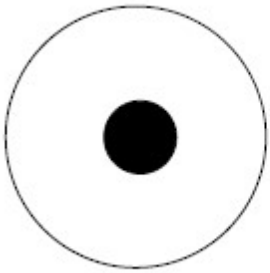
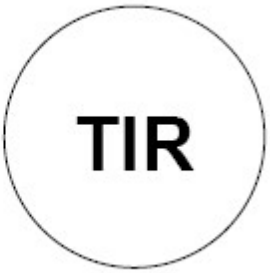




The IPSI Transactions on Internet Research

Multi-, Inter-, and Trans-disciplinary Issues in Computer Science and Engineering

A publication of IPSI Internet Research Society New York, Frankfurt, Tokyo, Belgrade
July 2020 Volume 16 Number 2 (ISSN 1820-4503)

Special issue: „Selected Topics in Computer Science“

Guest Editor: William Steingartner

Table of Contents:

The IPSI Internet Research Society

The IPSI Internet Research Society is an association of people with professional interest in the field of the Internet. All members will receive IPSI Transaction Journals upon registering at the Society.

Member copies of Transactions are for personal use only
IPSI TRANSACTIONS ON INTERNET RESEARCH
www.ipsitransactions.org

Editorial Board

Editorial

Selected Topics in Computer Science

William Steingartner, Guest Editor
Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia

This volume on Selected Topics in Computer Science contains ten refereed and extended papers written by authors of the most interesting and promising results presented during the 2019 IEEE 15th International Scientific Conference on Informatics. This conference was held in Poprad, Slovakia, November 20-22, 2019. It became a significant international forum for academic scientists, engineers, researchers and young IT experts together to exchange and share their experiences and research results about most aspects of science and social research, discuss the practical challenges encountered and the solutions adopted and bringing new ideas.

It is my pleasure to present the research works of authors who were awarded by the conference committee and allowed to publish their new scientific results in this selection. The topics of this volume (even the conference) cover theoretical and practical results, along with methods for transferring these research results into real-life domains, by scientist and experts working in computer science and computing-related fields.

Papers in this volume are grouped into four main topics: Theoretical Informatics, Software Engineering, Applied Informatics and Artificial Intelligence.

Papers focused on research in Theoretical Informatics are the following. The first paper **Mathematical Model Checking Based on Semantics and SMT** by the authors Wolfgang Schreiner and Franz-Xaver Reichl reports on the usage and implementation of RISCAL, a model checker for mathematical theories and algorithms based on a variant of first-order logic with finite models, and describes the semantics-based implementation of the checker as well as a recently developed alternative based on SMT solving, and experimentally compares their performance. Furthermore, authors report on their experience with RISCAL for enhancing education in computer science and mathematics, in particular in academic courses on logic, formal methods, and formal modeling.
The second paper **Software support for visualizing of the graph algorithms in a novel approach in educating of young IT experts** by authors Karina Mocinecová and William Steingartner presents a software module which provides simulation and dynamic visualization (algorithm animation) of selected graph algorithms and its role in the education of young IT experts.
The next paper **Totally bounded metric spaces, their model theoretic stability and similarity detecting algorithms** written by authors Gábor Sági and Karrar Al-Sabti investigates model-theoretic properties of certain metric spaces. Based on those results, authors present how to build some metric spaces from their finite subspaces. Furthermore, authors present a similarity detecting algorithm for a finite-dimensional Euclidean space and totally bounded metric space and they discuss an analysis of the complexity of the algorithm.
In the last paper in this topic **Classification rules for temporal databases,** authors deal with temporal architectures highlighting granularity and they propose own complex classification rules to determine individual access principles, technology, architecture, and security aspects whereby the classification consists of 14 evaluated aspects. Furthermore, a discussion about the undefined states and reliability provided by the consistency is done, as well.

Software engineering topic includes the following papers. The paper **Evaluation of Static Analysis Methods of Python Programs** by authors Hristina Gulabovska and Zoltán Porkoláb focuses on static analysis methods currently applied for Python. Authors investigate the advantages and shortages of those methods and highlight the restrictions of current tools and suggest further research directions to tackle these problems. Furthermore, they report their experiences applying static analysis methods on an open-source Python software system where they found numerous issues confirmed by the developers. Based on these findings, authors suggest refined configuration settings on static analysis tools.
The next paper **Advanced Code Comprehension using Version Control Information** presented by authors Tibor Brunner and Zoltán Porkoláb the authors investigate how a version control information of the project can be utilized for extending the apprehension of large legacy systems providing a better understanding of the software under examination. Authors show that some of the hidden structural connections between the elements of the program can be revealed most easily by the development history of the system.
The third paper in this topic **The Automated Creation of Logical Constructions from Natural Language Sentences** written by authors Zuzana Bilanová and Martin Štancel describes the theoretical design and implementation of a prototype semantic machine of transparent intensional logic. Procedural semantics of this logic causes that it can be used in the field of logical analysis of natural language. The semantic machine described in this paper makes it possible to analyze sentences in natural language in the first two of the above steps, for the mentioned logical system. Furthermore, authors discuss the possibilities of the presented solution are compared with the already existing semantic machines processing the meaning of natural language sentences into intensional constructions.

The topic of Applied Informatics includes two interesting papers. The first one **Automated hardening of a Linux web server** by authors Ján Perháč and Michal Olenčin is focused on design and implementation of automated hardening a Linux web server after the clean installation. The article deals with the analysis of the cybersecurity, focus, and scope of the configuration. Based on that, the design of an automated security configuration with a detailed description was developed. The second paper **Edge coloring of set of graphs with the use of data decomposition and clustering** written by authors Adam Dudáš and Jarmila Škrinárová presents a solution to the problem of parallel edge coloring of sizable sets of cubic graphs. Authors present a design and implementation of a methodology based on the clustering of graphs and data decomposition model. The methodology aims to satisfy time criterium of effective decomposition of problem to set of subproblems.

