain vocabulary. Table 2 presents the most used words and their occurrence. Words identified as domain terms are emphasized with bold letters. Potential domain terms recognized in two software products are emphasized with italic. Words that belong to only one software vocabulary are typed ordinary. Thanks to WordNet we can also identify semantically similar (synonyms, homonyms, hypernyms, and so on) words as a domain or potential domain term.

Glass.	**JBoss**	**JOnAS**	**GWT**	**Echo2**	**Tap.**
name (28727)	**name** (11774)	**name** (6950)	*type* (1774)	action (801)	**name** (1653)
value (9067)	**test** (7332)	**ejb** (2570)	**name** (1168)	*property* (506)	**value** (1140)
type (7550)	**id** (3169)	**test** (2179)	*method* (540)	**value** (475)	*type* (970)
class (6953)	**bean** (3154)	**id** (1602)	**class** (466)	*test* (395)	*class* (936)
object (5277)	**ejb** (2962)	*home* (1437)	*logger* (353)	*component* (369)	field (798)
id (4266)	**value** (2885)	**server** (1379)	**value** (344)	*element* (347)	page (778)
key (4641)	**type** (2473)	**type** (1251)	info (254)	**id** (331)	*component* (776)

Table 2: Application server and web framework domain terms recognition

5 An experiment on concept location

Within the next step we locate concepts encoded in keywords of the product. We use again WordNet for searching keywords. Programmers knows only the domain the software product it belongs to. They do not need to use exact words used in source code.

We present here an example of concept location. Lets suppose that somebody wants to change the algorithm for determining the parts of speech in our program. S/he needs to locate the concept of determining the parts of speech in the source code of the examined program. It is known that programmers and maintainers use different words to describe essentially the same or similar concepts [5]. Therefore the use of full-text search for concept location is very limited. We will try to find concepts based on semantic search.

In our example we assume that a concept is the identifier of a method or a class. We want to find a fragment in the source code where the parts of speech are located. We will try to find this code fragment based on this key-phrase: "*find word form*". For every keyword in our key-phrase we will make a database of related words – words that are in some semantic relationship to the keyword. Then, we will try to locate code fragment in our source code, where at least 1 occurrence for every keyword is found. This process is shown on the Fig. 2. However we are not looking only for the keywords itself, but also for semantically related words. In our example, as a result we find a method with this definition:

```
public String getType(String word) {
  //Method source code
}
```

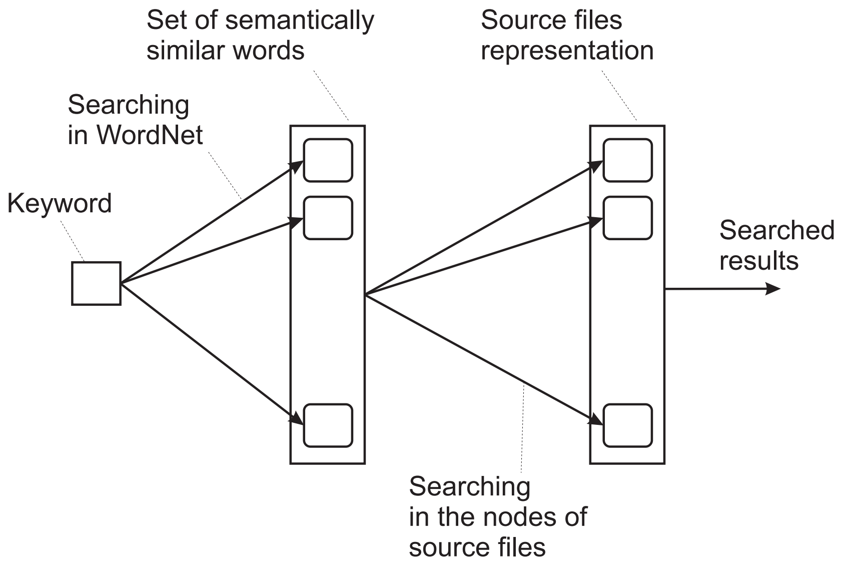

Set of semantically similar words
Source files representation
Searching in WordNet
Keyword
Searched results
Searching in the nodes of source files

Figure 2: Process of concept location

We found the three keywords in this method definition based on these semantic relations:

1. *find–get*: "get" is a hypernym of "find". We found the word "get" in the method name.
2. *word*: we found the term "word" itself in the parameter name.
3. *form–type*: "type" is a hyponym of "form". We found the word "type" in the method name.

We can see on this example that we could not locate this concept easily using fulltext search, but we can locate it using search based on semantic relations.

6 Related and further steps

The study of software vocabularies. This study is focused on three research questions: (1) to what degree relate terms found in the source code to a particular domain?; (2) which is the preponderant source of domain terms: identifiers or comments?; and (3) to what degree are domain terms shared between several systems from the same domain? Within the studied software, we found that in average: 42% of the domain terms were used in the source code; 23% of the domain terms used in the source code are present in comments only, whereas only 11% in the identifiers alone, and there is a 63% agreement in the use of domain terms between any two software systems. They manually selected the most common concepts, based on several books and online sources. They chose 135 domain concepts. From the same resources, for each of these concepts one or more terms and standard abbreviations that describe the concept were manually selected and included in the domain vocabulary [2].

Our aim was to define the domain vocabulary automatically. Results from the experiments will be used to build domain vocabularies for other domains too. These vocabularies support more detailed automatic classification of software products. Our next experiments will include inspection of comments in the source code. This stream of research is strongly promoted by [2, 10, 11].

Concept location. One of the experiments in the area of mapping between source code and conceptualizations shared as ontology has been published in [4]. The programs regard themselves as knowledge bases built on the programs' identifiers and their relations implied by the programming language. This approach extracts concepts from code by mapping the identifiers and the relations between them to ontology. As a result, they explicitly link the sources with the semantics contained in ontology. This approach is demonstrated using on the one hand the relations within Java programs generated by the type and the module systems and on the other hand the WordNet ontology.

We are locating concepts by keywords specified by programmers. Concept location is based on searching names in the identifiers that are in some relation to the specified keywords. This approach supports easier understanding of higher-level abstractions within the inspected application. We will work further on the concept visualization as well as on concept location refinement issues.

7 Conclusions

We can conclude the experiment results briefly as follows:

- In general, the most used parts of speech for all inspected element types are nouns (57%).
- Application servers as well as GWT use a lot of not recognized words due to different identifiers.
- The most number of recognized words is used in Tapestry (80%) and Echo2 (86%) web frameworks. We can assume that the source code of both products could be well understandable.
- From the comprehension point of view the application servers are more complex than web frameworks.
- In spite of application servers' complexity, they are using more common domain terms. Application server domain vocabulary consists of other well-known terms like: "context", "session", "service", and so on.
- Concept location gives us opportunity to find source code fragments more efficiently and with better results than using classical keyword search.

Acknowledgement

This work was supported by VEGA Grant No. 1/4073/07 - Aspect-oriented Evolution of Complex Software System and by APVV Grant No. SK-CZ-0095-07 - Cooperation in Design and Implementation of Language Systems.

References

[1] Ch. Aschwanden, M. Crosby, Code scanning patterns in program comprehension, *Proc. 39th Hawaii International Conference on System Sciences (HICSS)*, Hawaii, 2006. ⇒41

[2] S. Haiduc, A. Marcus, On the use of domain terms in source code, *Proc. 16th IEEE International Conference on Program Comprehension*, 2008, pp. 113–122. ⇒48

[3] D. Lawrie, C. Morrell, H. Feild, D. Binkley, What's in a name? A study of identifiers, *Proc. 14th IEEE International Conference on Program Comprehension*, Athens, Greece, June 2006, pp. 3–12. ⇒43

[4] D. Ratiu, F. Deissenböck, Programs are knowledge bases, *Proc. 14th IEEE International Conference on Program Comprehension*, 2006, pp. 79–83. ⇒48

[5] V. Rajlich, N. Wilde, The role of concepts in program comprehension, *Proc. 10th International Workshop on Program Comprehension*, 2002, pp. 271–278. ⇒41, 46

[6] L. Samuelis, Notes on the emerging science of software evolution, in: *Handbook of Research on Modern Systems Analysis and Design Technologies and Applications*, Hershey: Information Science Reference, 2008, pp. 161–167. ⇒41

[7] L. Samuelis, Cs. Szabó, On the role of the incrementality principle in software evolution, *Egyptian Comput. Sci. J.*, **29,** 2 (2007) 107–112. ⇒ 41

[8] Cs. Szabó, L. Samuelis, Notes on the software evolution within test plans, *Acta Electrotechnica et Inform.*, **8,** 2 (2008) 56–63. ⇒41

[9] M. A. Storey, Theories, methods and tools in program comprehension: past, present and future, *Proc. 13th International Workshop on Program Comprehension*, 2005, pp. 181–191. ⇒41

[10] B. L. Vinz, L. H. Etzkorn, A synergistic approach to program comprehension, *Proc. 14th IEEE international Conference on Program Comprehension*, 2006, pp. 69–73. ⇒48

[11] B. L. Vinz, L. H. Etzkorn, Improving program comprehension by combining code understanding with comment understanding, *Knowledge-Based Syst.*, **21,** 8 (2008) 813–825. ⇒48

Received: August 30, 2009 • Revised February 25, 2010

Acta Univ. Sapientiae, Informatica, **2,** 1 (2010) 51–79

A multivalued knowledge-base model

Ágnes Achs
University of Pécs
Faculty of Engineering
Department of Computer Science
email: achs@pmmk.pte.hu

Abstract. The basic aim of our study is to give a possible model for handling uncertain information. This model is worked out in the framework of DATALOG. At first the concept of fuzzy Datalog will be summarized, then its extensions for intuitionistic- and interval-valued fuzzy logic is given and the concept of bipolar fuzzy Datalog is introduced. Based on these ideas the concept of multivalued knowledge-base will be defined as a quadruple of any background knowledge; a deduction mechanism; a connecting algorithm, and a function set of the program, which help us to determine the uncertainty levels of the results. At last a possible evaluation strategy is given.

1 Introduction

The dominant portion of human knowledge can not be modelled by pure inference systems, because this knowledge is often ambiguous, incomplete and vague. Several and often very different approaches have been used to study the inference systems. When knowledge is represented as a set of facts and rules, this uncertainty can be handled by means of fuzzy logic.

A few years ago in [1, 2] a possible combination of Datalog-like languages and fuzzy logic was presented. In these works the concept of fuzzy Datalog has been introduced by completing the Datalog-rules and facts by an uncertainty level and an implication operator. The level of a rule-head can be inferred

Computing Classification System 1998: I.2.4
Mathematics Subject Classification 2010: 68T30
Key words and phrases: fuzzy Datalog, multivalued Datalog, fuzzy knowledge-base, multivalued knowledge-base

from the level of the body and the level of the rule by the implication operator of the rule. Based upon our previous works, later on a fuzzy knowledge-base was developed, which is a possible background of an agent-model [3]. In the last years new steps were taken into the direction of multivalued knowledge-base: the fuzzy Datalog was extended to intuitionistic- and interval-valued fuzzy logic and the concept of bipolar fuzzy Datalog was introduced [4, 6]. In this year the concept of fuzzy knowledgebase was generalized into multivalued direction [7]

This paper wants to give an overview of the knowledge-bases based on fuzzy- or multivalued Datalog. In the first part the fuzzy, the intuitionistic, the interval-valued and the bipolar extensions of Datalog will be summarized. In the second part the concept of a possible multivalued knowledge-base will be discussed. This knowledge-base is a quadruple of a deduction mechanism; a background knowledge; an algorithm connecting the deduction and the knowledge; and a decoding set computing the uncertainty levels of the consequences. At last a possible evaluation of knowledge-bases will be shown.

2 Extensions of Datalog

Datalog is a logical programming language designed for use as a data-base query language.

A Datalog program consists of facts and rules. Using these rules new facts can be inferred from the program's facts. It is very important that the solution of a program be logically correct. This means that evaluating the program, the result be a model of the first order logic formulas, being its rules. On the other hand it is also important that this model would contain only those true facts which are the consequences of the program, that is the minimality of this model is expected, i.e. in this model it is impossible to make any true fact false and still have a model consistent with the database. An interpretation assigns truth or falsehood to every possible instance of the program's predicates. An interpretation is a model, if it makes the rules true, no matter what assignment of values from the domain is made for the variables in each rule. Although there are infinite many implications, it is proved that it is enough to consider only the Herbrand interpretation defined on the Herbrand universe and the Herbrand base.

The Herbrand universe of a program P (denoted by $\mathsf{H_P}$) is the set of all possible ground terms constructed by using constants and function symbols occurring in P. The Herbrand base of P ($\mathsf{B_P}$) is the set of all possible ground atoms whose predicate symbols occur in P and whose arguments are elements of H_P.

In general a term is a variable, a constant or a complex term of the form $f(t_1, \dots, t_n)$, where f is a function symbol and $t_1, \dots, t_n$ are terms. An atom is a formula of the form $p(t)$, where p is a predicate symbol of a finite arity (say n) and t is a sequence of terms of length n (arguments). A literal is either an atom (positive literal) or its negation (negative literal). A term, atom or literal is ground if it is free of variables. As in fuzzy extension we did not deal with function symbols, so in our case the ground terms are the constants of the program.

In the case of Datalog programs there are several equivalent approaches to define the semantics of the program. In fuzzy extension we mainly rely on the fixed-point base aspect. The above concepts are detailed in classical works such as [15, 20, 21].

2.1 Fuzzy Datalog

In fuzzy Datalog (fDATALOG) we can complete the facts with an uncertainty level, the rules with an uncertainty level and an implication operator. We can infer for the level of a rule-head from the level of the rule-body and the level of the rule by the implication operator of the rule. As in classical cases, the logical correctness is extremely important as well, i.e., the solution would be a model of the program. This means that for each rule of the program, evaluating the fuzzy implication connecting to the rule, its truth-value has to be at least as large as the given uncertainty level. More precisely, the notion of fuzzy rule is the following:

An fDATALOG rule is a triplet $r; \beta; I$, where r is a formula of the form

$$A \leftarrow A_1, \dots, A_n (n \geq 0),$$

A is an atom (the head of the rule), $A_1, \dots, A_n$ are literals (the body of the rule); I is an implication operator and $\beta \in (0, 1]$ (the level of the rule).

For getting a finite result, all the rules in the program must be safe. An fDATALOG rule is safe if all variables occurring in the head also occur in the body, and all variables occurring in a negative literal also occur in a positive one. An fDATALOG program is a finite set of safe fDATALOG rules.

There is a special type of rule, called fact. A fact has the form $A \leftarrow; \beta; I$. From now on, we refer to facts as (A, β), because according to implication I, the level of A can easily be computed and in the case of the implication operators detailed in this paper it is β.

For defining the meaning of a program, we need again the concepts of Herbrand universe and Herbrand base. Now a ground instance of a rule $r; \beta; I$ in P is a rule obtained from r by replacing every variable in r with a constant of H_P. The set of all ground instances of $r; \beta; I$ is denoted by $ground(r); \beta; I$. The ground instance of P is $ground(P) = \cup_{(r;\beta;I)\in P}(ground(r); \beta; I)$.

An interpretation of a program P is a fuzzy set of the program's Herbrand base, B_P, i.e. it is: $\cup_{A\in B_P}(A, \alpha_A)$. An interpretation is a model of P if for each $(ground(r); \beta; I) \in ground(P)$, $ground(r) = A \leftarrow A_1, \ldots, A_n$

$$I(\alpha_{A_1\wedge\ldots\wedge A_n}, \alpha_A) \geq \beta$$

A model M is least if for any model N, $M \leq N$. A model M is minimal if there is no model N, where $N \leq M$.

To be short we sometimes denote $\alpha_{A_1\wedge\ldots\wedge A_n}$, by α_{body} and α_A by α_{head}.

In the extensions of Datalog several implication operators are used, but in all cases we are restricted to min-max conjunction and disjunction, and to the complement to 1 as negation. So: $\alpha_{A\wedge B} = min(\alpha_A, \alpha_B)$, $\alpha_{A\vee B} = max(\alpha_A, \alpha_B)$ and $\alpha_{\neg A} = 1 - \alpha_A$.

The semantics of fDATALOG is defined as the fixed points of consequence transformations. Depending on these transformations, two semantics can be defined [1]. The deterministic semantics is the least fixed point of the deterministic transformation DT_P, the nondeterministic semantics is the least fixed point of the nondeterministic transformation NT_P. According to the deterministic transformation, the rules of a program are evaluated in parallel, while in the nondeterministic case the rules are considered independently and sequentially. These transformations are the following:

The $ground(P)$ is the set of all possible rules of P the variables of which are replaced by ground terms of the Herbrand universe of P. $|A_i|$ denotes the kernel of the literal A_i, (i.e., it is the ground atom A_i, if A_i is a positive literal, and $\neg A_i$, if A_i is negative) and $\alpha_{body} = \min(\alpha_{A_1}, \ldots, \alpha_{A_n})$.

Let B_P be the Herbrand base of the program P, and let $F(B_P)$ denote the set of all fuzzy sets over B_P. The consequence transformations $DT_P : F(B_P) \to F(B_P)$ and $NT_P : F(B_P) \to F(B_P)$ are defined as

$$DT_P(X) = (\cup_{R_A\in ground(P)}\{(A, \alpha_A)\}) \cup X \tag{1}$$

and

$$NT_P(X) = \{(A, \alpha_A)\} \cup X, \tag{2}$$

where $R_A : (A \leftarrow A_1, \ldots, A_n; \beta; I) \in ground(P)$, $(|A_i|, \alpha_{A_i}) \in X$, $1 \leq i \leq n$; $\alpha_A = \max\left(0, \min\{\gamma \mid I(\alpha_{body}, \gamma) \geq \beta\}\right)$. $|A_i|$ denotes the kernel of the literal

A_i, (i.e., it is the ground atom A_i, if A_i is a positive literal, and $\neg A_i$, if A_i is negative) and $\alpha_{body} = \min(\alpha_{A_1}, \dots, \alpha_{A_n})$.

In [1] it is proved that starting from the set of facts, both DT_P and NT_P have fixed points which are the least fixed points in the case of positive P. These fixed points are denoted by $lfp(DT_P)$ and $lfp(NT_P)$. It was also proved, that $lfp(DT_P)$ and $lfp(NT_P)$ are models of P, so we could define $lfp(DT_P)$ as the deterministic semantics, and $lfp(NT_P)$ as the nondeterministic semantics of fDATALOG programs. For a function- and negation-free fDATALOG, the two semantics are the same, but they are different if the program has any negation. In this case the set $lfp(DT_P)$ is not always a minimal model, but the nondeterministic semantics – $lfp(NT_P)$ – is minimal under certain conditions. These conditions are referred to as stratification. Stratification gives an evaluating sequence in which the negative literals are evaluated first [2].

To compute the level of rule-heads, we need the concept of the uncertainty-level function, which is:

$$f(I, \alpha, \beta) = \min(\{\gamma \mid I(\alpha, \gamma) \geq \beta\}).$$

According to this function the level of a rule-head is: $\alpha_{head}= f(I, \alpha_{body}, \beta)$.