The paper **Experiments on Rough Sets Clustering with Various Similarity Measures** written by author group Arnold Szederjesi-Dragomir, Radu D. Gaceanu, Horia F. Pop and Costel Sarbu as the only paper in topic Artificial Intelligence is aimed to investigate the influence of several similarity measures in a rough sets clustering context and, in this sense, datasets with overlapping regions are of particular interest. Authors are also interested in the way the overlapping detection mechanism is affected by the chosen similarity measure and because the standard datasets do not provide specific information about overlapping regions, they also propose an approach to obtain such data for benchmarking purposes. The importance of properly choosing the similarity measure is outlined by the experiments carried out on standard datasets.

This volume focuses on actual problems in Computer Science and related fields, and it presents its both theoretical and practical results and applications thus exploiting the

innovative approaches. It reflects the main thrust of research in Computer Science and related fields and establishes contacts with the experts of these fields.

I would like to express my gratitude and appreciation to the authors for contributing their important research results in the journal. Due to their fruitful research and helpful cooperation, it was possible to provide such broad coverage of these areas of scientific research.

I also would like to thank all reviewers. They have worked very hard in reviewing papers and making valuable suggestions for the authors to improve their works.

Mathematical Model Checking Based on Semantics and SMT

Schreiner, Wolfgang; Reichl; Franz-Xaver

Abstract: ***We report on the usage and implementation of RISCAL, a model checker for mathematical theories and algorithms based on a variant of first-order logic with finite models; this allows to automatically decide the validity of all formulas and to verify the correctness of all algorithms specified by such formulas. We describe the semantics-based implementation of the checker as well as a recently developed alternative based on SMT solving, and experimentally compare their performance. Furthermore, we report on our experience with RISCAL for enhancing education in computer science and mathematics, in particular in academic courses on logic, formal methods, and formal modeling. By the use of this software, students are encouraged to actively engage with the course material by solving concrete problems where the correctness of a solution is automatically checked; if a solution is not correct or the student gets stuck, the software provides additional insight and hints that aid the student towards the desired result.***

Index Terms: ***model checking, logic, semantics, formal verification, reasoning about programs, computer science education***

1. Introduction

SOFTWARE based on formal logic plays an ever-increasing role in areas where a mathematically precise understanding of a subject domain and sound rules for reasoning about the properties of this domain are essential. A prime example is the formal modeling, specification, and verification of computer programs and computing systems, but there are many other applications in areas such as knowledge-based systems, computer mathematics, or the semantic web.

As for reasoning in these domains, there are two main approaches, *proving* and *model-checking*. The main advantage of proof-based systems [6], [9] is their generality: they can operate with rich formal systems such as first-order logic and reason about domains of infinite size. Their disadvantage, however, is that such rich logics are generally undecidable such that the failure to construct a proof for a conjecture does not indicate its invalidity. This is a major problem in areas such as program verification where these conjectures are verification conditions derived not only from the specifications of the program but also from extra (in practice human-provided) information such as loop invariants; if these invariants are too strong or too weak, the verification conditions are not valid, even if the program satisfies its specification.

The main advantage of model checkers [2] is their automatism: they decide without user assistance the validity of conjectures; furthermore, if a conjecture is invalid, they produce a counterexample that demonstrates its invalidity. Their disadvantage is that they have to be based on decidable calculi, which usually entails finite domains of interpretation or weaker formal systems than first-order logic. For instance, bounded model checkers consider only finite (prefixes of) program runs and typically only check for specific properties such as preconditions of built-in operations.

In this paper, we present an approach that combines the generality of a rich logic with the automation of model checking. This approach is implemented in *RISCAL* [11], [12], [16], a software system for the formalization of mathematical theories and the specification and verification of algorithms operating in such theories. The software also provides various means to aid the understanding of results, e.g., by producing counterexamples or by the graphical visualization of evaluation trees [15]. The checking mechanism is based on an executable version of the denotational semantics of the specification language; recently we have developed an alternative approach based on the translation of this language to the language SMT-LIB supported by various satisfiability modulo theories solvers [10].

Various other modeling languages also support checking but differ from RISCAL in the application domain of the language or the exhaustiveness of the checks. Alloy [4] is based on a relational logic designed for modeling abstract systems but little suitable for mathematical theories. TLA$^+$ [5] embeds first-order logic but is untyped and also supports models of infinite size; thus, the TLA$^+$ toolkit does not really represent a reliable decision procedure. In contrast, RISCAL is based on full first-order logic and also allows, e.g., implicitly defined

Manuscript received June 8, 2020. This work was supported by the Johannes Kepler University Linz, Linz Institute of Technology, Project LOGTECHEDU "Logic Technology for Computer Science Education" and by the OEAD WTZ project SK 14/2018 SemTech.

W. Schreiner (contact person) is an associate professor at the Research Institute for Symbolic Computation of the Johannes Kepler University Linz, Austria (e-mail: Wolfgang.Schreiner@risc.jku.at).

F.-X. Reichl is a student of computer mathematics at the Johannes Kepler University, Linz, Austria (e-mail: franz.x.reichl@gmail.com).

functions and nondeterministic computations; still by its restriction to models of (parametrizable) finite size, the validity of formulas and the correctness of algorithms is fully decidable [13].

A main application of RISCAL is in educational scenarios where questions and problem statements have a precise and machine-understandable meaning and where the correctness of answers and problem solutions can be automatically checked. The use of this software gives students the possibility to self-check the correctness of their solutions and to use the feedback of the software to correct their errors. The goal is a style of "self-directed learning" where students do not just passively consume educational material but actively interact with it by producing problem solutions with the help of software. Ultimately such software might also be integrated into software for the automatic grading of assignments [20].

In this paper, we describe our actual experience with the use of RISCAL in academic courses on formal specification and verification, formal modeling, and logic; these courses have addressed various types of audiences, computer scientists as well as mathematicians, from absolute beginners to master students in the later phases of their education. RISCAL has been partially developed in the context of two projects, "Logic Technologies for Computer Science Education" (LogTechEdu) at Johannes Kepler University (JKU) Linz [7], and "SemTech" at JKU Linz and the Technical University (TU) of Košice [17], that investigate and further develop such tools for their use in computer science education. An example for other software developed in these projects is the toolset Jane [18], [19] that illustrates graphically the formal semantics of a simple programming language and that has been applied in various courses.

The remainder of this paper is structured as follows: In Section 2 we sketch the use of RISCAL on small examples. Section 3 discusses the semantics-based implementation of the RISCAL checker, the alternative method based on SMT solving, and their experimental comparison. In Section 4 we describe our experience with the use of RISCAL in education. Section 5 presents our conclusions and outlines further work.

This paper is a revised version of [14] extended by previously unpublished material (Section 3).

2. The RISCAL Software

RISCAL (RISC Algorithm Language) is a language and associated software system for formulating theories in first-order logic, describing algorithms in a high-level language, and specifying the behavior of these algorithms by formal contracts. The language is based on a type system where all types have finite sizes (specified by the user); this allows to fully automatically decide formulas and to verify the correctness of algorithms for all possible inputs. To this end, the system translates every syntactic phrase into an executable form of its denotational semantics; the RISCAL model checker evaluates this semantics to determine the results of algorithms and the truth values of formulas such as the postconditions of algorithms. Figure 1 displays the user interface of the software with an editor panel on the left and control elements and an output terminal on the right.

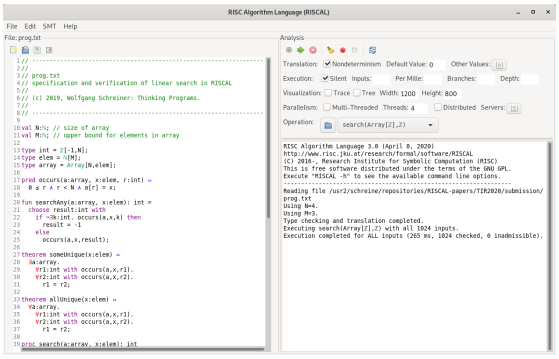


Fig. 1. The RISCAL Graphical User Interface

As an example for the use of RISCAL, take the following constant and type declarations:

```
val N:ℕ; val M:ℕ;
type int = ℤ[-1,N]; type elem = ℕ[M];
type array = Array[N,elem];
```

These introduce a type $array$ of arrays that have some length N and elements of type $elem$; each such element is a natural number up to some maximum M. Likewise an auxiliary type int of integers from -1 to N is defined. Furthermore, we introduce by the definition

```
pred occurs(a:array, x:elem, r:int) ⇔
  0 ≤ r ∧ r < N ∧ a[r] = x;
```

a predicate $occurs$ that is true if array a holds element x at position r. From this, the declaration

```
fun searchAny(a:array, x:elem): int =
  choose result:int with
    if ¬∃k:int. occurs(a,x,k) then
      result = -1
    else
      occurs(a,x,result);
```

introduces an implicitly defined function $searchAny$ that returns an arbitrary position at which x occurs in a, respectively the value -1, if x does not occur in a. RISCAL can execute this definition, e.g., for $N = 4$ and $M = 3$; if we select for this execution the "nondeterministic" mode, for every input *all* possible outputs are determined:

```
Executing searchAny(Array[ℤ],ℤ) with
  all 1024 inputs.
Branch 0:0 of nondeterministic function
  searchAny([0,0,0,0],0):
Result (0 ms): 0
...
Branch 4:1023 of nondeterministic ...
  searchAny([3,3,3,3],3):
No more results (15 ms).
```

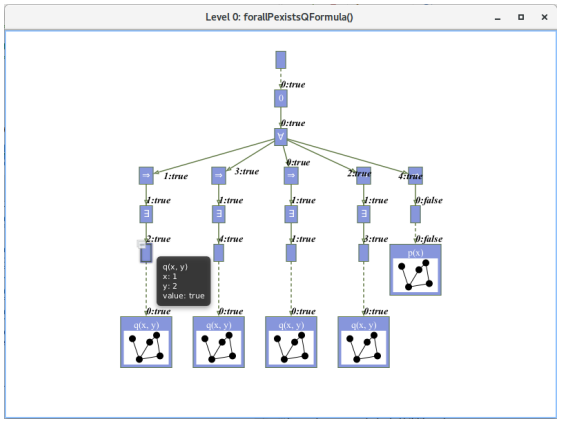


Fig. 2. A Pruned Formula Evaluation Tree in RISCAL

```
Execution completed for ALL inputs
  (6741 ms, 1024 checked, ...).
```

This is possible, because in nondeterministic mode, the semantics of functions, predicates, and procedures does not denote a single value but a finite set of values which is implemented by a lazily evaluated stream (see Section 3).

Based on the predicate $occurs$, we may define and check the (apparently valid) theorem that for every element x there is *some* array that holds x at a unique position:

```
theorem someUnique(x:elem) ⇔ ∃a:array.
  ∀r1:int with occurs(a,x,r1).
  ∀r2:int with occurs(a,x,r2). r1 = r2;
Executing someUnique(ℤ) with all 4 ...
Execution completed for ALL inputs ...
```

However, we may also define and check the (invalid) theorem that above is true for every array:

```
theorem allUnique(x:elem) ⇔ ∀a:array.
  ∀r1:int with occurs(a,x,r1).
  ∀r2:int with occurs(a,x,r2). r1 = r2;
Executing allUnique(ℤ) with all 4 ...
ERROR in execution of allUnique(0):
evaluation of
  allUnique
at line 33 in file prog.txt:
  theorem is not true
ERROR encountered in execution.
```

We may then ask for a counterexample:

```
Executing allUnique_refute().
This sequence of variable assignments
  leads to a counterexample ...:
x=0
a=[0,0,0,0]
r1=0
r2=1
Execution completed (5 ms).
```

The truth values of formulas can be also visualized in RISCAL by the help of "pruned evaluation trees" (see Figure 2 for an example of such a tree) that depict those paths in the evaluation of quantified formulas that determine the overall outcome.