In the former papers [1, 2] several implications were detailed (the operators treated in [17]), and the conditions of the existence of an uncertainty-level function was examined for all these operators. For intuitionistic cases three of them is extended in this paper. They are the following:

Gödel	$I_G(\alpha, \gamma) = \begin{cases} 1 & \alpha \leq \gamma \\ \gamma & \text{otherwise} \end{cases}$	$f(I_G, \alpha, \beta) = \min(\alpha, \beta)$
Lukasiewicz	$I_L(\alpha, \gamma) = \begin{cases} 1 & \alpha \leq \gamma \\ 1 - \alpha + \gamma & \text{otherwise} \end{cases}$	$f(I_L, \alpha, \beta) = \max(0, \alpha + \beta - 1)$
Kleene-Dienes	$I_K(\alpha, \gamma) = \max(1 - \alpha, \gamma)$	$f(I_K, \alpha, \beta) = \begin{cases} 0 & \alpha + \beta \leq 1 \\ \beta & \alpha + \beta > 1 \end{cases}$

Example 1 *Let us consider the next program:*

$$\begin{aligned}
&(p(a), 0.8).\\
&(r(b), 0.6).\\
&s(x) \quad \leftarrow \; q(x,y); 0.7; I_L.\\
&q(x,y) \; \leftarrow \; p(x), r(y); 0.7; I_G.\\
&q(x,y) \; \leftarrow \; \neg q(y,x); 0.9; I_K.
\end{aligned}$$

As the program has a negation, so according to the stratification the right order of rule-evaluation is 2.,3,.1. Then

$$\begin{aligned}
&\mathrm{lfp}(NT_P) =\\
&\{(p(a), 0.8); (r(b), 0.6); (q(a,b), 0.6);\\
&(q(b,a), 0.9); (s(a), 0.3); (s(b), 0.6)\}.
\end{aligned}$$

2.2 Multivalued extensions of fuzzy Datalog

In fuzzy set theory the membership of an element in a fuzzy set is a single value between zero and one, and the degree of non-membership is automatically just the complement to 1 of the membership degree. However a human being who expresses the degree of membership of a given element in a fuzzy set, very often does not express a corresponding degree of non-membership as its complement. That is, there may be some hesitation degree. This illuminates a well-known psychological fact that linguistic negation does not always correspond to logical negation. Because of this observation, as a generalization of fuzzy sets, the concept of intuitionistic fuzzy sets was introduced by Atanassov in 1983 [9, 11]. In the next paragraphs some possible multivalued extensions will be discussed.

2.2.1 Intuitionistic and interval-valued extensions of fuzzy Datalog

While in fuzzy logic the uncertainty is represented by a single value (μ), in intuitionistic-(IFS) and interval-valued (IVS) fuzzy logic it is represented by two values, $\vec{\mu} = (\mu_1, \mu_2)$. In the intuitionistic case the two elements must satisfy the condition $\mu_1 + \mu_2 \leq 1$, while in the interval-valued case the condition is $\mu_1 \leq \mu_2$. In IFS μ_1 is the degree of membership and μ_2 is the degree of non-membership, while in IVS the membership degree is between μ_1 and μ_2. It is obvious that the relations $\mu_1' = \mu_1$, $\mu_2' = 1 - \mu_2$ create a mutual connection between the two systems. (The equivalence of IVS and IFS was stated first in [10].) In both cases an ordering relation can be defined, and according to this ordering a lattice is taking shape:

L_F and L_V are lattices of IFS and IVS respectively, where:

$$\begin{aligned} L_F &= \{(x_1, x_2) \in [0,1]^2 \mid x_1 + x_2 \leq 1\}, \\ &\quad (x_1, x_2) \leq_F (y_1, y_2) \Leftrightarrow x_1 \leq y_1, x_2 \geq y_2 \\ L_V &= \{(x_1, x_2) \in [0,1]^2 \mid x_1 \leq x_2\}, \\ &\quad (x_1, x_2) \leq_V (y_1, y_2) \Leftrightarrow x_1 \leq y_1, x_2 \leq y_2. \end{aligned}$$

It can be proved that both L_F and L_V are complete lattices [16]. In both cases the extended fDATALOG is defined on these lattices and the necessary concepts are generalizations of the ones presented above.

Definition 2 *The i-extended fDATALOG program (ifDATALOG) is a finite set of safe ifDATALOG rules* $(r; \vec{\beta}; \vec{I}_{FV})$*;*

- *the i-extended consequence transformations* iDT_P *and* iNT_P *are formally the same as* DT_P *and* NT_P *in (1), (2) except:* $\vec{\alpha}_A = \max(\vec{0}_{FV}, \min\{\vec{\gamma} \mid \vec{I}_{FV}(\vec{\alpha}_{body}, \vec{\gamma}) \geq_{FV} \vec{\beta}\})$ *and*
- *the* i*-extended uncertainty-level function is* $f(\vec{I}_{FV}, \vec{\alpha}, \vec{\beta}) = \min(\{\vec{\gamma} \mid \vec{I}_{FV}(\vec{\alpha}, \vec{\gamma}) \geq_{FV} \vec{\beta}\})$,

where $\vec{\alpha}$, $\vec{\beta}$, $\vec{\gamma}$ *are elements of* L_F, L_V *respectively,* $\vec{I}_{FV} = \vec{I}_F$ *or* $\vec{I}_V$ *is an implication of* L_F *or* L_V*,* $\vec{0}_{FV}$ *is* $\vec{0}_F = (0,1)$ *or* $\vec{0}_V = (0,0)$ *and* $\geq_{FV}$ *is* $\geq_F$ *or* $\geq_V$*.*

As iDT_P and iNT_P are inflationary transformations over the complete lattices L_F or L_V, thus according to [15] they have an inflationary fixed point denoted by $lfp(iDT_P)$ and $lfp(iNT_P)$. If P is positive (without negation), then $iDT_P = iNT_P$ is a monotone transformation, so $lfp(iDT_P) = lfp(iNT_P)$ is the least fixed point.

The fixed point is an interpretation of P, which is a model, if for each

$$(A \leftarrow A_1, \ldots, A_n; \vec{\beta}; \vec{I}_{FV}) \in \mathrm{ground}(P), \quad \vec{I}_{FV}(\vec{\alpha}_{body}, \vec{\alpha}_A) \geq_{FV} \vec{\beta}.$$

It can easily be proved that these fixed points are models of the program.

Proposition 3 $lfp(iDT_P)$ *and* $lfp(iNT_P)$ *are models of* P*.*

Proof. For $T = iDT_P$ or $T = iNT_P$ there are two kinds of rules in $\mathrm{ground}(P)$:
a/ $A \leftarrow A_1, \ldots, A_n; \vec{\beta}; \vec{I}_{FV}$; $(A, \vec{\alpha}_A) \in lfp(T)$; $(|A_i|, \vec{\alpha}_{A_i}) \in lfp(T)$, $1 \leq i \leq n$.

b/ $\mathsf{A} \leftarrow \mathsf{A}_1, \dots, \mathsf{A}_n; \vec{\beta}; \vec{\mathrm{I}}_{\mathsf{FV}}; \exists i : (|\mathsf{A}_i|, \vec{\alpha}_{\mathsf{A}_i}) \notin \mathsf{lfp}(\mathsf{T})$.
In the first case according to the construction of $\vec{\alpha}_{\mathsf{A}}$, $\vec{\mathrm{I}}_{\mathsf{FV}}(\vec{\alpha}_{\mathsf{body}}, \vec{\alpha}_{\mathsf{A}}) \geq_{\mathsf{FV}} \vec{\beta}$ holds, in the second case A_i is not among the facts, so $\vec{\alpha}_{\mathsf{A}_i} = \vec{0}_{\mathsf{FV}}$, therefore $\vec{\alpha}_{\mathsf{body}} = \vec{0}_{\mathsf{FV}}$ and $\vec{\mathrm{I}}_{\mathsf{FV}}(\vec{\alpha}_{\mathsf{body}}, \vec{\alpha}_{\mathsf{A}}) = \vec{1}_{\mathsf{FV}} \geq_{\mathsf{FV}} \vec{\beta}$. That is $\mathsf{lfp}(\mathsf{T})$ is a model. □

According to the above statements the next theorem is true:

Theorem 4 *Both* $\mathsf{iDT_P}$ *and* $\mathsf{iNT_P}$ *have a fixed point, denoted by* $\mathsf{lfp(iDT_P)}$ *and* $\mathsf{lfp(iNT_P)}$. *If* P *is positive, then* $\mathsf{lfp(iDT_P)} = \mathsf{lfp(iNT_P)}$ *and this is the least fixed point.* $\mathsf{lfp(iDT_P)}$ *and* $\mathsf{lfp(iNT_P)}$ *are models of* P*; for negation-free ifDATALOG this is the least model of the program.*

As the intuitionistic or interval-valued extension of Datalog has no influence on the stratification, the propositions detailed in the case of stratified fDATALOG programs are true in the case of i-extended fuzzy Datalog programs as well:

Proposition 5 *For stratified ifDATALOG program* P*, there is an evaluation sequence, in which* $\mathsf{lfp(iNT_P)}$ *is a unique minimal model of* P.

After defining the syntax and semantics of i-extended fuzzy Datalog, it is necessary to examine the properties of possible implication operators and the i-extended uncertainty-level functions. A number of intuitionistic implications are established in [16, 12, 13] and other papers, four of which are the extensions of the above three fuzzy implication operators thus chosen for now. For these operators the suitable interval-valued operators will be decided, and for both kinds of them we will deduce the uncertainty-level functions. Now the computations will not be shown, only the starting points and results are presented.

The coordinates of intuitionistic and interval-valued implication operators can be determined by each other. The uncertainty-level functions can be computed according to the applied implication. The connection between $\mathrm{I_F}$ and $\mathrm{I_V}$ and the extended versions of uncertainty-level functions are given below:

$$\vec{\mathrm{I}}_{\mathsf{V}}(\vec{\alpha}, \vec{\gamma}) = (\mathrm{I}_{\mathsf{V1}}, \mathrm{I}_{\mathsf{V2}}) \text{ where}$$

$$\begin{aligned} \mathrm{I}_{\mathsf{V1}} &= \mathrm{I}_{\mathsf{F1}}(\vec{\alpha}', \vec{\gamma}'), & \vec{\alpha}' &= (\alpha_1, 1 - \alpha_2), \\ \mathrm{I}_{\mathsf{V2}} &= 1 - \mathrm{I}_{\mathsf{F2}}(\vec{\alpha}', \vec{\gamma}')); & \vec{\gamma}' &= (\gamma_1, 1 - \gamma_2). \end{aligned}$$

$$\begin{aligned} f(\vec{I}_F, \vec{\alpha}, \vec{\beta}) &= (\min(\{\gamma_1 \mid I_{F1}(\vec{\alpha}, \vec{\gamma}) \geq \beta_1\}), \max(\{\gamma_2 \mid I_{F2}(\vec{\alpha}, \vec{\gamma}) \leq \beta_2\})); \\ f(\vec{I}_V, \vec{\alpha}, \vec{\beta}) &= (\min(\{\gamma_1 \mid I_{V1}(\vec{\alpha}, \vec{\gamma}) \geq \beta_1\}), \min(\{\gamma_2 \mid I_{V2}(\vec{\alpha}, \vec{\gamma}) \geq \beta_2\})). \end{aligned}$$

The studied operators and the related uncertainty-level functions are the following:

Extension of the Kleene-Dienes implication

One possible extension of the Kleene-Dienes implication for IFS is:

$$\vec{I}_{FK}(\vec{\alpha}, \vec{\gamma}) = (\max(\alpha_2, \gamma_1), \min(\alpha_1, \gamma_2)).$$

The appropriate computed elements are:

$$\vec{I}_{VK}(\vec{\alpha}, \vec{\gamma}) = (\max(1 - \alpha_2, \gamma_1), \max(1 - \alpha_1, \gamma_2));$$

$$f_1(\vec{I}_{FK}, \vec{\alpha}, \vec{\beta}) = \begin{cases} 0 & \alpha_2 \geq \beta_1 \\ \beta_1 & \text{otherwise} \end{cases}, \quad f_1(\vec{I}_{VK}, \vec{\alpha}, \vec{\beta}) = \begin{cases} 0 & 1 - \alpha_2 \geq \beta_1 \\ \beta_1 & \text{otherwise} \end{cases},$$

$$f_2(\vec{I}_{FK}, \vec{\alpha}, \vec{\beta}) = \begin{cases} 1 & \alpha_1 \leq \beta_2 \\ \beta_2 & \text{otherwise} \end{cases}, \quad f_2(\vec{I}_{VK}, \vec{\alpha}, \vec{\beta}) = \begin{cases} 0 & (1 - \alpha_1 \leq \beta_2 \\ \beta_2 & \text{otherwise} \end{cases}.$$

Extension of the Lukasiewicz implication

One possible extension of the Lukasiewicz implication for IFS is:

$$\vec{I}_{FL}(\vec{\alpha}, \vec{\gamma}) = (\min(1, \alpha_2 + \gamma_1), \max(0, \alpha_1 + \gamma_2 - 1)).$$

The appropriate computed elements are:

$$\vec{I}_{VL}(\vec{\alpha}, \vec{\gamma}) = (\min(1, 1 - \alpha_2 + \gamma_1), \min(1, 1 - \alpha_1 + \gamma_2));$$

$$\begin{aligned} f_1(\vec{I}_{FK}, \vec{\alpha}, \vec{\beta}) &= \min(1 - \alpha_2, \max(0, \beta_1 - \alpha_2)), \\ f_2(\vec{I}_{FK}, \vec{\alpha}, \vec{\beta}) &= \max(1 - \alpha_1, \min(1, 1 - \alpha_1 + \beta_2)); \end{aligned}$$

$$\begin{aligned} f_1(\vec{I}_{VK}, \vec{\alpha}, \vec{\beta}) &= \max(0, \alpha_2 + \beta_1 - 1), \\ f_2(\vec{I}_{VK}, \vec{\alpha}, \vec{\beta}) &= \max(0, \alpha_1 + \beta_2 - 1). \end{aligned}$$

Extension of the Gödel implication

There are several alternative extensions of the Gödel implication, now we present two of them:

$$\vec{I}_{FG1}(\vec{\alpha},\vec{\gamma})=\begin{cases}(1,0) & \alpha_1 \le \gamma_1,\\ (\gamma_1,0) & \alpha_1>\gamma_1, \alpha_2 \ge \gamma_2,\\ (\gamma_1,\gamma_2) & \alpha_1>\gamma_1, \alpha_2<\gamma_2,\end{cases}$$

$$\vec{I}_{FG2}(\vec{\alpha},\vec{\gamma})=\begin{cases}(1,0) & \alpha_1 \le \gamma_1, \alpha_2 \ge \gamma_2,\\ (\gamma_1,\gamma_2) & \text{otherwise.}\end{cases}$$

The appropriate computed elements are:

$$\vec{I}_{VG1}(\vec{\alpha},\vec{\gamma})=\begin{cases}(1,1) & \alpha_1 \le \gamma_1,\\ (\gamma_1,1) & \alpha_1>\gamma_1, \alpha_2 \ge \gamma_2,\\ (\gamma_1,\gamma_2) & \alpha_1>\gamma_1, \alpha_2<\gamma_2,\end{cases}$$

$$\vec{I}_{VG2}(\vec{\alpha},\vec{\gamma})=\begin{cases}(1,1) & \alpha_1 \le \gamma_1, \alpha_2 \le \gamma_2,\\ (\gamma_1,\gamma_2) & \text{otherwise,}\end{cases}$$

$$f_1(\vec{I}_{FG1},\vec{\alpha},\vec{\beta})=\min(\alpha_1,\beta_1), \qquad f_1(\vec{I}_{FG2},\vec{\alpha},\vec{\beta})=\min(\alpha_1,\beta_1),$$

$$f_2(\vec{I}_{FG1},\vec{\alpha},\vec{\beta})=\begin{cases}1 & \alpha_1\le\beta_1,\\ \max(\alpha_2,\beta_2) & \text{otherwise;}\end{cases} \qquad f_2(\vec{I}_{FG2},\vec{\alpha},\vec{\beta})=\max(\alpha_2,\beta_2),$$

$$f_1(\vec{I}_{VG1},\vec{\alpha},\vec{\beta})=\min(\alpha_1,\beta_1), \qquad f_1(\vec{I}_{VG2},\vec{\alpha},\vec{\beta})=\min(\alpha_1,\beta_1),$$

$$f_2(\vec{I}_{VG1},\vec{\alpha},\vec{\beta})=\begin{cases}0 & \alpha_1\le\beta_1,\\ \min(\alpha_2,\beta_2) & \text{otherwise;}\end{cases} \qquad f_2(\vec{I}_{VG2},\vec{\alpha},\vec{\beta})=\min(\alpha_2,\beta_2).$$

An extremely important question is whether the consequences of the program remain within the scope of intuitionistic or interval-valued fuzzy logic. That is if the levels of the body and the rule satisfy the conditions referring to intuitionistic or interval-valued concepts, does the resulting level of the head also satisfy these conditions?

Unfortunately for implications other than G2, the resulting degrees do not fulfill these conditions in all cases. For example in the case of the Kleene-Dienes and the Lukasiewicz intuitionistic operators the condition of intuitionism satisfies for the levels of the rule-head, only if the sum of the levels of the rule-body is at least as large as the sum of the levels of the rule:

$$f_1(\vec{I}_F, \vec{\alpha}, \vec{\beta}) + f_2(\vec{I}_F, \vec{\alpha}, \vec{\beta}) \leq 1 \text{ if } \alpha_1 + \alpha_2 \geq \beta_1 + \beta_2.$$

That is the solution is inside the scope of IFS, if the level of the rule-body is less “intuitionistic” than the level of the rule.

In the case of the first Gödel operator the condition is simpler:

$$f_1(\vec{I}_F, \vec{\alpha}, \vec{\beta}) + f_2(\vec{I}_F, \vec{\alpha}, \vec{\beta}) \leq 1 \text{ if } \alpha_1 > \beta_1,$$

i.e. the solution is inside the scope of IFS, only if the level of the rule-body is more certain than the level of the rule.

Maybe in practical cases these conditions are satisfied, but the examination of this question may be the subject of further research. For now the next proposition can easily be proved:

Proposition 6 *For* $\vec{\alpha} = (\alpha_1, \alpha_2)$, $\vec{\beta} = (\beta_1, \beta_2)$
if $\alpha_1 + \alpha_2 \leq 1$, $\beta_1 + \beta_2 \leq 1$ *then* $f_1(\vec{I}_{FG2}, \vec{\alpha}, \vec{\beta}) + f_2(\vec{I}_{FG2}, \vec{\alpha}, \vec{\beta}) \leq 1$;
if $\alpha_1 \leq \alpha_2$, $\beta_1 \leq \beta_2$ *then* $f_1(\vec{I}_{VG2}, \vec{\alpha}, \vec{\beta}) \leq f_2(\vec{I}_{VG2}, \vec{\alpha}, \vec{\beta})$.

A further important question is whether the fixed-point algorithm terminates or not, that is whether or not the consequence transformations reach the fixed point in finite steps. As P is finite, the fixed point contains only finite many elements. The only problem may occur with the uncertainty levels of recursive rules. It can be seen that the recursion terminates if $\vec{f}(\vec{I}_{FV}, \vec{\alpha}, \vec{\beta}) \leq_{FV} \vec{\alpha}$ for each $\vec{\alpha} \in L_{FV}$. As

$$\vec{f}(\vec{I}_{FG2}, \vec{\alpha}, \vec{\beta}) = (\min(\alpha_1, \beta_1), \max(\alpha_2, \beta_2)) \leq_F \vec{\alpha}$$

and

$$\vec{f}(\vec{I}_{VG2}, \vec{\alpha}, \vec{\beta}) = (\min(\alpha_1, \beta_1), \min(\alpha_2, \beta_2)) \leq_V \vec{\alpha}$$

G2 satisfies this condition, so:

Proposition 7 *In the case of G2 operator the fixed-point algorithm terminates.*

2.2.2 Bipolar extension of fuzzy Datalog

The above mentioned problem of extended implications other than G2 and the results of certain psychological researches have led to the idea of bipolar fuzzy Datalog. The intuitive meaning of intuitionistic degrees is based on psychological observations, namely on the idea that concepts are more naturally approached through separately envisaging positive and negative instances [14, 18, 19]. Taking a further step, there are differences not only in the instances but also in the way of thinking as well. There is a difference between positive and negative thinking, between deducing positive or negative uncertainty. The idea of bipolar Datalog is based on the previous observation: we use two kinds of ordinary fuzzy implications for positive and negative inference, namely we define a pair of consequence transformations instead of a single one. Since in the original transformations lower bounds are used with degrees of uncertainty, therefore starting from IFS facts, the resulting degrees will be lower bounds of membership and non-membership respectively, instead of the upper bound for non-membership. However, if each non-membership value μ is transformed into membership value $\mu' = 1 - \mu$, then both members of head-level can be inferred similarly. Therefore, two kinds of bipolar evaluations have been defined.

Definition 8 *The bipolar fDATALOG program (bfDATALOG) is a finite set of safe bfDATALOG rules* $(r; (\beta_1, \beta_2); (I_1, I_2))$;

- *in variant "a" the elements of bipolar consequence transformations:* $b\vec{DT}_P = (DT_{P1}, DT_{P2})$ *and* $b\vec{NT}_P = (NT_{P1}, NT_{P2})$ *are the same as* DT_P *and* NT_P *in (1), (2),*
- *in variant "b" in* DT_{P2} *and* NT_{P2} *the level of rule's head is:* $\alpha'_{A2} = \max(0, \min\{\gamma'_2 | I_2(\alpha'_{body2}, \gamma'_2) \geq \beta'_2\})$; *where* $\alpha'_{body2} = \min(\alpha'_{A_1 2}, \ldots, \alpha'_{A_N 2})$

According to the two variants the uncertainty-level functions are:

$$\begin{aligned}
\vec{f}_a &= (f_{a1}, f_{a2}); \quad \vec{f}_b = (f_{b1}, f_{b2}); \\
f_{a1} &= f_{b1} = \min\{\gamma_1 \mid I_1(\alpha_1, \gamma_1) \geq \beta_1\}; \\
f_{a2} &= \min\{\gamma_2 \mid I_2(\alpha_2, \gamma_2) \geq \beta_2\}; \\
f_{b2} &= 1 - \min\{1 - \gamma_2 \mid I_2(1 - \alpha_2, 1 - \gamma_2) \geq 1 - \beta_2\}.
\end{aligned}$$

It is evident, that applying the transformation $\mu'_1 = \mu_1$, $\mu'_2 = 1 - \mu_2$, for each IFS levels of the program, variant "b" can be applied to IVS degrees as well. Contrary to the results of ifDATALOG, the resulting degrees of most variants of bipolar fuzzy Datalog satisfy the conditions referring to IFS and IVS respectively. A simple computation can prove the next proposition:

Proposition 9 *For* $\vec{\alpha} = (\alpha_1, \alpha_2)$*,* $\vec{\beta} = (\beta_1, \beta_2)$ *and for* $(I_1, I_2) = (I_G, I_G)$*;* $(I_1, I_2) = (I_L, I_L)$*;* $(I_1, I_2) = (I_L, I_G)$*;* $(I_1, I_2) = (I_K, I_K)$*;* $(I_1, I_2) = (I_L, I_K)$

$$\text{if } \alpha_1 + \alpha_2 \leq 1,\ \beta_1 + \beta_2 \leq 1 \quad \text{then } f_{a1}(I_1, \vec{\alpha}, \vec{\beta}) + f_{a2}(I_2, \vec{\alpha}, \vec{\beta}) \leq 1$$
$$\text{and } f_{b1}(I_1, \vec{\alpha}, \vec{\beta}) + f_{b2}(I_2, \vec{\alpha}, \vec{\beta}) \leq 1;$$

further on

$$f_{a1}(I_G, \vec{\alpha}, \vec{\beta}) + f_{a2}(I_L, \vec{\alpha}, \vec{\beta}) \leq 1; \qquad f_{a1}(I_G, \vec{\alpha}, \vec{\beta}) + f_{a2}(I_K, \vec{\alpha}, \vec{\beta}) \leq 1;$$
$$f_{a1}(I_K, \vec{\alpha}, \vec{\beta}) + f_{a2}(I_G, \vec{\alpha}, \vec{\beta}) \leq 1; \qquad f_{a1}(I_K, \vec{\alpha}, \vec{\beta}) + f_{a2}(I_L, \vec{\alpha}, \vec{\beta}) \leq 1.$$

Although there are more results for variant "a", it seems that the model realised by variant "b" is more useful.

Because of the construction of bipolar consequence transformations the following proposition is evident:

Proposition 10 *Both variations of bipolar consequence transformations have a least fixed point, which are models of* P *in the following sense:*
a/ for each

$$(A \leftarrow A_1, \ldots, A_n; \vec{\beta}; \vec{I}) \in \mathrm{ground}(P)\ I(\alpha_{body1}, \alpha_{A1}) \geq \beta_1; I(\alpha_{body2}, \alpha_{A2}) \geq \beta_2,$$

b/ for each

$$(A \leftarrow A_1, \ldots, A_n; \vec{\beta}; \vec{I}) \in \mathrm{ground}(P)\ I(\alpha_{body1}, \alpha_{A1}) \geq \beta_1; I(\alpha'_{body2}, \alpha'_{A2}) \geq \beta'_2.$$

Proof. The termination of the consequence transformations based on these three implication operators was proved in the case of fDATALOG, and since this property does not change in bipolar case, the bipolar consequence transformations terminate as well. □

As the bipolar extension of Datalog has no influence on the stratification, therefore the propositions detailed in the case of stratified fDATALOG programs are true in the case of bipolar fuzzy Datalog programs as well:

Proposition 11 *For stratified bfDATALOG program* P*, there is an evaluation sequence, in which* $\mathrm{lfp}(b\vec{N}T_P)$ *is a unique minimal model of* P*.*

Example 12 *Let us consider the next program:*

$$(\mathsf{p}(a,b),(0.6,0.2)). \qquad (\mathsf{p}(a,c),(0.7,0.3)).$$
$$(\mathsf{p}(b,d),(0.5,0.3)). \qquad (\mathsf{p}(d,e),(0.8,0.1)).$$
$$\mathsf{q}(x,y) \leftarrow \mathsf{p}(x,y); \vec{I_1}; (0.75, 0.2).$$
$$\mathsf{q}(x,y) \leftarrow \mathsf{p}(x,z), \mathsf{q}(z,y); \vec{I_2}; (0.7, 0.2).$$

According to it uncertainty levels this program can be evaluated in intuitionistic or bipolar manner. At first let us see the intuitionistic evaluation.

Let $\vec{I}_1 = \vec{I}_2 = \vec{I}_{FG2}(\vec{\alpha},\vec{\gamma}) = \begin{cases} (1,0) & \alpha_1 \le \gamma_1, \alpha_2 \ge \gamma_2 \\ (\gamma_1,\gamma_2) & \textit{otherwise} \end{cases}.$
Then $f_1(\vec{I}_{FG2},\vec{\alpha},\vec{\beta}) = \min(\alpha_1,\beta_1), f_2(\vec{I}_{FG2},\vec{\alpha},\vec{\beta}) = \max(\alpha_2,\beta_2)$.

Without regarding all of the details only three computations will be shown: from the first rule the uncertainty of $\mathsf{q}(a,b)$ *and* $\mathsf{q}(b,d)$ *and from the second one the uncertainty of* $\mathsf{q}(a,d)$.

The uncertainty of $\mathsf{q}(a,b)$ *is:* $(\min(0.6,0.75),\max(0.2,0.2)) = (0.6,0.2)$. *Similarly the uncertainty of* $\mathsf{q}(b,d)$ *is* $(0.5,0.3)$.

In the case of $\mathsf{q}(a,d)$ *its uncertainty can be computed from the appropriate* $(\mathsf{ground}(r);\beta;I)$*, where r is the second rule, that is ground(r) is* $\mathsf{q}(a,d) \leftarrow \mathsf{p}(a,b),\mathsf{q}(b,d)$. *The uncertainty of the rule-body is* $\min_F((0.6,0.2),(0.5,0.3)) = (\min(0.6,0.5),\max(0.2,0.3)) = (0.5,0.3)$. *According to the uncertainty function the level of* $\mathsf{q}(b,d)$ *is:* $(\min(0.5,0.7),\max(0.3,0.2)) = (0.5,0.3)$.

Computing the other atoms, the fixed point is:

$$\{(\mathsf{p}(a,b),(0.6,0.2)),(\mathsf{p}(a,c),(0.7,0.3)),(\mathsf{p}(b,d),(0.5,0.3)),(\mathsf{p}(d,e),(0.8,0.1)),$$
$$(\mathsf{q}(a,b),(0.6,0.2)),(\mathsf{q}(a,c),(0.7,0.3)),(\mathsf{q}(b,d),(0.5,0.3)),(\mathsf{q}(d,e),(0.75,0.2),$$
$$(\mathsf{q}(a,d),(0.5,0.3)),(\mathsf{q}(b,e),(0.5,0.3)),(\mathsf{q}(a,e),(0.5,0.3))\}.$$

Now let the program be evaluated in bipolar manner according to variant "b" and let $\vec{I}_1 = (I_L, I_K), \vec{I}_2 = (I_G, I_G)$*, i.e.*

$$f(I_G,\alpha,\beta) = \min(\alpha,\beta),$$
$$f(I_L,\alpha,\beta) = \max(0,\alpha+\beta-1),$$
$$f(I_K,\alpha,\beta) = \begin{cases} 0 & \alpha+\beta \le 1, \\ \beta & \alpha+\beta > 1. \end{cases}$$

Then the first coordinates of the computed uncertainties are:

$$(q(a,b),(\max(0,0.6+0.75-1)=0.35,\_)),(q(a,c),(0.45,\_)),$$
$$(q(b,d),(0.25,\_)),(q(d,e),(0.55,\_)),$$
$$(q(a,d): \textit{as } \min(0.6,0.25)+0.7\leq 1 \textit{ so } (q(a,d),(0,\_)),$$
$$(q(b,e),(0.7,\_)),(q(a,e),(0.7,\_)).$$

The second coordinates are computed after the appropriate transformation $\alpha_2' = 1-\alpha_2$*. So*

$$(q(a,b),(\_,1-\min(1-0.2,1-0.2)=0.2)),(q(a,c),(\_,0.3)),(q(b,d),(\_,0.3)),$$
$$(q(d,e),(\_,0.2),(q(a,d),(\_,0.3)),(q(b,e),(\_,0.3)),(q(a,e),(\_,0.3)).$$

So the fixed point is:

$$\{(p(a,b),(0.6,0.2)),(p(a,c),(0.7,0.3)),(p(b,d),(0.5,0.3)),$$
$$(p(d,e),(0.8,0.1)),(q(a,b),(0.35,0.2)),(q(a,c),(0.45,0.3)),$$
$$(q(b,d),(0.25,0.3)),(q(d,e),(0.55,0.2),(q(a,d),(0,0.3)),$$
$$(q(b,e),(0.7,0.3)),(q(a,e),(0.7,0.3))\}.$$

2.3 Evaluation of programs

The above examples show that the fixed point-query – that is the bottom-up evaluation – may involve many superfluous calculations, because sometimes we want to give an answer to a concrete question, and we are not interested in the whole sequence. If a goal is specified together with an fDATALOG (or ifDATALOG, bfDATALOG) program, it is enough to consider only the rules and facts necessary to reaching the goal.

Generally by the top down evaluation the goal is evaluated through subqueries. This means that all the possible rules whose head can be unified with the given goal are selected and the atoms of the body are considered as new sub-goals. This procedure continues until the facts are obtained. The evaluation of a fuzzy Datalog or multivalued Datalog program does not terminate by obtaining the facts, because we need to determine the uncertainty level of the goal. The evaluation has a second part; it calculates this level in a bottom-up manner: starting from the leaves of the evaluating graph, going backward to the root, and applying the uncertainty-level functions along the suitable path of this graph, finally we get the uncertainty level of the root.

For a fuzzy Datalog program a goal is a pair $(Q;\alpha_Q)$, where Q is an atom, α_Q is the uncertainty level of the atom. Q may contain variables, and its level may be known or unknown value. An fDATALOG program enlarged with a goal is a fuzzy query.

For a multivalued Datalog program the goal is very similar, except instead of one level, the goal-atom's belonging level is a pair of levels. An ifDATALOG or a bfDATALOG program enlarged with a goal is an intuitionistic-, interval-valued- or bipolar-query. As the evaluating algorithm applies the uncertainty level function independently of its meaning, therefore this algorithm is suitable for all discussed type of fuzzy Datalog or extended Datalog.

Now the evaluating algorithm of the queries will not be detailed because the aim of this paper is to build a knowledge-base system and the evaluation of a knowledgebase differs from the evaluation of a program. The evaluation algorithm of multivalued Datalog programs is discussed in [8].

3 Multivalued knowledge-base

As fuzzy Datalog is a special kind of its each multivalued extension, so further on both fDATALOG and any of above extensions will be called multivalued Datalog (mDATALOG).

3.1 Background knowledge

The facts and rules of an mDATALOG program can be regarded as any kind of knowledge, but sometimes we need some other information in order to get an answer for a query. In this section we give a possible model of background knowledge. Some kind of synomyms will be defined between the potential predicates and between the potential constans of the given problem, so it can be examined in a larger context. More precisely a proximity relation will be defined between predicates and between constants and these structures of proximity will serve as a background knowledge.

Definition 13 *A multivalued proximity on a domain* D *is an IFS or IVS valued relation* $\vec{R}_{FV_D} : D \times D \to [\vec{0}_{FV}, \vec{1}_{FV}]$ *which satisfies the following properties:*

$$\begin{aligned}
\vec{R}_{F_D}(x,y) &= \vec{\mu}_F(x,y) = (\mu_1,\mu_2), \quad \mu_1+\mu_2 \le 1\\
\vec{R}_{V_D}(x,y) &= \vec{\mu}_V(x,y) = (\mu_1,\mu_2), \quad 0 \le \mu_1 \le \mu_2 \le 1\\
\vec{R}_{FV_D}(x,x) &= \vec{1}_{FV} \quad \forall x \in D \quad (reflexivity)\\
\vec{R}_{FV_D}(x,y) &= \vec{R}_{FV_D}(y,x) \quad \forall x,y \in D \quad (symmetry).
\end{aligned}$$

A proximity is similarity if it is transitive, that is

$$\vec{R}_{FV_D}(x,z) \geq \min(\vec{R}_{FV_D}(x,y), \vec{R}_{FV_D}(y,z)) \; \forall x,y,z \in D.$$

In the case of similarity equivalence classifications can be defined over D allowing to develop simpler or more effective algorithms, but now we deal with the more general proximity.

In our model the background knowledge is a set of proximity sets.

Definition 14 *Let* $d \in D$ *any element of domain* D*. The proximity set of* d *is an IFS or IVS subset over* D*:*

$$R_{FV_d} = \{(d_1, \vec{\lambda}_{FV_1}), (d_2, \vec{\lambda}_{FV_2}), \ldots, (d_n, \vec{\lambda}_{FV_n})\},$$

where $d_i \in D$ *and* $\vec{R}_{FV_D}(d, d_i) = \vec{\lambda}_{FV_i}$ *for* $i = 1, \ldots, n$*.*

Based on proximities a background knowledge can be constructed which signify some information about the proximity of terms and predicate symbols.

Definition 15 *Let* G *be any set of ground terms and* S *any set of predicate symbols. Let* RG_{FV} *and* RS_{FV} *be any proximity over* G *and* S *respectively. The background knowledge is:*

$$Bk = \{RG_{FV_t} \mid t \in G\} \cup \{RS_{FV_p} \mid p \in S\}$$

3.2 Computing of uncertainties

So far two steps was made on the way leading to the concept of multivalued knowledge-base: the concept of a multivalued Datalog program and the concept of background knowledge was defined. Now the question is: how can we connect this program with the background knowledge? How can we deduce to the "synonyms"? For example if $(r(a), (0.8, 0.1))$ is an IFS fact and $RS_F(r,s) = (0.6, 0.3), RG_F(a,b) = (0.7, 0.2)$ then what is the uncertainty of $r(b), s(a)$ or $s(b)$?

To solve this problem a new extended uncertainty function will be introduced. According to this function the uncertainty levels of synonyms can be computed from the levels of original fact and from the proximity values of actual predicates and its arguments. It is expectable that in the case of identity the level must be unchanged, but in other cases it is to be less or equal then the original level or then the proximity values. Furthermore we require this function to be monotonically increasing. This function will be ordered to each atom of a program.

Let p be a predicate symbol with n arguments, then p/n is called the functor of the atom characterized by this predicate symbol.

Definition 16 *A* kb*-extended uncertainty function of* p/n *is:*

$$\vec{\varphi}_p(\vec{\alpha},\vec{\lambda},\vec{\lambda_1},\dots,\vec{\lambda_n}) : (\vec{0}_{FV},\vec{1}_{FV}]^{n+2} \to [\vec{0}_{FV},\vec{1}_{FV}]$$

where

$$\begin{aligned}\vec{\varphi}_p(\vec{\alpha},\vec{\lambda},\vec{\lambda_1},\dots,\vec{\lambda_n}) &\le \min(\vec{\alpha},\vec{\lambda},\vec{\lambda_1},\dots,\vec{\lambda_n}),\\ \vec{\varphi}_p(\vec{\alpha},\vec{1}_{FV},\vec{1}_{FV},\dots,\vec{1}_{FV}) &= \vec{\alpha}\end{aligned}$$

and $\vec{\varphi}_p(\vec{\alpha},\vec{\lambda},\vec{\lambda_1},\dots,\vec{\lambda_n})$ *is monoton increasing in each argument.*

It is worth to mention that any triangular norm is suitable for kb-extended uncertainty function, for example

$$\vec{\varphi}_{p_1}(\vec{\alpha},\vec{\lambda},\vec{\lambda_1},\dots,\vec{\lambda_n}) = \min(\vec{\alpha},\vec{\lambda},\vec{\lambda_1},\dots,\vec{\lambda_n}),$$
$$\vec{\varphi}_{p_2}(\vec{\alpha},\vec{\lambda},\vec{\lambda_1},\dots,\vec{\lambda_n}) = \min(\vec{\alpha},\vec{\lambda},\vec{\lambda_1}\cdots\vec{\lambda_n}),$$

where the product is:

$$(\mu_1,\mu_2)\cdot(\lambda_1,\lambda_2) = (\mu_1\cdot\lambda_1,\mu_2\cdot\lambda_2),$$

are kb-extended uncertainty functions, but

$$\vec{\varphi}_{p_3}(\vec{\alpha},\vec{\lambda},\vec{\lambda_1},\dots,\vec{\lambda_n}) = \vec{\alpha}\cdot\vec{\lambda}\cdot\vec{\lambda_1}\cdots\vec{\lambda_n}$$

is a kb-extended uncertainty function only in the interval valued case.

Example 17 *Let* $(r(a),(0.8,0.1))$ *be an IFS fact and* $\mathsf{RS}_F(r,s)=(0.6,0.3)$, $\mathsf{RG}_F(a,b)=(0.7,0.2)$ *and* $\vec{\varphi}_r(\vec{\alpha},\vec{\lambda},\vec{\lambda_1}) = \min(\vec{\alpha},\vec{\lambda},\vec{\lambda_1})$ *then the uncertainty levels of* $r(b), s(a)$ *and* $s(b)$ *are:*

$$\begin{aligned}(r(b),(\min(0.8,1,0.7),\max(0.1,0,0.2))) &= (r(b),(0.7,0.2)),\\ (s(a),(\min(0.8,0.6,1),\max(0.1,0.3,0))) &= (s(a),(0.6,0.3)),\\ (s(b),(\min(0.8,0.6,0.7),\max(0.1,0.3,0.2))) &= (s(b),(0.6,0.3)).\end{aligned}$$

We have to order kb-extended uncertainty functions to each predicate of the program. The set of these functions will be the function-set of the program.

Definition 18 *Let* P *be a multivalued Datalog program, and* F_P *be the set of the program's functors. The function-set of* P *is:*

$$\Phi_P = \{\vec{\varphi}_p(\vec{\alpha},\vec{\lambda},\vec{\lambda_1},\dots,\vec{\lambda_n}) \mid \forall\, p/n \in F_P.\}$$

3.3 Connecting algorithm

Let P be a multivalued Datalog program, Bk be any background knowledge and Φ_{P} be the function-set of P. The deducing mechanism consist of two alternating part: starting from the fact we determine their "synonyms", then applying the suitable rules another facts are derived, then their "synonyms" are derived and again the rules are applied, etc. To define it in a precise manner the concept of modified consecution transformation will be introduced.