After these preliminaries, we define a deterministic procedure that returns the *smallest* position r of element x in array a; if x does not occur in a, the procedure returns -1:

```
proc search(a:array, x:elem): int {
  var i:int = 0; var r:int = -1;
  while i < N ∧ r = -1 do {
    if a[i] = x
      then r := i;
      else i := i+1;
  }
  return r;
}
```

To verify its correctness, we annotate the procedure with the following postcondition:

```
ensures if ¬∃k:int. occurs(a,x,k) then
  result = -1
else occurs(a,x,result) ∧
  ∀k:int with occurs(a,x,k). result≤k;
```

We may then check the correctness of the algorithm for all possible inputs:

```
Executing search(...) with all 1024...
Execution completed for ALL inputs ...
```

However, if we replace the test $a[i] = x$ erroneously by $a[i] \neq x$, we get the following error:

```
ERROR in execution of search(...):
evaluation of ensures ... at line ...:
  postcondition is violated by result
    -1  for application search(...)
```

Furthermore, we may annotate the above loop by an invariant and termination measure:

```
invariant 0 ≤ i ∧ i ≤ N;
invariant
  ∀k:int. 0 ≤ k ∧ k < i ⇒ a[k] ≠ x;
invariant
  r = -1 ∨ (r = i ∧ i< N ∧ a[r] = x);
decreases if r = -1 then N-i else 0;
```

Every execution of the procedure checks the validity of these annotations; this in particular ensures that the given invariant is not too strong. However, we may also let the system generate from this invariant verification conditions whose validity implies the correctness of the program. These conditions are theorems that can be automatically checked in RISCAL by a single mouse-click (see Figure 3), which ensures that the specified invariants are strong enough (they are "inductive"). Thus, we may subsequently use some other environment to verify the correctness of the algorithm by formal proof for *arbitrary* values of M and N.

The big advantage of RISCAL is that it allows to formulate rich formal/mathematical/logical contents (theories and algorithms) in an expressive language (first order logic including expressions that do not necessarily have unique values) and still have their adequacy fully automatically checked over small domains. In this way, errors in the formulations can be easily caught and thus the formalism be quickly validated; this is not so simply possible with proof-based approaches where failed proof attempts more often than not indicate the inadequacy of proof strategies rather than the invalidity of proof goals. Only when we are after such a validation reasonably convinced about the

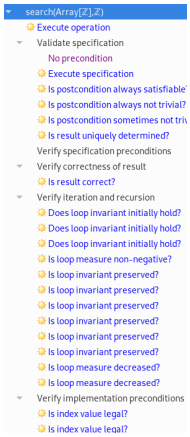


Fig. 3. The Validity of Verification Conditions in RISCAL

correctness of the formulations, we will turn to proof-based verification over general domains.

3. Checking and SMT Solving in RISCAL

The basic mechanism of RISCAL for checking the correctness of algorithms and theorems relies on the evaluation of their denotational semantics; however in [10] an alternative mechanism is presented which is based on the translation of RISCAL formulas to the language SMT-LIB supported by various satisfiability modulo theories (SMT) solvers. In this section, we sketch both approaches and compare their performance by a number of benchmarks.

3.1. Model Checking

RISCAL implements an executable form of the denotational semantics of every kind of phrase of its language. If a phrase is deterministic, i.e., its evaluation yields a uniquely defined result, its semantics is typically an element of the domain

$$Single[T] := Context \to T$$

i.e., it maps a context (which determines the values of the free variables of the phrase) to a value of the semantic domain T. However, a phrase may also be nondeterministic, i.e., its evaluation may yield multiple results. Then its semantics is typically an element of the domain

$$Multiple[T] := Context \to Seq[T]$$

where $Seq[T]$ is a collection of T-values which we will for the moment consider as a set (more details will be given below).

For instance, in the case of imperative commands we have $T := Context$, i.e., the result is another context (which denotes the values of the variables after the execution of the command); in the case of logic formulas we have $T := Bool$, i.e., the result is a truth value; in the case of terms we have $T := Value$, i.e., the result is a value of the object domain of the logic.

The denotation of a phrase of syntactic domain S is defined by a mapping $[\![.]\!] : S \to Single[T] + Multiple[T]$. For instance, assuming two deterministic commands C_1 and C_2, the deterministic semantics $[\![C_1;C_2]\!] \in Single[Context]$ of their composition can be defined as follows:

$$[\![C_1;C_2]\!] := \lambda c.\ \text{let}\ c_1 = [\![C_1]\!](c)\ \text{in}\ [\![C_2]\!](c_1)$$

However, if C_1 and C_2 are nondeterministic, then we have the nondeterministic semantics $[\![C_1;C_2]\!] \in Multiple[Context]$ defined as follows:

$$[\![C_1;C_2]\!] := \lambda c.\ \text{let}\ cs_1 = [\![C_1]\!](c)\ \text{in}\ \bigcup_{c_1 \in cs_1} [\![C_2]\!](c_1)$$

RISCAL implements these mathematical notions by corresponding constructions in Java. For instance, we have data types like

```
public interface Single<T> extends
  Function<Context,T> { }
public interface Multiple<T> extends
  Function<Context,Seq<T>> { }
```

The domain $Seq[T]$ is implemented as a class with an object method `get()` that returns either `null` (the end of the sequence) or a pair of a sequence element and another object of this class. Thus $Seq[T]$ actually denotes the domain of *lazily* evaluated sequences of T-values as the basis of the nondeterministic execution semantics. The implementation of the (deterministic/nondeterministic) semantics of phrases such as command sequences is based on functions like

```
static Single<Context>
seqCommand(Single<Context> C1,
           Single<Context> C2)
{ return (Context c) ->
    { Context c1 = C1.apply(c);
      return C2.apply(c1); } }
static Multiple<Context>
seqCommand(Multiple<Context> C1,
           Multiple<Context> C2)
{ return (Context c) ->
   { Seq<Context> cs1 = C1.apply(c);
     return cs1.mapJoin(C2); } }
```

The whole implementation heavily depends on the lambda expressions introduced in Java 8; in fact, the RISCAL implementation of the denotational semantics resembles very much a (higher-order) functional program.