The original consequence transformation is defined over the set of all multivalued sets of P's Herbrand base, that is over $\mathsf{F}(\mathsf{B}_{\mathsf{P}})$. To define the modified transformation's domain, let us extend P's Herbrand universe with all possible ground terms occurring in background knowledge: this way, we obtain the modified Herbrand universe $\mathsf{modH}_{\mathsf{P}}$. Let the modified Herbrand base $\mathsf{modB}_{\mathsf{P}}$ be the set of all possible ground atoms whose predicate symbols occur in $\mathsf{P}\cup\mathsf{Bk}$ and whose arguments are elements of $\mathsf{modH}_{\mathsf{P}}$. This leads to

Definition 19 *The modified consequence transformation*

$$\mathsf{modNT}_{\mathsf{P}} : \mathsf{FV}(\mathsf{modB}_{\mathsf{P}}) \to \mathsf{FV}(\mathsf{modB}_{\mathsf{P}})$$

is defined as

$$\begin{aligned}\mathsf{modNT}_{\mathsf{P}}(X) = \{(q(s_1, \dots, s_n), \vec{\varphi}_p(\vec{\alpha_p}, \vec{\lambda}_q, \vec{\lambda}_{s_1}, \dots, \vec{\lambda}_{s_n}) \mid \\ (q, \vec{\lambda}_q) \in \mathsf{RS}_{\mathsf{FV}_p}; \\ (s_i, \vec{\lambda}_{s_i}) \in \mathsf{RG}_{t_i},\ 1 \le i \le n\} \cup X,\end{aligned}$$

where

$$\begin{aligned}&(p(t_1, \dots, t_n) \leftarrow A_1, \dots, A_k; \vec{I}; \vec{\beta}) \in \textit{ground}(\mathsf{P}),\\ &(\ |A_i|, \alpha_{A_i}) \in X,\ 1 \le i \le k, \quad (|A_i| \textit{ is the kernel of } A_i)\end{aligned}$$

and $\vec{\alpha_p}$ is computed according to the actual extension of (1).

It is obvious that this transformation is inflationary over $\mathsf{FV}(\mathsf{modB}_{\mathsf{P}})$ and it is monotone if P is positive.
(A transformation T over a lattice L is inflationary if $X \le \mathsf{T}(X)\ \forall X \in \mathsf{L}$. T is monotone if $\mathsf{T}(X) \le \mathsf{T}(Y)$ if $X \le Y$.)

According to [15] an inflationary transformation over a complete lattice has a fixed point moreover a monotone transformation has a least fixed point, so

Proposition 20 *The modified consequence transformation* $\mathsf{modNT}_{\mathsf{P}}$ *has a fixed point. If* P *is positive, then this is the least fixed point.*

It can be shown that this fixed point is a model of P, but $\mathrm{lfp}(\mathsf{NT_P}) \subseteq \mathrm{lfp}(\mathrm{mod}\mathsf{NT_P})$, so it is not a minimal model.

As the modifying of original transformation that is the modifying algorithm has no effect on the order of rules, therefore it does not change the stratification. Therefore we can state

Proposition 21 *In the case of stratified program* P*,* $\mathrm{mod}\mathsf{NT_P}$ *has least fixed point as well.*

Now we have all components together to define the concept of a multivalued knowledge-base. But before doing it, it is worth mentioning that the above modified consequence transformation is not the unique way to connect the background knowledge with the deduction mechanism, there could be other possibilities as well.

Definition 22 *A multivalued knowledge-base (*$\mathrm{m}\mathsf{KB}$*) is a quadruple*

$$\mathrm{m}\mathsf{KB} = (\mathsf{Bk}, \mathsf{P}, \Phi_{\mathsf{P}}, \mathsf{cA}),$$

where Bk *is a background knowledge,* P *is a multivalued Datalog program,* Φ_{P} *is a function-set of* P *and* cA *is any connecting algorithm.*

The result of the connected and evaluated program is called the consequence of the knowledge-base, denoted by

$$\mathsf{C}(\mathsf{Bk}, \mathsf{P}, \Phi_{\mathsf{P}}, \mathsf{cA}).$$

So in our case $\mathsf{C}(\mathsf{Bk}, \mathsf{P}, \Phi_{\mathsf{P}}, \mathsf{cA}) = \mathrm{lfp}(\mathrm{mod}\mathsf{NT_P})$.

Example 23 *Let the IVS valued* $\mathrm{m}\mathsf{DATALOG}$ *program and the background knowledge be as follows*

$$\begin{aligned}&\mathrm{lo}(x, y) \ \leftarrow \ \mathrm{gc}(y), \mathrm{mu}(x); (0.7, 0.9); \vec{\mathrm{I}}_{\mathsf{VG}}.\\&(\mathrm{fv}(V), (0.85, 0.9).\\&(\mathrm{mf}(M), (0.7, 0.8).\end{aligned}$$

	B	V	M
B	$(1,1)$	$(0.8, 0.9)$	
V	$(0.8, 0.9)$	$(1,1)$	
M			$(1,1)$

	lo	*li*	*gc*	*fv*	*mu*	*mf*
lo	$(1,1)$	$(0.7,0.9)$				
li	$(0.7,0.9)$	$(1,1)$				
gc			$(1,1)$	$(0.8,0.9)$		
fv			$(0.8,0.9)$	$(1,1)$		
mu					$(1,1)$	$(0.6,0.7)$
mf					$(0.6,0.7)$	$(1,1)$

According to the connecting algorithm, it is enough to consider only the extended uncertainty functions of head-predicates. Let these functions be as follows:

$$\begin{aligned}\vec{\varphi}_{lo}(\vec{\alpha},\vec{\lambda},\vec{\lambda_1},\vec{\lambda_2}) &:= \min(\vec{\alpha},\vec{\lambda},\vec{\lambda_1}\cdot\vec{\lambda_2}),\\ \vec{\varphi}_{fv}(\vec{\alpha},\vec{\lambda},\vec{\lambda_1}) &:= \min(\vec{\alpha},\vec{\lambda},\vec{\lambda_1}),\\ \vec{\varphi}_{mf}(\vec{\alpha},\vec{\lambda},\vec{\lambda_1}) &:= \vec{\alpha}\cdot\vec{\lambda}\cdot\vec{\lambda_1}.\end{aligned}$$

The modified consequence transformation takes shape in the following steps:

$$X_0 = \{(fv(V),(0.85,0.9)),(mf(M),(0.7,0.8))\}$$

$$\Downarrow \quad \textit{(according to the proximity)}$$

$$\begin{gathered}X_1 = modNT_P(X_0) = X_0\cup\\ \{(gc(V),\vec{\varphi}_{fv}((0.85,0.9),(0.8,0.9),(1,1)) =\\ (min(0.85,0.8,1),min(0.9,0.9,1)) = (0.8,0.9)),\\ (fv(B),\vec{\varphi}_{fv}((0.85,0.9),(1,1),(0.8,0.9)) = (0.8,0.9)),\\ (gc(B),\vec{\varphi}_{fv}((0.85,0.9),(0.8,0.9),(0.8,0.9)) = (0.8,0.9)),\\ (mu(M),\vec{\varphi}_{mf}((0.7,0.8),(0.6,0.7),(1,1)) =\\ (0.7\cdot 0.6\cdot 1, 0.8\cdot 0.7\cdot 1) = (0.42,0.56))\}\end{gathered}$$

$$\Downarrow \quad \textit{(applying the rules)}$$

$$\begin{aligned}lo(M,V) &\leftarrow gc(V),mu(M);(0.7,0.9);\vec{I}_{VG}.\\ lo(M,B) &\leftarrow gc(B),mu(M);(0.7,0.9);\vec{I}_{VG}.\end{aligned}$$

$$\textit{here}: f_V(\vec{I}_{VG},\vec{\alpha},\vec{\beta}) = \min(\vec{\alpha}_{body},\vec{\beta}),\ \textit{so}$$

$$\begin{gathered}X_2 = modNT_P(X_1) = X_1\cup\\ \{(lo(M,V),(0.42,0.56)),(lo(M,V),(0.42,0.56))\}\end{gathered}$$

$$\Downarrow \quad \textit{(according to the proximity)}$$

$$\begin{gathered}X_3 = \mathsf{modNT_P}(X_2) = X_2 \cup \\ \{(\mathsf{li}(M,V),(\min(0.42,0.7,1\cdot 1),\min(0.56,0.9,1\cdot 1))), \\ (\mathsf{li}(M,B),(\min(0.42,0.7,1\cdot 1),\min(0.56,0.9,1\cdot 1)))\}\cup \\ \{(\mathsf{li}(M,V),(\min(0.42,0.7,0.64),\min(0.56,0.9,0.81))), \\ (\mathsf{li}(M,B),(\min(0.42,0.7,0.64),\min(0.56,0.9,0.81)))\}\end{gathered}$$

X_3 *is a fixed point, so the consequence of the knowledge-base is:*

$$\begin{aligned}&C(Bk,P,\Phi_P,cA) = \\ &\{(\mathsf{fv}(V),(0.85,0.9)),(\mathsf{mf}(M),(0.7,0.8)), \\ &(\mathsf{gc}(V),(0.8,0.9)),(\mathsf{fv}(B),(0.8,0.9)), \\ &(\mathsf{gc}(B),(0.8,0.9)),(\mathsf{mu}(M),(0.42,0.56)), \\ &(\mathsf{lo}(M,V),(0.42,0.56)),(\mathsf{lo}(M,V),(0.42,0.56)) \\ &(\mathsf{li}(M,V),(0.42,0.56)),(\mathsf{li}(M,B),(0.42,0.56))\}\end{aligned}$$

To illustrate our discussion with some realistic content, in the above example the knowledge-base could have the following interpretation. Let us suppose that music listeners “generally” (level between 0.7, 0.9) are fond of the greatest composers. Assume furthermore that Mary is a “rather devoted” (level between 0.7, 0.8) fan of classical music (mf), and Vivaldi is “generally accepted” (level between 0.85, 0.9) as a “great composer”. It is also widely accepted that the music of Vivaldi and Bach are fairly “similar”, being related in overall structure and style. On the basis of the above information, how strongly state that Mary likes Bach? To continue with this idea, next we can assume that an internet agent wants to suggest a good CD for Mary, based on her interests revealed through her actions at an internet site. A multivalued knowledge-base could help the agent to get a good answer. As some of the readers may well know, similar mechanisms – but possibly based on entirely different modelling paradigms – are in place in prominent websites such as Amazon and others.

4 Evaluation strategies

As the above example shows (especially in the case of enlarging the program with other facts and rules), the fixed point-query may involve many superfluous calculations, because sometimes we want to give an answer for a concrete question, and we are not interested in the whole sequence.

If a goal (query) is specified together with the multivalued knowledge-base, then it is enough to consider only the rules and facts being necessary to reach the goal. In this section we deal with a possible evaluation of knowledge-base, which is a combination of top-down and bottom-up evaluation.

A goal is a pair $(q(t_1, t_2, \dots t_n), \vec{\alpha})$, where $(q(t_1, t_2, \dots t_n)$ is an atom, $\vec{\alpha}$ is the fuzzy, the intuitionistic, the interval-valued or the bipolar level of the atom. q may contain variables, and its levels may be known or unknown values.

According to the top down evaluation a goal is evaluated through sub-queries. This means that there are selected all possible rules, whose head can be unified with the given goal, and the atoms of the body are considered as new sub-goals. This procedure continues until the facts are obtained.

The top-down evaluation of a multivalued Datalog program does not terminate by obtaining the facts, because we need to determine the uncertainty levels of the goal. The algorithm given in [5, 8] calculates this level in a bottom-up manner: starting from the leaves of the evaluating graph, going backward to the root, and applying the uncertainty-level functions along the suitable path of this graph, finally we get the uncertainty level of the root.

In the case of knowledge-base, we rely on the bottom-up evaluation, but the selection of required starting facts takes place in a top-down fashion. Since only the required starting facts are sought, in the top-down part of the evaluation there is no need for the uncertainty levels. Hence, we search only among the ordinary facts and rules. To do this, we need the concept of substitution and unification which are given for example in [8, 15, 21], etc.

But now sometime we also need other kinds of substitutions: to substitute some predicate p or term t for their proximity sets R_p and R_t, and to substitute some proximity sets for their members.

Next, for the sake of simpler terminology, we mean by goal, rules and facts these concepts without uncertainty levels. An AND/OR tree arises during the evaluation, this is the searching tree. Its root is the goal; its leaves are either YES or NO. The parent nodes of YES are the required starting facts. This tree is build up by alternating proximity-based and rule-based unification.

The proximity-based unification unifies the predicate symbols of sub-goals by the members of its proximity set, except the first and last unification. The first proximity-based unification unifies the ground terms of the goal with their proximity sets, and the last one unifies the proximity sets among the parameters of resulting facts with their members.

The rule-based unification unifies the sub-goals with the head of suitable rules, and continues the evaluating by the bodies of these rules. During this unification the proximity sets of terms are considered as ordinary constants, and a constant can be unify with its proximity set. The searching graph according to its depth is build up in the following way:

If the goal is on depth 0, then every successor of any node on depth $3k+2(k = 0, 1, \ldots)$ is in AND connection, the others are in OR connection. In detail:

The successors of a goal $q(t_1, t_2, \ldots t_n)$ will be all possible $q'(t'_1, t'_2, \ldots, t'_n)$, where $q' \in R_q$; $t'_i = t_i$ if t_i is some variable and $t'_i = R_{t_i}$ if t_i is a ground term.

If the atom $p(t_1, t_2, \ldots t_n)$ is in depth $3k(k = 1, 2, \ldots)$, then the successor nodes be all possible $p'(t_1, t_2, \ldots t_n)$, where $p' \in R_p$.

If the atom L is in depth $3k + 1(k = 1, 2, \ldots)$, then the successor nodes will be the bodies of suitable unified rules, or the unified facts, if L is unifiable with any fact of the program, or NO, if there is not any unifiable rule or fact. That is, if the head of rule $M \leftarrow M_1, \ldots, M_n, (n > 0)$ is unifiable with L, then the successor of L be $M_1\theta, \ldots, M_n\theta$, where θ is the most general unification of L and M.

If $n = 0$, that is in the program there is any fact with the predicate symbol of L, then the successors be the unified facts. If $L = p(t_1, t_2, \ldots, t_n)$ and in the program there is any fact with predicate symbol p, then the successor nodes be all possible $p(t'_1, t'_2, \ldots, t'_n)$, where $t'_i \in R_{t_i}$ if $t_i = R_{t_i}$ or $t'_i = t_i\theta$, if t_i is a variable, and θ is a suitable unification.

According the previous paragraph, there are three kinds of nodes in depth $3k + 2(k = 1, 2, \ldots)$: a unified body of a rule; a unified fact with ordinary ground term arguments; or the symbol NO.

In the first case the successors are the members of the body. They are in AND connection, which is not important in our context, but maybe important for possible future development. If the body has only one literal, then the length of evaluating path would be reduced to one, but it would "damage" the view of homogeneous treatment. In the second case the successors are the symbol YES or NO, depending on whether the unified fact is among the ground atoms of the program. The NO-node has not successor.

From the construction of searching graph, we conclude

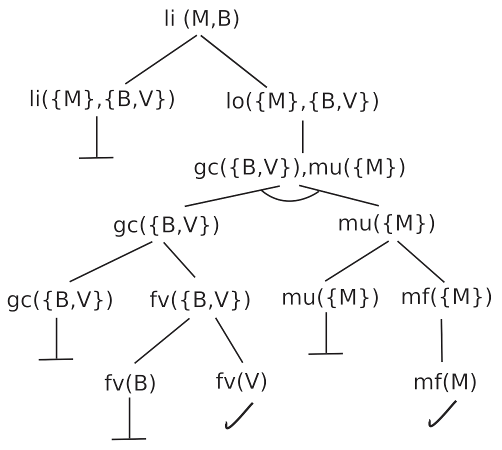
Proposition 24 *Let X_0 be the set of ground facts being in parent-nodes of symbols YES. Starting from X_0, the fixed point of mNT_P contains the answer for the query.*

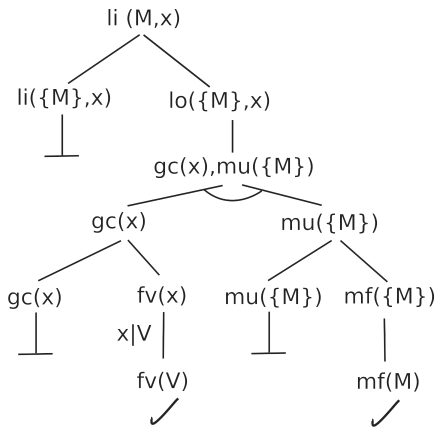
From the viewpoint of the query, this fixed point may contain more superfluous ground atom, but generally it is smaller then the consequence of knowledge-base. More reduction of the number of superfluous resulting facts is the work of a possible further development.

Example 25 *Let us consider the knowledge-base of example 23. (Now it is enough to consider only the program and the background knowledge.)*
Let the goal be:

a/ li(M,B).
b/ li(M,x), where x is a variable.

Then the searching graphs are:





According to the above construction, the searching algorithm is the following alternation of proximity-based and rule-based unification.

Algorithm

```
procedure evaluation(g($\underline{t}$))              /* g($\underline{t}$) is the goal */
   Heads := {the heads of the program's rules}
   Facts := {the facts of the program}
   Resulting_Facts := $\emptyset$              /* the set of resulting starting facts */
   for all t $\in \underline{t}$ do
      if is_variable(t) then s := t
            else s := St                  /* St is the proximity set of t */
      end_if
   end_for

   Nodes := {g($\underline{s}$)}                /* Nodes is the set of evaluable nodes,
                                           $\underline{s}$ is the vector of elements s
                                           in the original order */
```

```
  New_nodes := $\emptyset$                 /* the successor nodes of Nodes */
  while not_empty(Nodes) do
    $p(\underline{t})$ := element(Nodes)
    Spnodes := $\emptyset$                 /* the successor nodes of $p(\underline{t})$ */
    proximity_evaluation($p(\underline{t})$,Spnodes)
    New_nodes := New_nodes $\cup$ Spnodes
    Nodes := Nodes – $\{p(\underline{t})\}$
  end_while
  Nodes := New_nodes
  New_nodes := $\emptyset$
  while not_empty(Nodes) do
    $p(\underline{t})$ := element(Nodes)
    Spnodes := $\emptyset$                 /* the successor nodes of $p(\underline{t})$ */
    rule_evaluation($p(\underline{t})$,Spnodes)
    New_nodes := New_nodes $\cup$ Spnodes
    Nodes := Nodes – $\{p(\underline{t})\}$
  end_while
  return Resulting_Facts
end_procedure

procedure proximity_evaluation($p(\underline{t})$,Spnodes)
  for all $q \in Sp$ do                /* $Sp$ is the proximity set of $p$ */
    Spnodes := Spnodes $\cup$ $\{q(\underline{t})\}$
  end_for
end_procedure

procedure rule_evaluation($p(\underline{t})$,Spnodes)
  for all $p(\underline{v}) \in$ Heads do
    if is_unifiable($p(\underline{t})$,$p(\underline{v})$) then
    Spnodes := Spnodes $\cup$
                        {unified predicates of the body belonging to $p(\underline{v}\theta)$}
                              /* $\theta$ is the suitable unifier */
    end_if
  end_for
  for all $p(\underline{v}) \in$ Facts do
    if is_unifiable($p(\underline{t})$,$p(\underline{v})$) then
    for all $St \in \underline{v}\theta$ do          /* $\theta$ is the suitable unifier */
      if is_variable($St$) then
      $t := St\tau$                  /* $\tau$ is the suitable unifier */
```

```
        else if is_proximity_set($\mathtt{St}$) then
        $\mathtt{t}$ := element($\mathtt{St}$)
        end_if
      end_for
      end_if
      for all possible $\underline{\mathtt{t}}$ do
                    /* $\underline{\mathtt{t}}$ is the vector of elements $\mathtt{t}$ in the right order */
        if $\mathtt{p}(\underline{\mathtt{t}}) \in$ Facts then
          Resulting_Facts := Resulting_Facts $\cup$ $\{\mathtt{p}(\underline{\mathtt{t}})\}$
        end_if
      end_for
    end_for
  end_procedure
```

This algorithm can be applied for stratified multivalued Datalog too by determining the successor of a rule-body without negation.

5 Conclusions

In this paper we have presented a model of handling uncertain information by defining the multivalued knowledge-base as a quadruple of background knowledge, a deduction mechanism, a decoding set and a modifying algorithm which connects the background knowledge to the deduction mechanism. We also have presented a possible evaluation strategy. To improve upon this strategy and/or the modifying algorithm and/or the structure of background knowledge will be the subject of further investigations. An efficient multivalued knowledge base could be the basis of decisions based on uncertain information, or would be a possible method for handling argumentation or negotiation of agents.

References

[1] Á. Achs, A. Kiss, Fixpoint query in fuzzy Datalog, *Ann. Univ. Sci., Budapest., Sect. Comput.* **15** (1995) 223–231. ⇒51, 54, 55

[2] Á. Achs, A. Kiss, Fuzzy extension of Datalog, *Acta Cybernet.* (Szeged), **12,** 2 (1995) 153–166. ⇒51, 55

[3] Á. Achs, Creation and evaluation of fuzzy knowledge-base, *J. UCS*, **12,** 9 (2006) 1087–1103. ⇒52

[4] Á. Achs, Twofold extensions of fuzzy datalog, *7th International Workshop on Fuzzy Logic and Applications, WILF 2007*, Camogli, Italy, 2007, pp. 298–306. ⇒52

[5] Á. Achs, Computed answer from uncertain knowledge: A model for handling uncertain information, *Comput. Inform.*, **26,** 1 (2007) 298–306. ⇒ 73

[6] Á. Achs, DATALOG-based uncertainty-handling, *The Twelfth IASTED International Conference on Artificial Intelligence and Soft Computing*, Mallorca, Spain, 2008, pp. 44–49. ⇒52

[7] Á. Achs, Multivalued Knowledge-Base based on multivalued Datalog, *Proc. World Academy of Science, Engineering and Technology*, **54** (2009) 160–165. ⇒52

[8] Á. Achs, Fuzziness and intuitionism in database theory, in: *Intuitionistic Fuzzy Sets: Recent Advances*, to appear ⇒66, 73

[9] K. Atanassov, Intuitionistic fuzzy sets, *VII ITKR's Session, Sofia (deposed in Central Science-Technical Library of Bulgarian Academy of Science, 1697/84)*, 1983. ⇒56

[10] K. Atanassov, G. Gargov, Interval-valued intuitionistic fuzzy sets, *Fuzzy Sets and Systems*, **31,** 3 (1989) 343–349. ⇒56

[11] K. Atanassov, *Intuitionistic fuzzy sets*, Springer-Verlag, Heidelberg, 1999. ⇒56

[12] K. Atanassov, Intuitionistic fuzzy implications and Modus Ponens, *Notes on Intuitionistic Fuzzy Sets*, **11** (2005) 1–5. ⇒58

[13] K. Atanassov, On some intuitionistic fuzzy implications, *C. R. Acad. Bulgare Sci.,* **59** (2006) 19–24. ⇒58

[14] J. T. Cacioppo, W. L. Gardner, G. G. Berntson, Beyond bipolar conceptualization and measures: the case of attitudes and evaluative spaces, *Personality and Social Psychol. Rev.*, **1** (1997) 3–25. ⇒62

[15] S. Ceri, G. Gottlob, L. Tanca, *Logic Programming and Databases*, Springer–Verlag, Berlin, 1990. ⇒53, 57, 69, 73

[16] C. Cornelis, G. Deschrijver, E. E. Kerre, Implication in intuitionistic fuzzy and interval-valued fuzzy set theory: construction, classification, application, *Internat. J. Approx. Reason.*, **35** (2004) 55–95. ⇒57, 58

[17] D. Dubois, H. Prade, Fuzzy sets in approximate reasoning, Part 1: Inference with possibility distributions, *Fuzzy Sets and Systems*, **40** (1991) 143–202. ⇒55

[18] D. Dubois, P. Hajek, H. Prade, Knowledge-Driven versus Data-Driven Logics, *J. Log. Lang. Inf.*, **9** (2000) 65–89. ⇒62

[19] D. Dubois, S. Gottwald, P. Hajek, J. Kacprzyk, H. Prade, Terminological difficulties in fuzzy set theory – The case of "Intuitionistic Fuzzy Sets", *Fuzzy Sets and Systems*, **15** (2005) 485–491. ⇒62

[20] J. W. Lloyd, *Foundations of Logic Programming*, Springer–Verlag, Berlin, 1990. ⇒53

[21] J. D. Ullman, *Principles of Database and Knowledge-base Systems*, Computer Science Press, Rockville, 1988. ⇒53, 73

Received: October 17, 2009 • Revised: February 22, 2010

Acta Univ. Sapientiae, Informatica, **2,** 1 (2010) 80–89

FIFO anomaly is unbounded

Péter Fornai
Eötvös Loránd University
Department of Computer Algebra
H-1117, Budapest, Hungary
Pázmány sétány 1/C
email: `peter.fornai@inf.elte.hu`

Antal Iványi
Eötvös Loránd University
Department of Computer Algebra
H-1117, Budapest, Hungary
Pázmány sétány 1/C
email: `tony@inf.elte.hu`

Abstract. Virtual memory of computers is usually implemented by demand paging. For some page replacement algorithms the number of page faults may increase as the number of page frames increases. Belady, Nelson and Shedler [5] constructed reference strings for which page replacement algorithm FIFO [10, 13, 36, 40] produces near twice more page faults in a larger memory than in a smaller one. They formulated the conjecture that 2 is a general bound. We prove that this ratio can be arbitrarily large.

1 Introduction

Let us consider a computer with two-level virtual memory [12, 26, 38]. Physical and secondary memory are divided into equal-sized blocks called frames. The processes reside in secondary memory and are broken into fixed-sized blocks called pages (the sizes of the frames and pages are the same). The run of the processes is described by reference strings.

According to the pure demand paging the execution of a process starts with empty physical memory (in the following simply *memory*) and a page is brought into the memory only when it is necessary. If a new page is demanded when the memory is full, the page replacement algorithm [38] chooses a page being in memory to be replaced by the demanded one.

Computing Classification System 1998: G.2.2
Mathematics Subject Classification 2010: 68M20
Key words and phrases: memory management, virtual memory, page replacement, FIFO, Belady anomaly

We use the formal model proposed by P. Denning [12]. Let m, M, n, and p be positive integers ($1 \leq m \leq M \leq n < \infty$), k a nonnegative integer, $A = \{a_1, a_2, \ldots, a_n\}$ a finite alphabet, A^k the set of k-length words and A^* be the set of finite words over A, where n is the number of pages, m is the number of frames in the small and M is the number of frames in the large memory, A is the set of pages.

Page replacement algorithms [38] are handled as automata having a memory of size m (or M), set of input signals $X = A$, set of output signals $Y = A \cup \{\epsilon\}$ and processing the sequence of input symbols $R = (r_1, r_2, \ldots, r_p)$ or $R = (r_1, r_2, \ \ldots \ )$. Memory state S_t ($t = 1,\ 2,\ \ldots$) is defined as a set of symbols stored in memory at the given moment t (after the processing of reference r_t). Any page replacement algorithm starts with an empty memory that is $S_0 = \{\}$. A concrete page replacement algorithm P is defined as a triple $P = (Q_P, q_0, g_P)$, where Q_P is the set of control states, $q_0 \in Q_P$ is the initial control state and g_P is the transition function determining the new memory state S', new control state q' and output symbol y using the old memory state S, old control state q and input symbol x.

We consider three page replacement algorithms: FIFO (First In First Out) [10, 13, 36, 40], LRU (Least Recently Used) [8, 10, 19, 26, 37] and MIN (Minimal) [1, 2, 5, 28, 29, 32, 34].

FIFO is defined by $q_0 = ()$ and

$$g_{\text{FIFO}}(S, q, x) = \begin{cases} (S, q, \epsilon), & \text{if } x \in S, \\ (S \cup \{x\}, q', \epsilon), & \text{if } x \notin S \text{ and } |S| < m, \\ (S \cap \{y_1\} \cup \{x\}, q", y_1), & \text{if } x \notin S \text{ and } |S| = m, \end{cases}$$

where $q = (y_1, y_2, \ldots, y_k)$, $q' = (y_1, y_2, \ldots, y_k, x)$, and $q'' = (y_2, y_3, \ldots, y_m, x)$.

LRU replaces the least recently used page of the memory, MIN replaces the page having maximal distance up to its next occurrence in the reference string.

The number of page faults (number of changes of the memory states) is denoted by $f_P(R, m)$. If $M > m$ and $f_P(R, M) > f_P(R, m)$, then we have **an anomaly** and the ratio $f_P(R, M)/f_P(R, m)$ is called **anomaly ratio**. The first anomaly was observed in the practice [4, 5, 34] and is called Belady's anomaly [28, 35]. Later other examples of unexpected events were described [6, 9, 14, 17, 18, 20, 21, 22, 23, 25, 27, 31, 33].

Stack algorithms [28, 30, 39] do not suffer from anomaly.

Mihnovskiy and Shor [24] and Gecsei et al. [15] proved independently that MIN guarantees the minimal number of page faults at a fixed size of the memory.

Arató [3] and Benczúr et al. [7] proved that LRU is optimal in statistical sense among the algorithms having no concrete information on the continuation of the reference string.

A possible measure of the efficiency $E_P(R, m)$ of P is called **paging rate** and is defined by $f_P(R, m)/p$ for finite $R = (r_1, r_2, \ldots, r_p)$ and by

$$E_P(R, m) = \liminf_{k \to \infty} \frac{f_P(R_k, m)}{k}$$

for an infinite reference string R, where $R_k = (r_1, r_2, \ldots, r_k)$.

Let $1 \leq m < n$ and $C = (1, 2, \ldots, n)^*$ be an infinite cyclical reference string. Then the paging rates of FIFO and MIN are given in [16] $E_{FIFO}(C, m) = 1$ and $E_{MIN}(C, m) = (n - m)/(n - 1)$.

2 Classical example and results

If we execute the reference string R = (1,2,3,4,1,2,5,1,2,3,4,5) [5] having 3 frames in the memory (for the simplicity natural numbers are used to denote the pages), then FIFO results the control state sequence $q_0 = ()$, $q_1 = (1)$, $q_2 = (1, 2)$, $q_3 = (1, 2, 3)$, $q_4 = (2, 3, 4)$, $q_5 = (3, 4, 1)$, $q_6 = (4, 1, 2)$, $q_7 = (1, 2, 5)$, $q_8 = (1, 2, 5)$, $q_9 = (1, 2, 5)$, $q_{10} = (2, 5, 3)$, $q_{11} = (5, 3, 4)$, $q_{12} = (5, 3, 4)$ and 9 page faults occurred.

The execution of R using 4 frames will end in the control state (2,3,4,5) and 10 page faults occurred, so $f_{FIFO}(R, M)/f_{FIFO}(R, m) = 10/9$.

Belady, Nelson and Shedler [5] gave a necessary and sufficient condition for the existence of anomaly and constructed reference strings resulting anomaly ratio which is close to 2.

Theorem 1 [5] *Let m and M be positive integers ($1 \leq m < M$) and let A be a finite alphabet of cardinality at least $M + 1$. Then there exists a reference string $R \in A^*$ which produces an anomaly if and only if $M < 2m - 1$.*

Theorem 2 [5] *If m and M are positive integers satisfying the relations $m < M < 2m - 1$, then for sufficiently large p there exists a reference string $R = (r_1, r_2, \ldots, r_p)$ resulting anomaly ratio $f_{FIFO}(R, M)/f_{FIFO}(R, m)$ arbitrarily close to 2.*

Belady, Nelson and Shedler [5] formulated the following conjecture.

Conjecture [5] *If* m *and* M *are positive integers and* R *is a reference string, then* $f_{\mathrm{FIFO}}(R, M)/f_{\mathrm{FIFO}}(R, m) \leq 2$.

3 Disprove of the conjecture

Let $m = 5,\ M = 6,\ n = 7,\ k \geq 1$ and $R = UV^k$, where U = (1,2,3,4,5,6,7,1,2, 4,5,6,7,3,1,2,4,5,7,3,6,2,1,4,7,3,6,2,5,7,3,6,2,5) and $V = (1, 2, 3, 4, 5, 6, 7)^3$.

Now execution of U using m frames results the control state (7,3,6,2,5) and 29 page faults. After this each execution of V results 7 new faults and the same control state. Execution of U using M frames results the control state (2,3,4,5,6,7) and 14 page faults. After this each execution of V results 21 new page fault and the same control state.

Choosing $k = 7$ we get an anomaly ratio $(14+7\times21)/(29+7\times7) = 161/78 >$ 2. If the number of repetitions of V grows, then the anomaly ratio tends to 3.

4 Anomaly is unbounded

We start with a well-known assertion of the number theory. Let $[a]_n$ be the equivalence class modulo n containing the number a and let $\mathbf{Z}_n$ be the set of all equivalence classes modulo n:

$$[a]_n = \{a + kn : \ k \in \mathbf{Z}\} \text{ and } \mathbf{Z}_n = \{[a]_n : 0 \leq a \leq n - 1\}.$$

Lemma 3 [16] *If the set* $c_1, c_2, \ldots, c_n$ *is a complete residue system* $\mathbf{Z}_n \pmod{n}$, *the greatest common divisor of* a *and* n *equals 1, then the set* $\{ac_1 + d, ac_2 + d, \ldots, ac_n + d\}$ *is also a complete residue system* $\pmod{n}$ *for arbitrary integer* d.

Let $a \pmod{b}$ denote the smallest positive representative of the residue class of numbers congruent with $a \pmod{b}$.

Lemma 4 *Let* n *be an odd positive integer. Then the elements of the sequence* $W = (w_1, w_2, \ldots, w_n) = \big(1 \pmod{n}, 1 + (n-1)/2 \pmod{n}, 1 + 2(n-1)/2 \pmod{n},\ \ldots,\ 1 + (n-1)(n-1)/2 \pmod{n}\big)$ *have the following properties.*

a) They form a complete system of residues $\pmod{n}$.

b) If $W^k = s_1, s_2, \ldots, s_{k\times n}$ *is the concatenation of* k *copies of* W*, then any* n *neighbouring elements of* W^k *form a complete residue system* (mod n).
c) If $j \geq 1$*, then* $s_{j+n+1} \equiv s_{j+1}$ (mod n).

Proof. It is sufficient to use Lemma 3 and some elementary calculations. □

Lemma 5 *Let* n *be an odd number* ($n \geq 5$)*,* $M = n - 1$ *and* $m = n - 2$*. If the small memory starts with control state* $(w_3, w_4, \ldots, w_n)$ *and the large memory starts with control state* $(2, 3, \ldots, n)$*, then the reference string* $R = (1, 2,\ \ldots,\ n)^{n-1)/2}$ *results n page faults in the small memory,* $n(n-1)/2$ *page faults in the large memory and the initial control states in both memories.*

Proof. This is a consequence of Lemma 4. □

Lemma 6 *Let* $1 \leq m < M < n$ *and* $b_1, b_2, \ldots, b_m$ *be arbitrary different elements of* $A = \{1, 2, \ldots, M+1\}$*. Then algorithm* ANOMALY *constructs a reference string resulting* $q = (b_1, b_2, \ldots, b_m)$ *in the small memory of size* m *while the pages are loaded into the large memory of size* M *in cyclical order* $1, 2, \ldots, n$.

The next algorithm is written in PASCAL-like pseudocode [11].

Construction algorithm ANOMALY(m, M, q, U).
Input: m size of the small memory; M size of the large memory; $q = (b_1, b_2, \ldots, b_m)$ the required control state of the small memory.
Output: $U = (u_1, u_2, \ldots)$ reference string.
Working variables: S actual state of the small memory; t index of the next reference; $A = \{1, 2, \ldots, M, M+1\}$ the set of pages.

```
 1. for $k \leftarrow 1$ to $M$
 2.       do $u_k \leftarrow k$
 3. $t \leftarrow M + 1$
 4. $i \leftarrow 0$
 5. while $i < m$ do
 6.       if $b_{i+1} \in S$
 7.             then while $b_{i+1} \in S$ do
 8.                         $u_t \leftarrow \min\{k | k \in A$ and $k \notin S\}$
 9.                         $t \leftarrow t + 1$
10.                   while $b_1 \in S(m)$ do
11.                         $u_k \leftarrow \min\{k | k \in A$ and $k \notin S$ and $k \neq b_{i+1}\}$
12.                         $t \leftarrow t + 1$
```

```
13.                $j \leftarrow 1$
14.                while $j < i + 1$ do
15.                    $u_t \leftarrow b_j$
16.                    $t \leftarrow t + 1$
17.                    $j \leftarrow j + 1$
18.        $u_t \leftarrow b_{i+1}$
19.        $t \leftarrow t + 1$
20.        $i \leftarrow i + 1$
```

Instead of a long proof of correctness of ANOMALY we explain how this algorithm generates U of the example used to disprove the conjecture. The input of the algorithm is $m = 5,\ M = 6$ and $q = (7,3,6,2,5)$. Lines 1–2 result the references 1, 2, 3, 4, 5, 6 and control state $q = (2,3,4,5,6)$ in the small memory. Now line 3 results $t = 7$, line 4 gives $i = 0$ and we start the while cycle in line 5. Since $b_1 = 7$ is missing from the small memory, we add $u_7 = 7$ to U (lines 18–20) changing the control state to $q = (3, 4, 5, 6, 7)$, increment t and i and return to line 5 where we observe that $b_2 = 3$ is in the memory. Therefore we continue with the while cycle in line 7 and add the minimal missing page to the reference string resulting control state (4,5,6,7,1). Since page 1 replaced b_2, we go to the next cycle in lines 10–12 and add the minimal missing pages differing from $b_2 = 3$ (that is 2, 4, 5, and 6) to U changing the control state to (1,2,4,5,6). Now this while cycle ends and using lines 13–17 we add 7 then using lines 18–20 add 3 to U ending in control state (4,5,6,7,3).

Now we return to line 5 and using lines 7-9 add 1 to U, then using lines 10–12 add 2, 4 and 5 to U changing the control state to (3,1,2,4,5). Now using lines 14–18 we add 7 and 3, then using lines 19–21 add 6 to U changing the control state to (4,5,7,3,6). Now lines 5–6 send us to lines 14–18 resulting reference $u_{22} = 2$ and control state (5,7,3,6,2).

We observe in lines 5–6 that b_5 is in the memory therefore we add 1 to U and so remove 2 from the memory. Now we continue U with 4 resulting the control state (3,6,2,1,4). Finally lines 14–18 add 7, 3, 6 and 2 and lines 19–21 add 5 to U implying the required control state.

Lemma 7 *Let* n *be an odd positive integer* $(n \geq 5)$ *and let* $M = n - 1,\ m = n-2$. *Then there exists a reference string* R *resulting anomaly ratio arbitrarily close to* $(n - 1)/2$.

Proof. This assertion is a consequence of Lemma 5 and Lemma 6. □

Main Theorem *For any large number* L *there exist parameters* m, M, *and* R *such that the anomaly ratio* $f_{\text{FIFO}}(R, M)/f_{\text{FIFO}}(R, m) > L$.

Proof. Let n be an odd integer satisfying $n > 2L + 1$ (and $n \geq 5$). Then the parameters m, M and R in Lemma 5 result an anomaly ratio greater than L. $\square$

5 Summary

If the memory parameters m and M are fixed, then the anomaly is bounded.

We suppose that the maximal anomaly occurs at cyclical reference strings, similar memory sizes, FIFO-like replacement in the large memory and MIN-like replacement in the small memory. If so then for fixed odd n our construction results the maximal possible anomaly ratio. We remark that there is a similar construction for even n resulting anomaly ratio arbitrarily close to $(n-2)/2$.

Acknowledgements

The authors are indebted to Mihály Molnár, former student of Eötvös Loránd University who using genetic algorithms discovered the first reference string resulting an anomaly ratio greater than 2. We thank the useful comments of Professor of Eötvös Loránd University András Benczúr and of the unknown referee.

The research was supported by the project TÁMOP-4.2.1/B-09/1/KMR-2010-003 of Eötvös Loránd University.