RISCAL first builds from the concrete syntax of a phrase an abstract syntax tree; then it applies a type checker to annotate this tree with symbolic information determined by static analysis; then it translates the annotated tree to its denotational semantics; finally, it executes this semantics of the phrase. Since modern JIT compilers heavily optimize at runtime the executions of the functional objects resulting from this translation, we get an evaluation mechanism for the various phrases, which is much more efficient than classical “interpretation”; in fact, the translation mechanism

achieves many of the advantages we get by "compilation" to a low-level machine language.

If the phrase is a formula, we thus have (since every RISCAL type is finite) a decision procedure for the validity of the formula. If the phrase is a program, we have an execution mechanism for the program; if the program is annotated with meta-information such as preconditions, postconditions, loop invariants, and termination measures, we have a model checker for the correctness of the program. Since RISCAL also implements a verification condition generator where the derived verification conditions are again RISCAL formulas, we thus also have a (limited form of) a program verification environment. From RISCAL Version 3 on, the language also supports the concept of concurrent systems whose semantics are transition systems; the evaluation of these systems yields a model checker for the verification of the invariance of safety conditions.

3.2. SMT Solving

The application of SMT solvers in RISCAL, presented in [10], relies on the translation of a theorem (and of the specification on which the theorem depends) into an SMT-LIB [1] script. For this purpose, we use the SMT-LIB logic of Quantifier Free Formulas with Fixed Size Bit Vectors and Uninterpreted Functions (QF_UFBV). This idea of a translation is related to [8] where a translation of TLA$^+$ enabled the application of SMT solvers.

The translation ensures that the original RISCAL theorem is valid if and only if the generated SMT-LIB formula is unsatisfiable. This can be decided by a variety of SMT solvers supporting the QF_UFBV logic, in particular Boolector, CVC4, Yices 2, and Z3. As already discussed above, RISCAL provides the functionality to generate verification conditions for algorithms. Since these conditions are just theorems in RISCAL, they can be also decided by the translation to SMT-LIB.

In the remaining part of this section, we will discuss selected aspects of this translation which proceeds in two steps:

1) We translate the input into a form that makes the later encoding substantially easier and negate the given theorem.
2) We encode the preprocessed RISCAL specification into SMT-LIB such that satisfiability of the negated theorem is preserved.

The translation step, e.g., renames overloaded functions such that every function gets a unique name. We also eliminate several kinds of expressions by rewriting them into logically equivalent forms; in particular we replace "choose expressions" (nondeterministic choices) by applications of newly introduced "choice functions", whose interpretations are restricted by appropriate axioms. For example, the theorem:

```
theorem T(x:ℕ[N]) ⇔ x ≥
  choose y:ℕ[N] with x=2·y ∨ x=2·y+1;
```

is preprocessed to:

```
fun chooseFun(x:ℤ[0,N]):ℤ[0,N]
axiom chooseAx ⇔ ∀x:ℤ[0,N].
  (∃y:ℤ[0,N]. (((x = (2·y)) ∨
  (x=((2·y)+1))) ∧ (chooseFun(x)=y)));
theorem T ⇔ ∃x:ℕ[N].(x<chooseFun(x));
```

The encoding step performs two tasks:

1) We eliminate all universal and existential quantifiers because the target of our translation (QF_UFBV) is a quantifier-free logic.
   To eliminate existential quantifiers, we mainly apply the technique of "skolemization", i.e., we replace existentially quantified variables by applications of uninterpreted "Skolem functions". However, since the result is only an equi-satisfiable (not a logically equivalent) formula, this transformation is only possible within the negated theorem whose satisfiability is to be decided.
   As a more general technique, we replace universally and existentially quantified formulas by expanding them into conjunctions respectively disjunctions of instantiations of the body formula for all values of the type of the quantified variable.
2) We encode the types and operations of RISCAL in the theory of bit vectors. While this translation is generally based on commonly known representations, major adaptations were necessary, e.g., to deal appropriately with integer domains of varying sizes, and to devise effectively computable mappings between the RISCAL types and the bit vector domains.

In the following we illustrate the translation on the basis of the following RISCAL specification:

```
type nat=ℕ[100];
type set=Set[nat];
fun diff(A:set,B:set):set=(A\B)∪(B\A);
theorem assoc(a:set,b:set,c:set)⇔
  diff(diff(a,b),c)=
  diff(a,diff(b,c));
```

Here we introduce a type `set` of sets that have elements of type `nat`. Furthermore, we introduce with `diff` a function that represents the symmetric difference of two sets. Finally, we introduce the theorem `assoc` that states that the symmetric difference is associative (obviously this is a valid theorem). We translate this RISCAL specification to the following SMT-LIB script:

```
(set-logic QF_UFBV)
(define-sort nat ()(_ BitVec 7))
(define-sort set ()(_ BitVec 101))
(define-fun diff ((A (_ BitVec 101))
  (B (_ BitVec 101)))(_ BitVec 101)
  (bvor (bvand A (bvnot B))
  (bvand B (bvnot A))))
(declare-fun f ()(_ BitVec 101))
(declare-fun f_1 ()(_ BitVec 101))
(declare-fun f_2 ()(_ BitVec 101))
```

TABLE 1
SMT PERFORMANCE

Model	Theorem	Values	RISCAL	Yices
catlogic	Imp1	N=2, M=1	71748	92
sets	associativeUnionT	N=8	235413	2
gcd2	_gcdf_8_PostUnique	N=100	Timeout	55
bubble	_swap_0_CorrOp0	N=6, M=6	253559	6
sat3	_DPLL2_14_CorrOp0	n=2, cn=2	Timeout	10
matrices	symAdd	N=4	Timeout	4
graphs	_handshaking Theorem_8_PreSat	N=3	0	9682
sat3	_DPLL2_14_PostNot TrivialSome	n=3, cn=2	0	920
bubble	_bubbleSort3_2 _CorrOp0	N=4, M=4	10399	25695
search	_bsearchp_1_CorrOp3	N=4, M=4	205	4635

```
(assert (let ((a f))(let ((b f_1))
  (let ((c f_2))(distinct
  (diff (diff a b) c)
  (diff a (diff b c)))))))
(check-sat)(exit)
```

Here we introduce sorts `nat` and `set` that represent the same named RISCAL types as well as a function `diff` that represents the corresponding RISCAL function. In the definition of this function we can see that we represent the union of sets by bitwise disjunction and the set difference by means of bitwise negation and conjunction (as $A \setminus B = A \cap B^c$). Finally, we represent the negation of the theorem `assoc`. For this purpose, we first introduce three nullary functions for the skolemization of the quantified variables a, b, and c (as we negate the theorem, we have existentially quantified variables). In the `assert` statement we then state the negated theorem.