References

[1] S. Albers, On generalized connection caching, in: *ACM Symposium on Parallel Algorithms and Architectures* (Bar Harbor, 2000), *Theory Comput. Syst.*, **35,** (3) (2002) 251–267. ⇒81

[2] S. Albers, L. M. Favrholdt, O. Giel, On paging with locality of reference, *J. Comput. System Sci.*, **70,** 2 (2005) 145–175. ⇒81

[3] M. Arató, A note on optimal performance of page storage, *Acta Cybernet.* (Szeged), **3,** 1 (1976/77) 25–30. ⇒82

[4] L. A. Belady, A study of replacement algorithms for a virtual storage computer, *IBM Syst. J.*, **5,** 2 (1965) 78–101. ⇒81

[5] L. A. Belady, R. A. Nelson, G. S. Shedler, An anomaly in space-time characteristics of certain programs running in paging machine, *Comm. ACM*, **12,** 1 (1969) 349–353. ⇒80, 81, 82, 83

[6] L. A. Belady, F. P. Palermo, On-line measurement of paging behavior by the multivalued MIN algorithm, *IBM J. Res. Develop.*, **18** (1974) 2–19. ⇒81

[7] A. Benczúr, A. Krámli, J. Pergel, On the Bayesian approach to optimal performance of page hierarchies, *Acta Cybernet.* (Szeged), **3,** 2 (1976/77) 78–79. ⇒82

[8] J. Boyar, L. M. Favrholdt, K. S. Larsen, The relative worst-order ratio applied to paging, *J. Comput. System Sci.*, **73,** 5 (2007) 818–843. ⇒81

[9] M. Caminada, L. Amgoud, On the evaluation of argumentation formalisms, *Artificial Intelligence*, **171,** 5–6 (2007) 286–310. ⇒81

[10] M. Chrobak, J. Noga, LRU is better than FIFO, *Algorithmica*, **23,** 2 (1999) 180–185. ⇒80, 81

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to algorithms* (Second edition), The MIT Press/ McGraw-Hill Book Company, Cambridge/Boston, 2009. ⇒84

[12] P. Denning, Virtual memory. *Comput. Surveys*, **2,** 3 (1980) 153–189. ⇒ 80, 81

[13] L. Epstein, Y. Kleiman, J. Sgall, R. van Stee, Paging with connections: FIFO strikes again, *Theoret. Comput. Sci.*, **377,** 1–3 (2007) 55–64. ⇒80, 81

[14] M. A. Franklin, G. S. Graham, R. K. Gupta, Anomalies with variable partition paging algorithms, *Comm. ACM*, **21,** 3 (1978) 232–236. ⇒81

[15] J. Gecsei, D. R. Slutz, I. L. Traiger, Evaluation techniques for storage hierarchies, *IBM Syst. J.*, **9,** 2 (1970) 78–117. ⇒81

[16] A. M. Iványi and L. N. Korolev, Classification of paging algorithms, in: *Problems of Interpretation of Experiments* (eds. A. N. Tihonov, P. N. Zaikin and V. Y. Galkin), Moscow State University, Moscow, 1975, pp. 105–125. ⇒82, 83

[17] A. M. Iványi, On dumpling-eating giants, in: *Finite and Infinite Sets* (eds. A. Hajnal, L. Lovász and V. T. Sós), North-Holland, Amsterdam, 1984, pp. 379–390. ⇒81

[18] A. M. Iványi, R. L. Smelyanskiy, *Elements of theoretical programming* (in Russian), Moscow State University, Moscow, 1980. ⇒81

[19] Ya. A. Kogan, A class of hierarchical paging algorithms, *MTA Számitástechn. Automat. Kutató Int. Tanulmányok* (3rd Visegrád Winter School *Theory of Operating Systems*, Visegrád, 1977), No. 69 (1977) 7–13. ⇒81

[20] S. Kolahi, Dependency-preserving normalization of relational and XML data, *J. Comput. System Sci.*, **73,** 4 (2007) 636–647. ⇒81

[21] R. Korf, Linear-time disk-based implicit graph search, *J. ACM*, **55,** 6 (2008) Art. 26, pp. 40. ⇒81

[22] K. V. Malkov, D. V. Tunitskii, On an extremal problem of adaptive machine learning that is connected with the detection of anomalies (in Russian), *Avtomat. i Telemekh.*, 2008, no. 6, 41–52; translation in *Autom. Remote Control*, **69,** 6 (2008) 942–952. ⇒81

[23] B. Mans, C. Roucairol, Performances of parallel branch and bound algorithms with best-first search, *Discrete Appl. Math.*, **66,** 1 (1996), 57–74. ⇒81

[24] S. D. Mihnovskiy, N. Z. Shor, Estimation of the number of page faults in paged virtual memory (in Russian), *Kibernetika*, **1,** 5 (1965) 18–20. ⇒81

[25] D. A. Naumann, Observational purity and encapsulation, *Theoret. Comput. Sci.*, **376,** 3 (2007) 205–224. ⇒81

[26] H. Raquibul, R. Sohel, DesynchLRU: An efficient page replacement algorithm with desynchronized cache and RAM, in: *Abstracts of Int. Conf. on Applied Informatics* (Eger, January 27–30, 2010). ⇒80, 81

[27] S. Roosta, *Parallel processing and parallel algorithms*, Springer-Verlag, New York, 1999. ⇒81

[28] A. Silberschatz, P. Galvin and G. Gagne, *Applied operating system Concepts*, John Wiley and Sons, New York, 2000. ⇒81

[29] D. Sleator, R. E. Tarjan, Amortized efficiency of list update and paging rules, *Comm. ACM*, **28,** 2 (1985) 202–208. ⇒81

[30] A. J. Smith, Analysis of the optimal, look-ahead demand paging algorithms, *SIAM J. Comput.*, **5,** (4) (1976) 743–757. ⇒81

[31] A. M. Sokolov, Vector representations for efficient comparison and search for similar strings (in Russian), *Kibernet. Sistem. Anal.*, **43,** 4 (2007), 18–38, 189; translation in *Cybernet. Systems Anal.*, **43,** 4 (2007) 484–498. ⇒ 81

[32] H.-G. Stork, On the paging-complexity of periodic arrangements, *Theoret. Comput. Sci.*, **4,** 2 (1977) 171–197. ⇒81

[33] N. B. Waite, A real-time system-adapted anomaly detector, *Inform. Sci.*, **115,** 1–4 (1999) 221–259. ⇒81

[34] Wikipedia, Belady's algorithm,
http://en.wikipedia.org/wiki/Belady%27s_Min#Belady.27s_Algorithm, 2010. ⇒81

[35] Wikipedia, Belady's anomaly,
http://en.wikipedia.org/wiki/Belady%27s_anomaly, 2010. ⇒81

[36] Wikipedia, FIFO page replacement algorithm,
http://en.wikipedia.org/wiki/FIFO, 2010. ⇒80, 81

[37] Wikipedia, LRU page replacement algorithm,
http://en.wikipedia.org/wiki/LRU, 2010. ⇒81

[38] Wikipedia, Page replacement algorithm,
http://en.wikipedia.org/wiki/Page_replacement_algorithm, 2010. ⇒ 80, 81

[39] C. Wood, Christopher, E. B. Fernandez, T. Lang, Minimization of demand paging for the LRU stack model of program behavior. *Inform. Process. Lett.*, **16,** 3 (1983) 99–104. ⇒81

[40] N. E. Young, On-line paging against adversarially biased random inputs, (*Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, 1998). *J. Algorithms*, **37,** 1 (2000) 218–235. ⇒80, 81

Received: October 17, 2009 • Revised: March 8, 2010

Acta Univ. Sapientiae, Informatica, **2,** 1 (2010) 90–98

Macro and micro view on steady states in state space

Branislav Sobota
Dept. of Computers and Informatics
FEEI, Technical University of Košice
Letná 9, 04200 Košice, Slovakia
email: `Branislav.Sobota@tuke.sk`

Milan Guzan
Dept. of Theoretical Electrotechnics
and Electrical Measurement
FEEI, Technical University of Košice
Letná 9, 04200 Košice, Slovakia
email: `Milan.Guzan@tuke.sk`

Abstract. This paper describes visualization of chaotic attractor and elements of the singularities in 3D space. 3D view of these effects enables to create a demonstrative projection about relations of chaos generated by physical circuit, the Chua's circuit. Via macro views on chaotic attractor is obtained not only visual space illustration of representative point motion in state space, but also its relation to planes of singularity elements. Our created program enables view on chaotic attractor both in 2D and 3D space together with plane objects visualization – elements of singularities.

1 Introduction

The visualization is good idea to show imagines, ideas, design, construction, realization or effects. It is also one way of verification before realization our goals. Computer based visualization brings utilization of physical, or simulated electric parameters course graphical interpretation in non-linear circuit theory together with other fields. From beginning it was used 2D visualization with possibility of color utilization to be more illustrative or to explain actions proceeding in non-linear circuits [1, 9, 13, 15]. High-performance or parallel

Computing Classification System 1998: B.7.2
Mathematics Subject Classification 2010: 94C99, 68U05
Key words and phrases: chaos, Chua's circuit, singularity, trajectory, 3D visualization

computers enable to take advantage of 3D state space axonometry [14]. Actual available solutions provide high performance visualization suitable for 3D interactive presentation of processes and effects with support for over million saturated colors in hi-resolution mode and for use in all graphics-intensive applications. This paper describes actual possibilities of PC for visualization of steady states of chaos generating circuit.

2 Methods used for trajectory visualization

In last 24 years there was intensive interest of scientific community to analyse and applied Chua's circuit generating chaos. Presentation of trajectories needs to solve system (1) describing physical Chua's circuit.

$$\begin{aligned} C_1(du_1/dt) &= G(u_2-u_1)-g(u_1)-I &&= Q_1 \\ C_2(du_2/dt) &= G(u_1-u_2)+i &&= Q_2 \\ L(di/dt) &= -u_2-\rho i &&= Q_3 \end{aligned} \tag{1}$$

where

$$\begin{aligned} g(u_1) = \; & m_2u_1 + 1/2(m_1-m_0)(|u_1-B_P|-|u_1+B_P|)+ \\ & +1/2(m_2-m_1)(|u_1-B_0|-|u_1+B_0|) \end{aligned} \tag{2}$$

Next we consider control pulse $I = 0$, the resistance of the inductance $\rho = 0$.

For parameters (3) in [5] there were found chaotic attractors showed in Fig. 1 in Monge projection.

$$\begin{gathered} C_1 = 1/9, C_2 = 1, L = 0.142857, G = 0.7, \\ m_0 = -0.8, m_1 = -0.5, m_2 = 5, B_p = 1, B_0 = 14 \end{gathered} \tag{3}$$

Computer program was designed in C language by author of [2] and used for clarifying of place in state space where chaos originates [3]. It is only short segment of intersection two surfaces related to circuit singularities $P+$ and 0, or 0 and $P-$. In despite of explaining with help of tables and 2D presentation was definite, 3D visualization provides faster and lighter illustration of actions which proceed in specific non-linear circuit. Therefore 3D visualization is valued as from scientific as from edifying point of view [4].

3 Visualization in 3-dimensional space

Chaos visualizing system was designed for visualization Chua's attractor in 3D space in real time and it is based on visualizing kernel developed on DCI

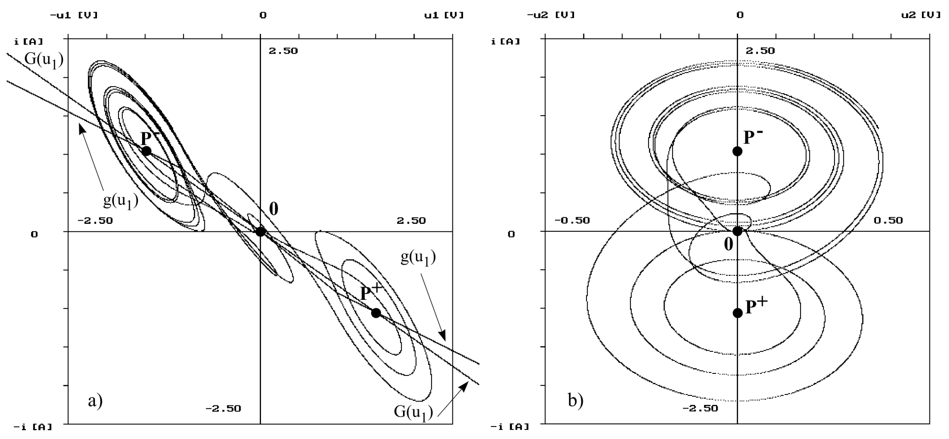


Figure 1: Monge projection of chaotic attractor system (1), for parameters (3) to plane: a) $i - u_1$, b) $i - u_2$

FEEI TU Košice [6]. An application is implemented in C++ language using OpenGL graphics library. The application can work with *I-V characteristics*, Chua's attractor trajectory, or limit cycle and it can visualize elements of the singularity planes. Additionally this visualization depicts representative point movement and it creates chaotic attractor using two basic modes (*continuous* and *sequential*). This application can be used not only for concrete Chua's circuit. It is usable also for Chua's circuit like structures analyzed in [7, 8].

Chaos visualizing system provides three basic visualizing modes: *continuous mode, sequential mode* and *I-V characteristics visualization mode*. System allows using four projection types (*3D* $u_1 - i - u_2$ *projection* (see Fig. 5), *2D* $i - u_2$ *projection, 2D* $i - u_1$ *projection* and *2D* $u_2 - u_1$ *projection*) for better-examined circuit understanding. Settable basic visualizing parameters for chaotic attractor visualization are: *drawing speed, points omission, chaotic attractor point size, comet length* and *attractor colour*. The combination of these parameters defines final visualization form of chaotic attractor. In case of 3D graphic accelerator supported acceleration of graphic interface OpenGL is drawing of 3D primitives accelerated by graphic card. It enables increasing performance and using some graphical improvements cannot be used without acceleration in real time. Second way is use of parallel computational system [10, 11] for faster or better visualization.

3.1 Visualization program utilization

3D visualization of chaotic attractor for parameters (3) is shown in Fig. 2. To next manipulation with chaotic attractor as 3D object is necessary to fulfill the following steps:

- Load input file – chaotic attractor. Input file size is above 500 MB, therefore program enables to choose number of trajectory points, which is loaded from file and consequently displayed.
- Load *I-V characteristics* `G(u1)` and `g(u1)` from files.
- Set background color for scene displaying and
- Set colors and line-width of *I-V characteristics* and *chaotic attractor*.

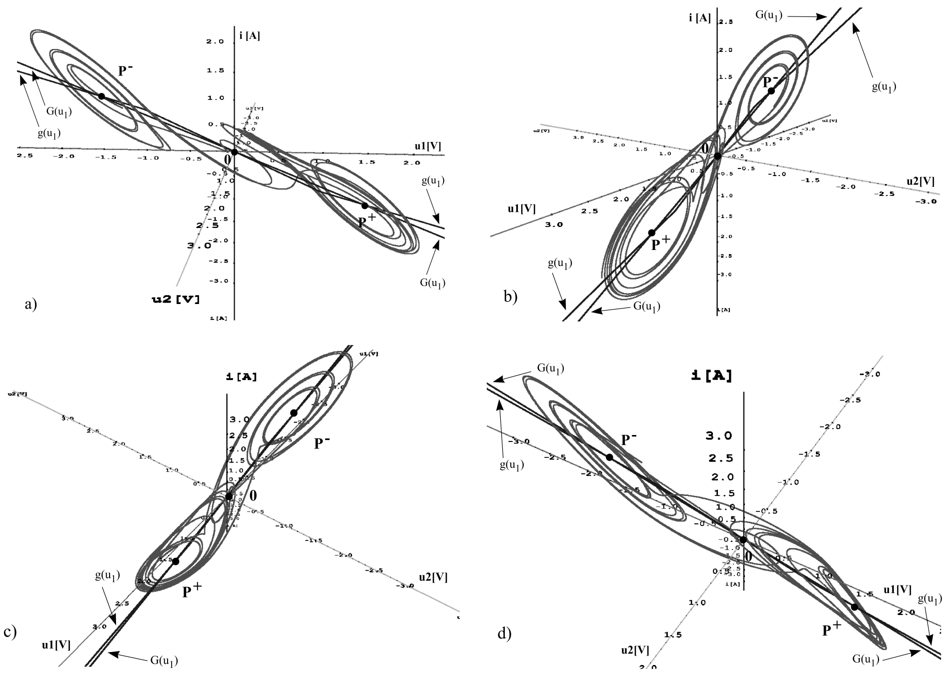


Figure 2: Chaotic attractor depicted in continuous mode

Views on chaotic attractor from various sides can be obtained by rotation of camera position horizontally and also vertically around visualized object. Fig. 2a and b show horizontal camera swing out, while Fig. 2c, d show vertical camera swing out to chaotic attractor. In this way top view on observed object can be obtained. On Fig. 3 are displayed different looks to the same chaotic

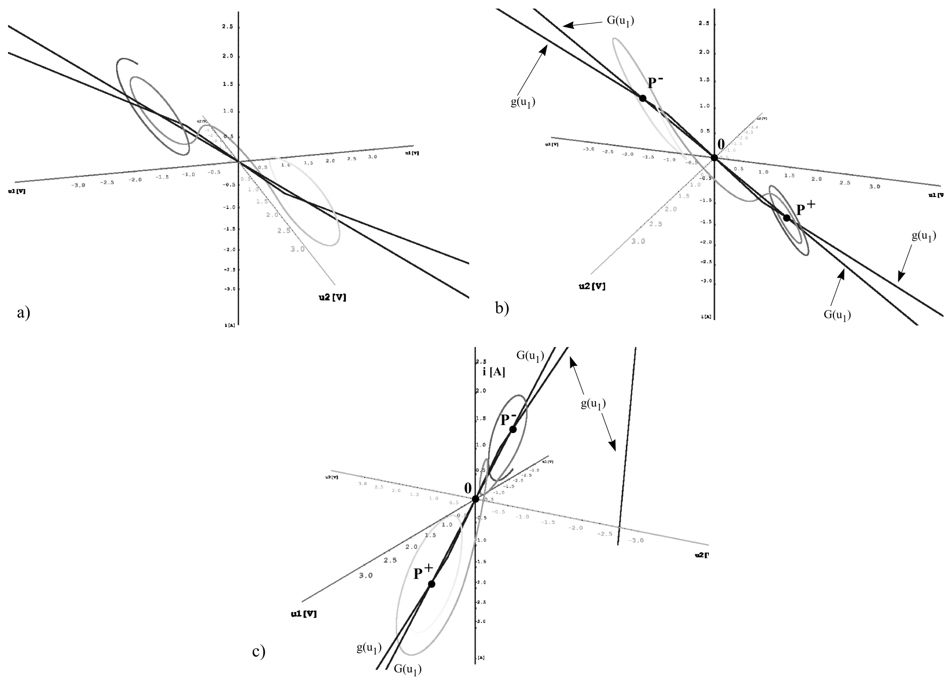


Figure 3: Chaotic attractor depicted in sequential mode with comet effect

attractor. It is visualizing in sequential mode with comet effect [12]. The comet length is adjustable from 512 to 16384 bits. Fig. 2 and also Fig. 3 show that left and right discs of chaotic attractor are situated in plane. It is possible to display this plane in the program. Mathematical description of this plane presented in [3] is outlined by the following equation:

$$y_1 = \alpha_{11}\Delta u_1 + \alpha_{12}\Delta u_2 + \alpha_{13}\Delta i. \qquad (4)$$

It is available to define in application menu input parameters for appropriate planes as e.g.: singularity coordinates (i, u_1, u_2), eigenvectors $(\alpha_{11}, \alpha_{12}, \alpha_{13})$, width, length and planes colors. The elements of the singularity planes are displayed by application using of selected colors. Fig. 4a shows these elements of the singularities $EP+$ and $EP-$ representation as parallel planes getting across singularities $P+$ and $P-$. Fig. 4b shows also the third singularity element 0, called $E0$. Fig. 4b shows, that plane $E0$ is not parallel with $EP+$

and EP−. Via macro views on chaotic attractor showed in Fig. 2– 4 we obtain not only visual space illustration of representative point motion in state space, but also its relation to planes of singularity elements. Vertical camera

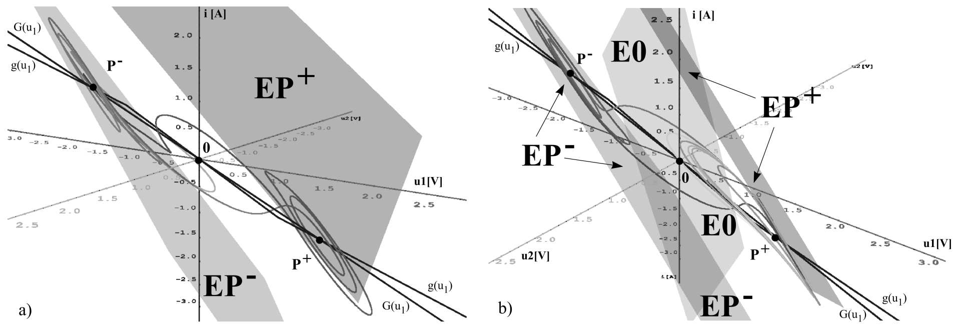
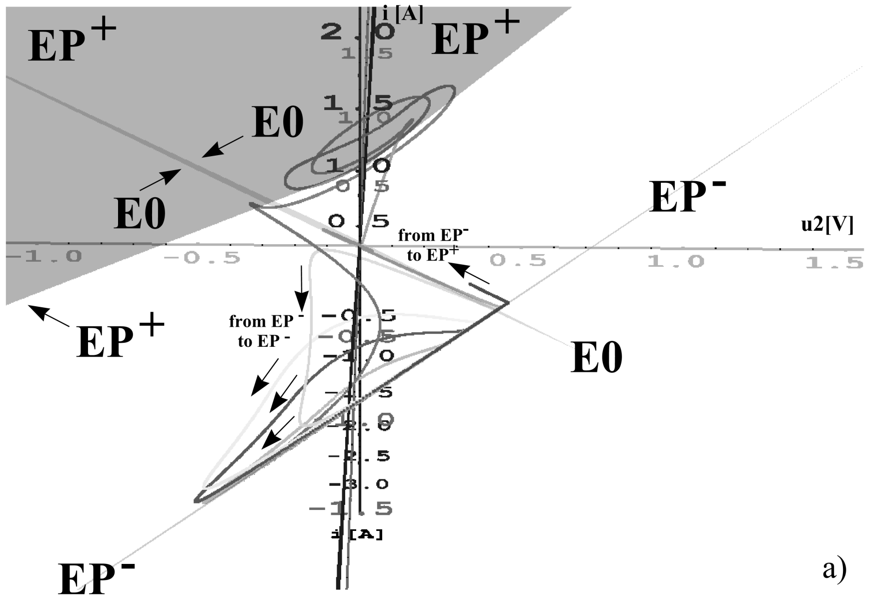


Figure 4: Visualization of singularity elements in state space: a) EP− and EP+ b) E0, EP− and EP+

swing round with regard to Fig. 4 enables to see such a part of state space, where chaos arises in Chua's circuit. It is intersection of two planes EP+ and E0 or E0 and EP−. This situation is displayed in macro view on Fig. 5a. Marking of conjunction region of mentioned elements E0 and EP− enables to obtain micro view to just site of state space, where chaos originates. Website http://kteem.fei.tuke.sk/guzan/ausi09 contains dynamic versions of some pictures mentioned in this article. Video appears jerky. This is caused by camera position rotation around object (usually 5°). Program enables to set angle step of camera position rotation. It was used when Fig. 5 was generated.

4 Conclusion

The 3D visualization brings new dimensions to the visualization of physical or electric effects. Visualization of chaotic attractor and elements of the singularities provides better understanding of representative point movement in state space, what was still possible only with help of representation in projection planes. From computer graphics point of view are produced big data-sets. Possible parallel processing (e.g. on multi-core or multi-computers platform) shortens computational time. Big-screen display solutions increase quality and ability of immersion into 3D space. It is possible to use finer integration step for better quality and more detailed visualization output (continuous trajectory



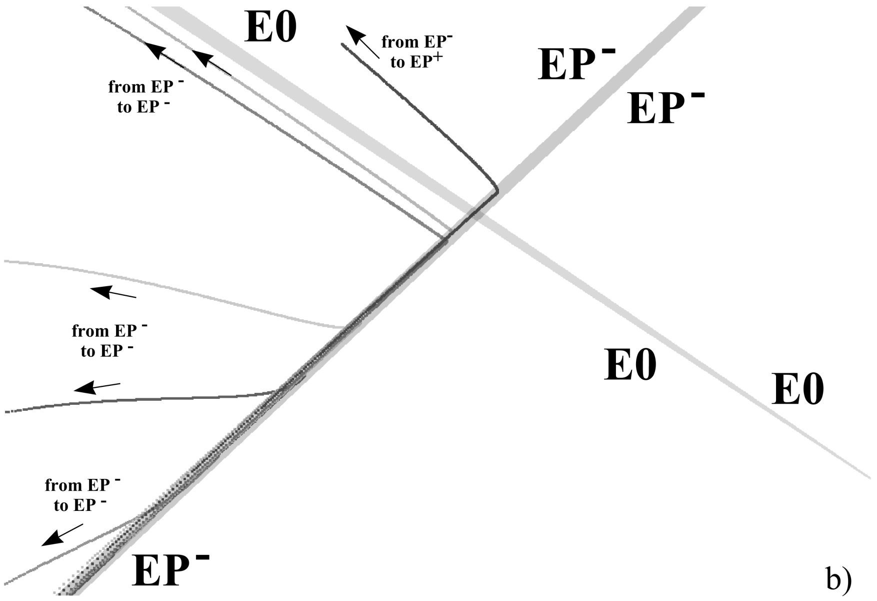


Figure 5: 3D view on intersection of singularity elements EP− and E0 (view on place of chaos origination in Chua's circuit)

displaying) in this case. Generally, there are two important application areas in physical or electric effects visualization where big-display environments are used: displaying images at very high resolution in real time exceeding those of available screens (monitors) and/or graphics cards and providing a larger field-of-view and better immersion into the explored attractor‘s space. Whole system enables visualization for user defined Chua‘s attractor where user has in standard visualizing mode (3D projection) 6 degrees of freedom for motion in explored attractor‘s space, which mean translation in 3 axis and rotating around them. It means that system is capable to visualize the Chua’s attractor from any point and from any angle. Actual available solution provide so unique big wide-screen high performance visualization solution suitable for 3D interactive presentation of attractor‘s space. Input devices are standard keyboard and mouse generally, but space mouse or other specialized input device can be also used.

Acknowledgements

This work is supported by VEGA grant project No. 1/0646/09: “Tasks solution for large graphical data processing in the environment of parallel, distributed and network computer systems”, by KEGA no. 3/6386/08: “E-learning and web oriented education technology of subjects from the field of electrical measurement for presentation and distance form of study” and by Agency of the Ministry of Education of the Slovak Republic for the Structural Funds of the EU under the project Center of Information and Communication Technologies for Knowledge Systems (project number: 26220120020).

References

[1] P. Galajda, V. Špány, M. Guzan, The state space mystery with negative load in multiple-valued logic, *Radioengineering*, **8,** 2 (1999) 2–7. ⇒90

[2] M. Guzan, Multifunctionality of Chua‘s circuit at R changes, *International Conference of Electrotechnic Teachers*, 16–18 September 2008, Košice-Herľany, Slovakia, TU Košice, 2008, pp. 61–66 (in Slovak). ⇒91

[3] M. Guzan, P. Galajda, L. Pivka, V. Špány, Element of singularity is a key to laws of chaos, *15th International Czech-Slovak Scientific Conference RADIOELEKTRONIKA 2005*, 2005, pp. 33–36. ⇒91, 94

[4] R. Krehel̆, Sensoren in der Prozessautomation und Prozessinformatik, in: *CO-MAT-TECH 2004: 12th International Scientific Conference*, Trnava, Slovakia, 14–15 October 2004, STU Bratislava, 2004, pp. 665–672. ⇒91

[5] T. Matsumoto, L. Chua, M. Komuro, The double scroll, *IEEE Trans. Circuits Syst.*, Vol. CAS-32, No. 8, 1985. ⇒91

[6] R. Mocnár, *Chaotic attractor visualization in state space*, Diploma Thesis, DCI FEEI TU Košice, 2006 (in Slovak). ⇒92

[7] J. Petržela, Modeling of the strange behavior of the selected nonlinear dynamical systems, Part I.: Oscillators, *PhD Thesis Edition*, Vol. 502. ⇒92

[8] J. Petržela, Z. Kolka, S. Hanus, Simple chaotic oscillator: from mathematical model to practical experiment, *Radioengineering*, **15,** 1 (2006) 6–11. ⇒92

[9] L. Pivka, V. Špány, Boundary surfaces and basin bifurcations in Chua's bircuit, *J. Circuits Syst. Comput.*, **3,** 2 (1993) 441–470. ⇒90

[10] B. Sobota, Parallel hierarchical model of visualization computing, *J. Inf. Control Manag. Syst.*, **5** (2007) 345–350. ⇒92

[11] B. Sobota, J. Perháč, Cs. Szabó, An application of parallel, distributed and network computer systems to solve computational processes in an area of large graphical data volumes processing, *Comput. Sci. Technol. Res. Survey*, KPI FEI TU Košice, 2008, 3, pp. 37–42. ⇒92

[12] V. Špány, Personal communication, Department of Electronics and Multimedia Communications, Faculty of Electrical Engineering and Informatics, TU Košice, January 2006. ⇒94

[13] V. Špány, P. Galajda, M. Guzan, Boundary surfaces of one-port memories, in: *Tesla III Milenium: Proc. 5th Int. Conf.*, Beograd, 1996, pp. IV. 130–IV. 137. ⇒90

[14] V. Špány, P. Galajda, M. Guzan, The state space mystery in multiple-valued logic circuit with load plane – part I., *Acta Electrotechnica Inform.*, **1,** 1 (2001) 17–22. ⇒90

[15] V. Špány, L. Pivka, Boundary surfaces in sequential circuits, *Internat. J. Circuit Theory Appl.*, **18,** 4 (1990) 349–360. ⇒90

Received: July 15, 2009 • Revised: February 28, 2010

Acta Univ. Sapientiae, Informatica, **2,** 1 (2010) 99–112

Public-key cryptography in functional programming context

Gyöngyvér Márton
Sapientia Hungarian University of Transylvania
Cluj
Department of Mathematics and Informatics
Tg. Mureş, Romania
email: `mgyongyi@ms.sapientia.ro`

Abstract. Cryptography is the science of information and communication security. Up to now, for efficiency reasons cryptographic algorithm has been written in an imperative language. But to get acquaintance with a functional programming language a question arises: functional programming offers some new for secure communication or not? This article investigates this question giving an overview on some cryptography algorithms and presents how the RSA encryption in the functional language Clean can be implemented and how can be measured the efficiency of a certain application.

1 Functional programming

Functional programming is based on the lambda-calculus, with its main principle developed in 1930, by Alonzo Church and Stephen Cole Kleene. With lambda-calculus we can define the notion of computable function, and using the rules defined by lambda-calculus we can express and evaluate any computable function. But in the early development stage of computer science the computer calculating mechanisms were based on another calculating model, such as a Turing machine's. The Turing machine functioning mechanism was described in 1936 by the mathematician Alan Turing, in the same time as

Computing Classification System 1998: D.1.1
Mathematics Subject Classification 2010: 68N18
Key words and phrases: functional programming, public-key cryptography

lambda calculus. The Turing machine was the base of Neumann's architecture computer model and determined the developmental direction of programming language. In consequence most of the real-world applications was based on imperative programming language.

Only with the development of software technologies it became possible to design programming languages based on stable mathematics like functional programming language and declarative programming language: Lisp (1958), Prolog (1970), Haskell (1990), Clean (1995) and so on. But the usage of these languages in the real-world applications is still very little. Some well-known applications of functional languages are the Yahoo shopping engine and the telecommunication software design by Ericsson.

A program in a functional programming language consists of a collection of function definition and an initial expression. So the basic method of computation is the application of functions to arguments with the goal to evaluate the initial expression. The basic concepts and the basic elements of a functional programming language can be defined as follows:

- persistent data structures: data once built never changes
- recursion: primary control structure
- high-order functions: functions that take functions as argument and in case return function as result.

By contrary to functional programming in the imperative programming style the main goal is to follow the changes of the states. The basic concepts of an imperative language are:

- mutable data structures: with assignment statement we can change the value of some data
- looping: the primary control structure
- first-order programming: we cannot operate with function as first-class entities.

The Clean programming language, that we are use for implementing the RSA file encryption scheme is a pure and lazy functional language, with a fast compiler. Was developed on the University of Nijmegen, Netherlands. The first version was released in 1995. This language has most of the characteristic that a functional language must to have, some of them: isn't allowed the destructive updates; has the referential transparency property; the basic computation form is the recursion; we can use high order functions; it is strongly typed; list comprehension is allowed; polymorphism is allowed too and so on.

In the real world one of the most known software for cryptography, writing in functional programming language is Cryptol [3]. It was designed by Galois Connections Inc. in consultation with expert cryptographers between 2003-2008. As developers say Cryptol is a high-level specification language for cryptography that means programmers who use Cryptol can focus on the cryptography itself, and are not attended by machine-level details. In the same time they can deal with low-level problems, namely it can work with low level data, such as array of bits. The code written in Cryptol can be converted to other languages such as Haskell, VHDL and C. It can use with various platform such as embedded systems, smart cards and FPGAs.

Another well-known software package in this field is the RSA-Haskell, [11], which was published in 2007, by David Sankel. It is written in Haskell and as the author says it is a "collection of command-line cryptography tools and a cryptography library". With RSA-Haskell using command-line tools users can do secure communication: encrypt/decrypt some message, can identify the sender and authenticate the message. The crypto library is licensed under the GPL (General Public License), and allowed for users to access some cryptography algorithms to incorporate these in their application. The size of the RSA-key is 2048 bit, and SHA512 hash algorithm is used in conjunction with OAEP (Optimal Asymmetric Encryption Padding), [12].

2 Cryptography

In the past Cryptography was the science of secret codes. Today due to the electronic world the data security is in the center of the attention. This means that it became more than producing secrete code. The topmost tasks are privacy, integrity, authentication, and nonrepudiation.

Cryptography algorithms can be classified in two groups such as symmetric-key cryptography and public-key cryptography according to what kind of keys are in use in the system: secret or public keys. In the case of symmetric-key cryptography the processes of encrypting and decrypting are coming to pass with the same secret key, in contrast to public-key cryptography where the encrypting are coming to pass with public key, and the decrypting with an other key, with private key. Besides encryption schemes more other schemes belong to the public-key cryptography such as digital signatures schemes, authentications schemes, and so on [10]. But on backwards our attention will focus only on public-key encryption schemes. So, now we will give the formal definition of the public-key encryption scheme, [4]:

Definition 1 *A public-key encryption scheme with message space* $\mathbb{M}$ *can be define with three algorithms:* $\mathsf{PKC} = (\mathsf{GEN}, \mathsf{ENC}, \mathsf{DEC})$*, where*

- GEN *is the key generation algorithm, which determines in a random way the public and secret key-pairs:* $(p_k, s_k) = \mathsf{GEN}(\epsilon)$*, where* p_k *is the public key and* s_k *is the secret key,*
- ENC *is the encryption algorithm, which encrypts a message* M*, producing the ciphertext:* $C = \mathsf{ENC}_{p_k}(M)$*, where* $M \in \mathbb{M}$*,*
- DEC *is the decryption algorithm, which decrypts the ciphertext* C*,* $M = \mathsf{DEC}_{s_k}(C)$.

For the correctness of the system we require that $\mathsf{DEC}_{s_k}(\mathsf{ENC}_{p_k}(M)) = M$. For the security of the system the corresponding requirements can not be claim so easy. One of them is that the public key inversion problem (finding the secret key for a given public key) must be based on a hard mathematical problem, in average case of the problem instances. Another requirement is that the ciphertext inversion problem (finding the encrypted M message for a given C ciphertext and p_k public key) must be hard. But very few problems we are known that achieves these requirements, thus not surprising that in a real word application the most of the cryptography systems are based on the following mathematical problems:

- factoring large integers, for instance the RSA cryptosystem
- computing discrete logarithms, for instance the ElGamal cryptosystem.

For these problems no one knows polynomial time algorithms, moreover these problems are those scarce problems that are not classify between the P and NP-complete classes, [10].

Because in our implementation we concern on RSA file encryption scheme, first we will present the basic RSA encryption scheme, [8]. This scheme consists in three steps, corresponding to the 3 algorithms specified in the formal definition of public-key encryption scheme: key generation, encryption, decryption. The message space is the $\mathbb{M} = \mathbb{Z}_m^*$, where m is an integer number, calculated in the key generation step. The value of m determines the order of magnitude of the RSA-key.

- Generating the RSA keys consists on the following steps, where ϕ is the Euler function, and gcd is the greatest common divisor of the arguments, [6]:
  - generating two big random prime numbers: p, q,

  - calculates the product $m = p \cdot q$, henceforth m we will call modulus,
  - selects randomly e, where $1 \le e \le \phi(m)$ and $gcd(e, \phi(m)) = 1$, henceforth e we will call encryption exponent,
  - computes d, where: $1 \le d \le \phi(m)$, $d \cdot e = 1 \pmod{\phi(m)}$, henceforth d we will call decryption exponent,
  - the public key consist: (e, m),
  - the private key consist: (d, m).

- For encryption of $M \in \mathbb{Z}_m^*$ we do: $C = M^e \pmod{m}$.
- For decryption of $C \in \mathbb{Z}_m^*$ we do: $M = C^d \pmod{m}$.

3 Algorithms in Clean

Now we shall present the implementation details of the RSA file encryption.

Firstly we mention that arithmetic with large numbers is quite easy in Clean, through importing the BigInt library, [5]. Henceforth we give the definition of this importing as well as the definitions of constants, are used several times in the system to be realized.

```
import BigInt
my_one :== toBigInt 1
my_two :== toBigInt 2
my_zero:== toBigInt 0
alph :== toBigInt 256
```

By the way of implementing RSA file encryption system, several questions are coming up: to find the modular multiplicative inverse; to perform the modular exponentiation; to generate big (more than 100 digit) random prime number; to convert the number from the base p in the base p^k and inverse; to perform the RSA encryption on numbers; do the file I/O task; create a graphical interface.

In the following sections, one after another we will briefly present how we are resolved this questions in the Clean programming language.

3.1 Multiplicative inverse

In order to find the multiplicative inverse of an integer $a \pmod{b}$ we need to resolve the congruence: $a \cdot x_1 = 1 \pmod{b}$ with the unknown coefficient x_1. This congruence can be solved by using the extended Euclid's algorithm, [2]. For this we write two functions:

- an auxiliary function: $\mathsf{seuclid}$, with the role of doing the proper computation, namely to calculate the coefficient of a,
- the main function $\mathsf{meuclid}$ doing the necessary initialization and the first call of $\mathsf{seuclid}$.

The Clean code that do this is the following:

```
seuclid :: BigInt BigInt BigInt BigInt->BigInt
seuclid a b x1 x2
    |b==my_zero = x2
    |otherwise = seuclid b (a rem b) (x2-(a/b)*x1) x1

meuclid :: BigInt BigInt -> BigInt
meuclid a b
    #res = seuclid a b my_zero my_one
    |res<my_zero = (res+b) rem b
    |otherwise = res rem b
```

3.2 Modular exponentiation

The modular exponentiation calculates the value of $x^p \pmod{m}$ using the fast exponentiation technique, [9]. For this purpose we write the function, mexp.

The Clean code that do this is the following:

```
mexp :: BigInt BigInt BigInt-> BigInt
mexp x n m
    |n == my_zero = my_one
    #x1 = (x*x) rem m
    |isOdd n = (x * mexp x1 (n/my_two) m) rem m
    = mexp x1 (n/my_two) m
```

3.3 Random prime number generation

To set the RSA-key we need prime random numbers. For testing if a random number with the right size is prime or not we use probabilistic primality test which quickly eliminates the composite numbers. For this test we use the Miller-Rabin primality test [9].

The formal definition of this test is the following:

Definition 2 *Let the odd number* q *and* j *such that* $n - 1 = 2^j q$*. If the odd number* n *is prime, then for all numbers* x*, where* $\mathsf{gcd}(x, n) = 1$ *one of the following statements is true:*

- $x^q = 1 \pmod{n}$,
- *for one of the* i*:* $x^{2^i q} = n - 1 \pmod{n}$*, where* $0 \leq i \leq j-1$.

For that in the Clean programming language we write the function $mfind$ which determines the value j and q in the exponent $n-1$ by calling the $sfind$ function. To test the second statement of the above definition we write the $subtest$ function. The validity of the first statement is tested by the main function $millerrabin$. For this purpose we calculate the value $x^q \pmod{n}$ with the modular exponentiation algorithm.

The Clean code that do this is the following:

```
millerrabin :: BigInt BigInt -> Bool
millerrabin n x
    #(q, j) = mfind n
    #y = mexp x q n
    |y == my_one = True
    = subtest n y j

mfind :: BigInt -> (BigInt, BigInt)
mfind n = sfind (n - my_one)my_zero

sfind :: BigInt BigInt -> (BigInt, BigInt)
sfind q k
    |isEven q = sfind (q / my_two) (k + my_one)
    = (q, k)

subtest :: BigInt BigInt BigInt -> Bool
subtest n y j
    |j < my_one = False
    |y == (n - my_one) = True
    |y == my_one = False
    = subtest n ((y * y) rem n) (j - my_one)
```

For generating random number we use the algorithm of linear congruential generator [9].