If we apply an SMT solver to this SMT-LIB script, it reports that the formula is unsatisfiable, which shows that the theorem `assoc` is valid. Indeed, the SMT solver Yices 2 reports a result almost instantaneously, whereas the basic checking mechanism of RISCAL does not produce an answer in a reasonable amount of time.

3.3. Experimental Comparison

In [10] we systematically compared the performance of the checking mechanisms presented in the previous two sections. For this purpose, we selected a variety of RISCAL specifications respectively theorems. On the one hand, we checked these theorems with the semantics-based mechanism of RISCAL. On the other hand, we applied the SMT-LIB translation and invoked the SMT solvers Boolector, CVC4, Yices 2, and Z3, on the generated scripts. Finally, we compared the obtained results. The conducted tests showed that in general Yices 2 [3] delivered the best results in terms of performance. Thus, here we only present the results of the tests performed with Yices 2.

In Table 1 we give a (small) selection of these results achieved with checking/deciding a variety of theorems that are all valid and that cover a variety of types, e.g., integers, maps/arrays, and sets. The entries of this table are to be read in the following way. Column 1 gives the name of the tested RISCAL specification. Column 2 gives the tested theorem. Column 3 gives the used model parameters. Column 4 gives the time needed by the semantics-based checking mechanism of RISCAL. Column 5 gives the time needed by the translation and the SMT solver Yices 2 to decide the given theorem. All figures represent wall clock times in milliseconds, measured on an Intel Xeon Gold 6128 processor with 1.5 TB of memory. A "timeout" is reported if the checking time exceeded a threshold of twenty minutes. The used RISCAL specifications and the generated SMT-LIB scripts are publicly available at https://www.risc.jku.at/research/formal/software/RISCAL/papers/thesis-Reichl.tgz.

The first six examples of Table 1 illustrate the speedups that can be achieved through the usage of SMT solvers. We want to point out that the tests presented in [10] indicate that such speedups cannot only be achieved for few selected cases, but for a broad variety of RISCAL specifications and theorems. Indeed, in approximately 75% of the tests, the translation used together with the application of Yices 2 was substantially faster than the semantics-based mechanism of RISCAL.

However, while often significant speedups can be achieved by the usage of SMT solvers, there are also cases where the semantics-based checking mechanism delivers superior results. We detected certain patterns in RISCAL specifications that seem to have a negative influence on the performance of the SMT based checking mechanism. The last four examples of Table 1 illustrate two such patterns that may make the usage of the translation disadvantageous.

Examples 7 and 8 illustrate that theorems that contain existentially quantified formulas may be checked significantly faster by the semantics-based method. There are, in particular, two reasons for this. On the one hand, RISCAL can often find a witness for an existential quantifier very fast. Thus, it can often report the validity of theorem very soon — like in the mentioned examples. On the other hand, the translation has to negate the theorems. Thus, existential quantifiers are transformed to universal quantifiers. But universal quantifiers have to be expanded, which can result in large formulas that seem to be disadvantageous for all the tested SMT solvers.

Examples 9 and 10 illustrate that RISCAL specifications with "choose expressions" may be disadvantageous for the application of SMT solvers. To justify this assertion, we have to consider that such expressions are translated to applications of uninterpreted functions. This, for example, means that `choose x:T`, where T is some type, is represented by a function, whose image is the set of

bit vectors that represent the type T. Often not all the bit vectors from the image of such a function correspond to a value that can actually be attained by the original choice. Therefore, it is necessary to use certain assertions that guarantee that the function only attains values that correspond to values that can be given by the choice. These assertions often seem to have a negative influence on the performance of the SMT solvers.

To sum up, the comparison of the two decision mechanisms shows that the SMT based mechanism has the potential to significantly speedup the decisions of the validity of formulas. Still, it cannot fully replace the semantics-based checking mechanism of RISCAL, as certain specification patterns favor the usage of this mechanism.

4. Logic and Semantic Software for Education

The intent of our projects LogTechEdu and SemTech is to further advance education in computer science and related topics: by utilizing the power of modern software based on formal logic and semantics, students shall engage with the material they encounter by actively producing problem solutions rather than just passively consuming them from the lecturer. For this purpose, we have experimented with various pieces of related software respectively further developed such software. For instance, David Cerna has in the frame of LogTechEdu at JKU Linz recently developed an Android app "AXolotl" for the touch-based training of first-order reasoning with term matching respectively substitution; likewise, William Steingartner and Valerie Novitzká have in the frame of SemTech at TU Košice developed the toolset "Jane" for the semantics-based execution and visualization of a simple procedural language that has been successfully employed in various courses on programming language semantics.

In this paper we will discuss in more detail those activities that the first author has been directly connected to, mostly related to the RISCAL software that was described in the previous sections.

a) Formal Specification and Verification: Since 2005, the first author has given at JKU Linz a yearly 4.5 ECTS course on "Formal Methods in Software Development"; the goal is to educate master students in computer science and computer mathematics in the formal specification and verification of computer programs with the help of various freely available software environments. Since 2009 we have also used our own "RISC ProofNavigator" and since 2011 our "RISC ProgramExplorer" for the purpose of verifying programs by deriving and proving verification conditions. In this course groups are of moderate size (about 25 participants) and have already considerable technical background (their formal background, however, is varying).

However, by relying on proof-based verification tools, the adequacy of formal specifications and annotations (loop invariants) could be only judged by proving the validity of the generated verification conditions. If such proofs did not immediately succeed (after applying some standard interactions with the respective proof assistants), many students were not really able to deduce from the failed proof attempt whether this was due to an inadequate proof strategy or due to deficiencies in the specifications/annotations; moreover, sometimes verification attempts trivially succeeded because preconditions were unsatisfiable or postconditions were generally valid.

In 2017, we introduced RISCAL into the course, replacing some of the initial use of the RISC ProofNavigator/ProgramExplorer. In a first exercise students had to validate specifications with the help of various techniques integrated into RISCAL, such as executing implicitly defined functions that were automatically generated from procedure contracts or checking whether pre- and postconditions satisfied various consistency criteria. In later exercises, students had to annotate procedures with invariants and termination measures, and check the verification conditions generated by RISCAL. If conditions were not valid, the RISCAL trace/visualization features could be applied to determine the sources of the errors.

Final anonymous evaluations of the course software performed in 2018 and 2019 indicated a very high satisfaction of students with RISCAL concerning its ease of use and learning success; the ratings were significantly better than for the proof-based RISC ProofNavigator/ProgramExplorer and summarily higher than for the six other toolsets used in the course (the extended static checker ESC/Java2 being the second most popular one). However, there was no objectively visible effect on exercise grades, which remained mostly in the 80–95% (good to very good) range.

Also outside of JKU Linz, at the Czech Technical University in Prague, Stefan Ratschan used in 2019 RISCAL in a 4.5 ECTS course on "Formal Methods and Specification" for 80 students. The software was applied with very good success; apart from some feedback for improving usability and requests for additional features, no problems were reported; in 2020, RISCAL has been employed in this course again.

Apart from courses, two bachelor students of technical mathematics used RISCAL to elaborate in their bachelor theses the formal specification and verification of algorithms from discrete mathematics (mostly relating to set and graph theory) respectively for searching and sorting of sequences in various representations (including the major asymptotically fast algorithms). Especially in the latter case, it was astonishing to see that the stu-

dent, without prior expertise in formal verification, was able to come up with sufficiently strong loop invariants to let the verification succeed. These are (anecdotal but nevertheless) strong indications that students that already have a certain background are indeed able to develop formally adequate theories and specifications.

b) Formal Modeling: In 2019, at JKU a new 3 ECTS course "Formal Modeling" for bachelor students of technical mathematics was introduced, as well as an accompanying proseminar. The course was given in three modules by three lecturers; in the module "Logic Models of Problems and Computations" of that course (in which about 15 students participated), we applied RISCAL to model classical "computational" problems but also "dynamic" search and scheduling problems, often disguised in the form of "puzzles" such as, e.g., the well-known "goat, wolf, and cabbage" river crossing puzzle.

In contrast to the computational problems specified by a pair of pre- and postconditions, the dynamic problems were modeled in RISCAL (which at that time did not yet have direct support for such systems) by nondeterministic algorithms of the following structure:

```
proc system(s0:State):
    Tuple[ℕ[N],Array[N,State]]
  requires init(s0);
{
  var s:State = s0;
  var i:ℕ[N] = 0;
  var t:Array[N,State] =
    Array[N,Action](s);
  while ¬goal(s) ∧ i < N do
  {
    choose s1:State with next(s,s1);
    s = s1; i = i+1; t[i] = s;
  }
  return ⟨i,t⟩;
}
```

The predicate $init$ constrains the initial state of the system; the predicate $goal$ describes the desired goal state. Starting with the initial state s_0 as the current state s, the program nondeterministically chooses a successor state s_1 that is related to s by the relation $next$. The computation terminates with the trace t of the states traversed when a desired goal state has been found or a bound for the number i of steps has been reached (to reduce the search space, typically the computation of the successor state is split into the nondeterministic choice of an "action" a and the subsequent deterministic computation of s_1 from s and a).

Students were handed out specification templates with all the necessary declarations; their task was to formalize the $goal$ predicate and the $next$ relation. By running the procedure $system$ in nondeterministic mode, the adequacy of the definitions could be evaluated.

Students apparently liked the "puzzle-like" nature of these problems; also, because of the possibility of self-checking, the submitted solutions were indeed mostly correct. In the proseminar (attended by five students), two students modeled self-selected problems, in one case the card game "Uno", in the other case the problem of the minimization of finite state automata.

All in all, we encountered the use of RISCAL in this novel way a success; it also demonstrated nicely how by the utilization of nondeterministic choices models of "computational systems" can be constructed, for which RISCAL was originally not designed. A crucial point, however, here was the appropriate modeling of the system to manage the exponential explosion of the search space of nondeterministic choices.

c) Logic: The courses presented so far mainly dealt with moderately sized groups of students with some prior technical and formal knowledge. However, since 2013 we are at JKU also (together with three other lecturers) engaged in a 4.5 ETCS course "Logic" for first semester bachelor students of computer science; this course is attended by 250–350 students most of which have just passed their high school exam, not all with a technical focus. The course is internally organized in three modules "SAT" (propositional logic and satisfiability solving), "FO" (first order predicate logic) and "SMT" (satisfiability modulo theories); the FO module takes half of the course.

Over the years we have also integrated more and more the use of logic-based software tools into the course, a SAT solver (Limboole), an interactive proving assistant (RISC ProofNavigator), an automated prover (Theorema), and SMT solvers (Boolector, Z3). However, prior to 2018, we confined the use of these tools to three optional "laboratory" assignments that students could perform on interest and/or as a substitute for three instances of weekly tests. The main reason was that we could not spend adequate time with the explanation and support of this software and thus did not want to make the use of the software mandatory; consequently, however, only a minority of 5–10% of the students used the software, mainly as a substitute for failed tests. To bring the software more into the "main stream" of the course, we introduced in 2018 weekly "bonus" assignments by which students could earn up to 20% of the grade points for the forthcoming tests. These assignments were of comparatively low complexity; they were mainly intended to raise more interest in the software and thus in the practical aspects of the course.