To be certain of that an odd number n is prime, we must have the result 'True' in the $millerrabin$ function, for more, different x random numbers. For that we examine the number n if it is odd or it is even, and in the case it is odd we generate t bit random number and test t times for these x, if the $millerarbin$ function is 'True' or not.

3.4 Base expansion

In the RSA encryption scheme we can encrypt only a single large number. In RSA file encryption we must to encrypt many bytes. To achieve this we do a pre-processing on bytes that we want to encrypt. We choose k bytes, that corresponds to the size of the block that we are encrypt all at once. We see these bytes as digits in base 256 and we make a conversion from 256 to base 256^k [6]. After that we have a single big number which we encrypt corresponding the scheme presented in section 2. The Clean function which resolves this conversion is $myconvert1$ by the help of auxiliary function $subconvert1$.

The Clean code that do this is the following:

```
myconvert1 :: BigInt BigInt BigInt-> [Char]
myconvert1 nr p pk = subconvert1 nr p pk my_one []

subconvert1 :: BigInt BigInt BigInt BigInt [Char] -> [Char]
subconvert1 nr p pk nr1 tomb
    | nr1 >= pk = tomb
    | otherwise = [(toChar (toInt (nr rem p))) :
                    (subconvert1 (nr/p) p pk (nr1+my_one) tomb )]
```

After encryption in the process of decryption we must do the inverse conversion, so we need an algorithm that makes the conversion from p^k to p.

The Clean code that do this is the following:

```
myconvert :: [Char] BigInt BigInt -> BigInt
myconvert n p pk = subconvert n p pk my_zero my_one

subconvert :: [Char] BigInt BigInt BigInt BigInt -> BigInt
subconvert [] p pk nr exp  = my_zero
subconvert [kezd : veg] p pk nr exp
    | pk == nr = my_zero
     = (toBigInt (toInt kezd)) * exp +
            (subconvert veg p pk (nr+my_one) (exp*p) )
```

3.5 RSA encryption on numbers

To attain the RSA encryption first we generate the private, secret key-pairs, after that we perform encryption/decryption with this keys. For efficiency reason, but without loss of security we have choose e as a constant: 65537 to be encryption exponent in the public key pair. This choice is commonly used in practice to speed up encryption. In contrast, for security issues, to avoid the small decryption exponent attack, decryption exponent can not be too small. d must have approximately the same size as modulus m, [7]. These choices determine the encryption and decryption time, so decryption time always will be much longer than encryption time. Several technics were developed to shorten the decryption time, one of them is using the Chinese remainder theorem. But even with these technics RSA encryption/decryption is much slower than the commonly used symmetric-key encryptions methods.

After that we generate two big prime numbers and calculate their product m. Now using the multiplicative inverse function we can calculate the private key d as an inverse of integer e modulo $\pmod{\phi(m)}$.

The Clean code that do this is the following:

```
privatk :: BigInt BigInt BigInt -> BigInt
privatk e p q
    #pq = (p - my_one)*(q - my_one)
    |gcd e pq <> my_one = abort "not relative prime"
    = meuclid e pq
```

To encrypt a number x, where $1 < x < (m-1)$ and $gcd(x, m) = 1$ we can do one modular exponentiation: $c = x^e \pmod{m}$. So the magnitude of the modulus m determines the magnitude of the number that we can encrypt at one go, and in the same time basically determines the running time of the application.

The Clean code that do this is the following:

```
rsacrypt :: BigInt BigInt BigInt -> BigInt
rsacrypt x e m = mexp x e m
```

For decryption we do the same computation, the differences consist in the value of actual parameter, we use the decryption exponent d instead of e.

3.6 File I/O

Because the Clean is a pure functional language the destructive updates are not admissible, but when we dealt with a file I/O we must to have the possibility to destructively update the file. In Clean this situation was resolved by introducing a new type, for which we use the * notation in the type name. With this type we can restrict the references of some date structure, and when the reference is unique, the update of date structure is allowed [5]. Using this technics the file I/O is quite simple in Clean.

To obtain more security in file encryption, several block cipher technics can be use. The most common block ciphers are the ECB (electronic codebook), CBC (cipher-block chaining), CFB (cipher feedback) and OFB (output feedback) modes. In our implementation we use the ECB mode, and we use the same key for every block encryption/decryption.

As long as we want to use the RSA file encryption as a block cipher we must set k, the size of the blocks. We assume that the plain text is a binary file so the alphabet size that we are using is 256, corresponding to possible byte values, thus the size of a block is $\log_{256} m$, where m is the RSA modulus.

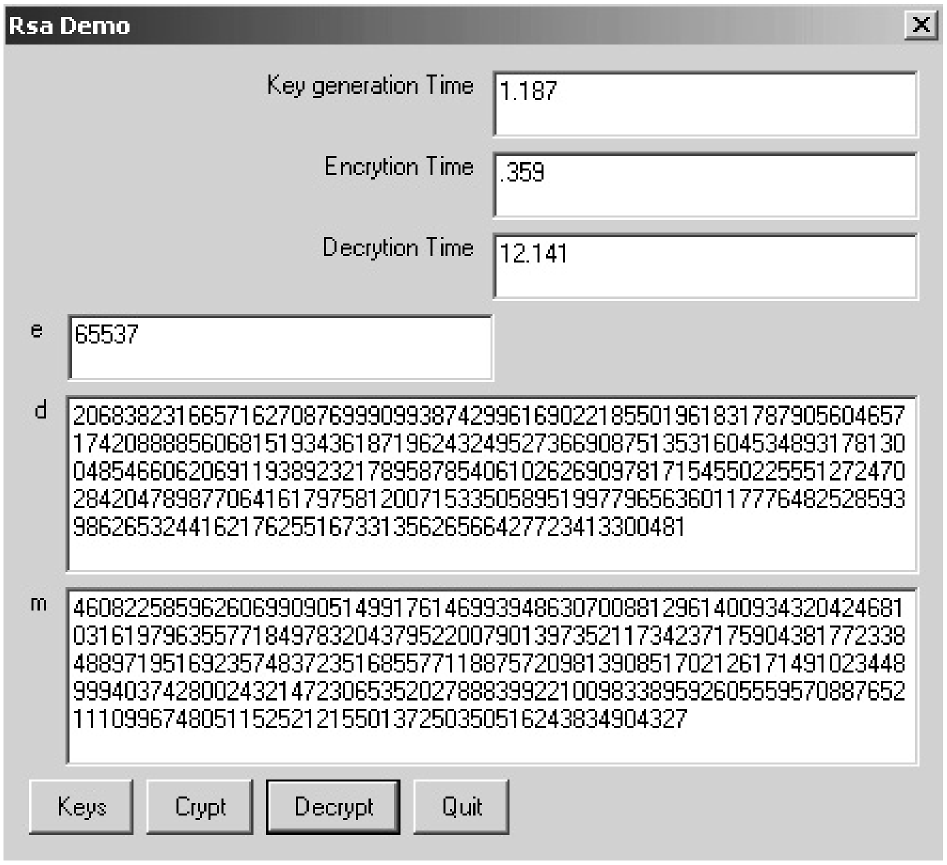
Now we present the function *filecrypt* which has the role: testing if we are at the end of the plain text file; reading $k-1$ bytes from this file; calculating $k2$, the number of bytes that were effectively read; converting these bytes from base p^k to base p; encrypting the result; converting the result to base p^{k+1}; writing in an encrypted file the number $k2$; writing the $k+1$ bytes in an output file, that will be the encrypted file.

The Clean code that do this is the following:

```
filecrypt :: *File *File (BigInt, BigInt) -> (*File, *File)
filecrypt inf outf (e, n)
    #! (atEnd, inf) = fend inf
    |atEnd = (inf, outf)
    # k = nrblok n alph
    # (inf, res) = mread (k-my_one) inf
    # (k2, cr) = filecrypt‘ res k e n
    # outf = fwritec (toChar k2) outf
    # outf = mwrite cr outf
    = filecrypt inf outf (e, n)

filecrypt‘ :: [Char] BigInt BigInt BigInt -> (Int, [Char])
filecrypt‘ res k e n
    #k2 = length res
    #nr = myconvert res alph (toBigInt k2)
    #scr = rsacrypt nr e n
    #cr = myconvert1 scr alph (k+my_one)
    = (k2, cr)
```

The function *mread* used in above code has the role to read a given number of bytes from a file while the function *mwrite* has the role to write a list of bytes in a file.



Key generation Time 1.187

Encrytion Time .359

Decrytion Time 12.141

e 65537

d
```
20683823166571627087699909938742996169022185501961831787905604657
17420888856068151934361871962432495273669087513531604534893178130
04854660620691193892321789587854061026269097817154550225551272470
28420478987706416179758120071533505895199779656360117776482528593
986265324416217625516733135626566427723413300481
```

m
```
46082258596260699090514991761469939486307008812961400934320424681
03161979635577184978320437952200790139735211734237175904381772338
48897195169235748372351685577118875720981390851702126171491023448
99940374280024321472306535202788839922100983389592605559570887652
111099674805115252121550137250350516243834904327
```

Figure 1: The graphical interface

3.7 The graphical interface

In Clean using the Object I/O library we can write flexible, platform independent programs, with a well designed graphical user interface, [1]. For obtaining an easy method to manipulate our input/output, such as generating public/secret key; showing the key generation time; selecting input/output file; doing the encryption/decryption; showing the encryption/decrytion time we design a graphical interface with different dialog items such as button controls, edit controls, text controls.

Our application graphical interface, for a certain jpg files with 48.9 KB size, and for 1024 bit key size have the following appearance, where we made out the value of public, secret keys (do this only for testing case), the key generation time for these values in seconds, and the encryption/decryption times in seconds are given in Fig. 1.

To measure the running time of certain function we got the computer tick, and determined the difference between two ticks. For this we have the following built in functions:

```
getCurrentTick :: !*env -> (!Tick, !*env)
tickDifference :: !Tick !Tick -> Int
```

To measure the performance of an application we have another possibility that Clean offers: we can enable Time Profiling option insight in the Clean environment, which means that after the execution the program will write a profile file. For our application the generated profile file is given in Fig. 2.

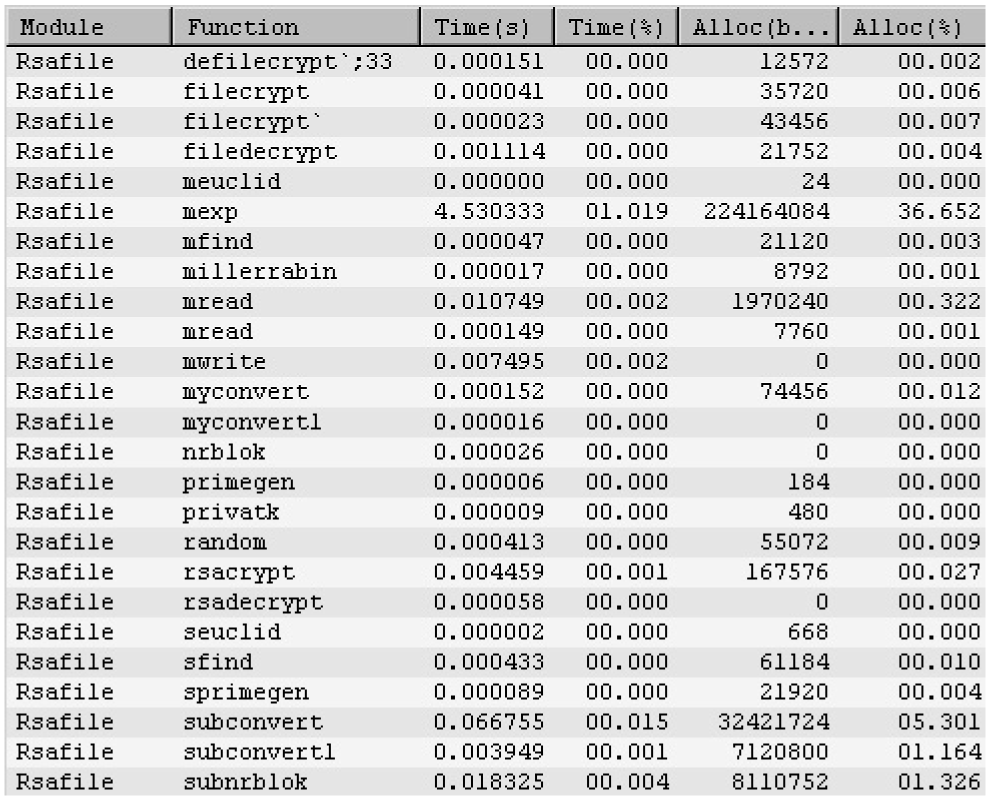
This profile file is overwritten after every program execution and will consist the real time measurements of each function in seconds, the number of bytes allocated in the heap by the function an so on.

4 Conclusions

Most of cryptography applications assure security on hardware and software level too. The main disadvantage using functional programming in cryptography is that applications written in functional programming language can guarantee security principally on software level. This fact constrains the applicability of functional programming language in area of cryptography. Another reason withdrawal in usage of functional programming in range of security is that the community of programmers who use functional programming language for building their cryptography's software is relatively small, and the available documentation is still very scarce.

Our program vulnerability is the usage of linear congruential generator for generating random number, which is known as not very safe. But our purpose not was to study the complex area of pseudorandom number generator algorithms.

Our experience shows that cryptography algorithms coded in a functional programming language are much shorter than those coded in C, Java, Maple. It is relatively easy to test the functional programming function independently from the entire program. The syntax "forces" the programmer to write more modular codes, so it is simple to locate and correct errors in these modules. Type errors are much easier to prevent and in case, to correct. We can easy



Module	Function	Time(s)	Time(%)	Alloc(b...	Alloc(%)
Rsafile	defilecrypt`;33	0.000151	00.000	12572	00.002
Rsafile	filecrypt	0.000041	00.000	35720	00.006
Rsafile	filecrypt`	0.000023	00.000	43456	00.007
Rsafile	filedecrypt	0.001114	00.000	21752	00.004
Rsafile	meuclid	0.000000	00.000	24	00.000
Rsafile	mexp	4.530333	01.019	224164084	36.652
Rsafile	mfind	0.000047	00.000	21120	00.003
Rsafile	millerrabin	0.000017	00.000	8792	00.001
Rsafile	mread	0.010749	00.002	1970240	00.322
Rsafile	mread	0.000149	00.000	7760	00.001
Rsafile	mwrite	0.007495	00.002	0	00.000
Rsafile	myconvert	0.000152	00.000	74456	00.012
Rsafile	myconvertl	0.000016	00.000	0	00.000
Rsafile	nrblok	0.000026	00.000	0	00.000
Rsafile	primegen	0.000006	00.000	184	00.000
Rsafile	privatk	0.000009	00.000	480	00.000
Rsafile	random	0.000413	00.000	55072	00.009
Rsafile	rsacrypt	0.004459	00.001	167576	00.027
Rsafile	rsadecrypt	0.000058	00.000	0	00.000
Rsafile	seuclid	0.000002	00.000	668	00.000
Rsafile	sfind	0.000433	00.000	61184	00.010
Rsafile	sprimegen	0.000089	00.000	21920	00.004
Rsafile	subconvert	0.066755	00.015	32421724	05.301
Rsafile	subconvertl	0.003949	00.001	7120800	01.164
Rsafile	subnrblok	0.018325	00.004	8110752	01.326

Figure 2: The content of profile file

issue efficiency in time and space. The process of the file I/O are relatively simple. It has a built-in library for large numbers, which means that working with large numbers becomes quite simple. The concept and syntax are permitting the correct use of necessary mathematics, so the programmer can focus on these, and not loosing time on circumstance of implementations. For those who know functional language it is easy to read and understand Clean code.

Taking account of our algorithms time consuming and correctness, and considered the capacity of software package presented in the first section we can establish that the usage of functional programming language can't be considered inconvenient. So we can conclude that functional programming is a very useful tool to write stable an efficient cryptography applications.

References

[1] P. Achten, M. Wierichurl, *A tutorial to the Clean object I/O library*, Technical Report CSI-R0003, February 2000, University of Nijmegen, 294 pag. ⇒109

[2] A. Bege, Z. Kása, *Algoritmikus kombinatorika és számelmélet* (Algorithmic combinatorics and number theory), Presa Universitară Clujeană (Cluj University Press), 2006. ⇒103

[3] Cryptol, http://www.galois.com/technology/communications_security/cryptol ⇒ 101

[4] D. Hofheinz, *Public key encryption from a mathematical perspective*, International School on Mathematical Cryptology, Barcelona, 2008. ⇒101

[5] P. Koopman, R. Plasmeijer, M. Van Eekelen, S. Smetsers, *Functional programming in CLEAN,* http://clean.cs.ru.nl/contents/Clean_Book/clean_book.html ⇒103, 107

[6] Gy. Márton, *Kriptográfiai alapismeretek* (Cryptography basics), Scientia Kiadó (Scientia Publishing House), Kolozsvár, 2008. ⇒102, 106

[7] A. J. Menezes, P. C. van Oorschot, S. A. Vanstone, *Handbook of applied cryptography*, CRC Press, Boca Raton, Florida, 1997. ⇒107

[8] R. L. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public key cryptosystems, *Comm. ACM*, **21,** 2 (1978) 120–126. ⇒102

[9] K. Rosen, *Elementary number theory and its applications*, Addison-Wesley, 2005. ⇒104, 105

[10] A. Salomaa, *Public-key cryptography*, Springer, 1996. ⇒101, 102

[11] D. Sankel, RSA-Haskell, http://www.netsuperbrain.com/netsuperbrain.html ⇒101

[12] H. van Tilborg, *Encyclopedia of cryptography and security*, Springer, 2005. ⇒101

Received: December 5, 2009 • Revised: February 23, 2010

Acta Universitatis Sapientiae

The scientific journal of Sapientia University publishes original papers and surveys in several areas of sciences written in English.
Information about each series can be found at
`http://www.acta.sapientia.ro`.

Acta Universitatis Sapientiae, Informatica

Contact address and subscription:
Acta Universitatis Sapientiae, Informatica
RO 400112 Cluj-Napoca
Str. Matei Corvin nr. 4.
Email: acta-inf@acta.sapientia.ro

This volume contains two issues

Sapientia University

Scientia Publishing House

ISSN 1844-6086
http://www.acta.sapientia.ro

Information for authors

Acta Universitatis Sapientiae, Informatica publishes original papers and surveys in various fields of Computer Science. All papers are peer-reviewed.

Papers published in current and previous volumes can be found in Portable Document Format (pdf) form at the address: `http://www.acta.sapientia.ro`.

The submitted papers should not be considered for publication by other journals. The corresponding author is responsible for obtaining the permission of coauthors and of the authorities of institutes, if needed, for publication, the Editorial Board is disclaiming any responsibility.

Submission must be made by email (`acta-inf@acta.sapientia.ro`) only, using the LaTeX style and sample file at the address: `http://www.acta.sapientia.ro`. Beside the LaTeX source a pdf format of the paper is needed too.

Prepare your paper carefully, including keywords, ACM Computing Classification System codes (`http://www.acm.org/about/class/1998`) and AMS Mathematics Subject Classification codes (`http://www.ams.org/msc/`).

References should be listed alphabetically using the following examples:

For papers in journals:
A. Hajnal, V. T. Sós, Paul Erdős is seventy, *J. Graph Theory*, **7,** 4 (1983) 391–393.

For books:
D. Stanton, D. White, *Constructive combinatorics*, Springer, New York, 1986.

For papers in contributed volumes:
Z. Csörnyei, Compilers, in: *Algorithms of informatics, Vol. 1. Foundations* (ed. A. Iványi), mondAt Kiadó, Budapest, 2007, pp. 80–130.

For papers in conference volumes:
C. Xiao, W. Wang, X. Lin, J. X. Yu, Efficient similarity joins for near duplicate detection, *Proc. 17th International Conference on World Wide Web*, Beijing, China, 2008, pp. 131–140.

For internet sources:
E. Ferrand, An analogue of the Thue-Morse sequence, *Electron. J. Comb.* **14** (2007) #R30, http://www.combinatorics.org/.

Illustrations should be given in Encapsulated Postscript (eps) format.

One issue is offered each author free of charge. No reprints will be available.

Publication supported by