Also starting with the mentioned year, RISCAL replaced the RISC ProofNavigator in the second module FO, specifically in one laboratory assignment and three of the bonus assignments. Questions were handed out in the form of RISCAL skeleton files in which students had to fill in some

missing parts; RISCAL itself was provided in the form of a virtual machine (to be downloaded and executed in the free VirtualBox environment) and on a remote server (to be used via an X2go client). In a “syntax” assignment students had to “parenthesize” formulas to make their structure unique (RISCAL checked their equivalence to the original unparenthesized formulas) and to translate informal statements into formal ones (RISCAL checked the equivalence to another formalization). In a “semantics” assignment, students had to determine satisfying assignments of first-order formulas (RISCAL checked the correctness of the answers) and to transform formulas into logically equivalent forms with certain syntactic constraints (RISCAL checked here the equivalence). In a “pragmatics” assignment, students had to translate given informal problem specifications (pre- and post-conditions) into logic formulas, which partially involved the definition of auxiliary functions and predicates; here RISCAL was used to validate the results by, e.g., checking the input/output behavior of a function implicitly defined by this condition.

At the end of the course, an anonymous evaluation gave the following results: about 40% of the students performed at least one RISCAL exercise, about the same number reported the use of the software as helpful. These numbers clearly trail the SAT solver Limboole used in the SAT module (about 60% used this one and reported it as helpful) but are also much ahead of the other tools in FO and SMT (used by about 25%). Of those who submitted bonus assignments, most indeed earned the full amount of potential grade points. As for the more general questions on why students used the software, twice as many reported as the main reason to earn the bonus points rather than intrinsic interest. Still, most positive impact on interest was reported to the software, while most impact on understanding was attributed to the exercises (three times more than to software). While the overall level of grades did not significantly differ from the previous years, we found a strong correlation between performance on bonus assignments and performance in classroom assignments; indeed, most students that failed the course did not perform the bonus assignments.

Thus, in a nutshell, many students performed the software-based bonus exercises and those who did so achieved also significantly better results in the classroom exercises. However, weak students (subsequently failing the course) mostly did not use the software. The reason to use the software was mainly the “extrinsic” motivation to earn additional grade points and not an “intrinsic” interest. Nevertheless, software was cited as a factor to improve interest in the course, but much less as a factor in improving understanding.

In case of RISCAL, a main deterring factor

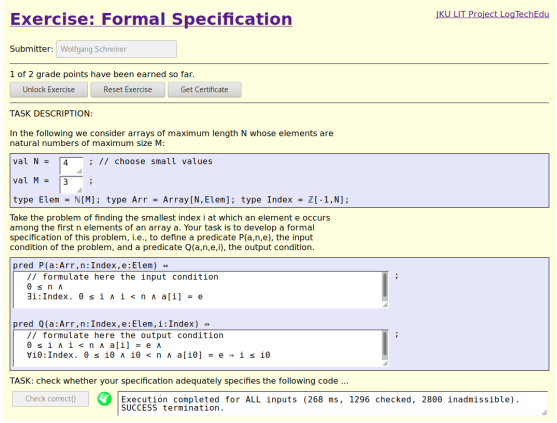


Fig. 4. A RISCAL Web Exercise

to use the software was the need for a local installation (even if by a virtual machine), the need to learn to use the software, and the need to manipulate text files. In the 2019 instance of the course, we therefore introduced a web-based frontend to a server installation of RISCAL that allowed students to perform the exercises within their web browsers (see Figure 4). Indeed, now about 170 students (from about 330 active students) submitted RISCAL-based exercises via the web interface; most with correct results. A preliminary statistical evaluation indicates that those who did so indeed performed in the course tests better than those who did not. Based on these results, we plan in 2020 to proceed with the RISCAL exercises via this frontend.

d) Correct Program and Algorithm Development: There is one kind of courses where the potential of RISCAL has not yet been applied: those on the development of programs (respectively algorithms) where their correctness with respect to given specifications should be checked. Here lecturers might hand out program assignments in the usual way by the desired interface of a procedure and an informal explanation of the inputs it can expect and the results it must deliver. However, additionally, the procedure would be equipped with a formal contract, against which the student could fully automatically check the correctness of her solution and the lecturer could fully automatically check the correctness of a submission. If a solution fails the check, the reported error also demonstrates a concrete input/output pair that demonstrates the failure. In this way, in particular, erroneous boundary cases (that very often give problems) can be quickly detected.

We are not in charge of a corresponding course where RISCAL can be applied in this sense but plan in the near future to approach other lecturers teaching program/algorithm development that may find interest in the use of RISCAL.

5. Conclusions

From the presented experience, we have good evidence that by the use of a "mathematical model checker" such as RISCAL the education in various areas of science and engineering may be substantially improved. This of course mainly applies to closely related "formal" topics, but may be also relevant for topics like programming, where RISCAL can check in small domains the correctness of programs with respect to their specifications.

Our main success lies so far on levels of education where students have already at least some prior (technical and/or formal) background. In courses targeted to absolute beginners, mainly the stronger students profit from the software, while the weaker students (already struggling with the basic material) are potentially overwhelmed by the additional "burden" to use software. Here the use of software requires careful evaluation and fine-tuning.

RISCAL itself has reached a stable state and has been applied in various courses, mostly but not only at JKU. However, while RISCAL is primarily targeted to educational scenarios that typically focus on small models, its semantics-based checking mechanism is efficient enough to also analyze models of non-trivial size. Furthermore, the novel SMT-based decision mechanism presented in this paper allows to check models of sizes that were out of the reach of RISCAL so far.

Further work will focus on the improvement of feedback mechanisms to help students understand the computed results and the integration with proof-based environments to let RISCAL be used as a "pre-checker" of formalizations in small domains before turning to general proofs in domains of arbitrary size. We also plan to extend the recently introduced notion of "concurrent systems" by a model checker for specifications in linear temporal logic (LTL).

References

[1] Barrett, C., Fontaine, P., and Tinelli, C. The Satisfiability Modulo Theories Library (SMT-LIB). http://www.SMT-LIB.org, 2016.

[2] Clarke, E. M., Henzinger, T. A., Veith, H., and Bloem, R., Eds. *Handbook of Model Checking*. Springer, Berlin, Germany, 2018. doi:10.1007/978-3-319-10575-8.

[3] Dutertre, B. Yices 2.2. In *Computer-Aided Verification (CAV'2014)* (July 2014), A. Biere and R. Bloem, Eds., vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 737–744. https://doi.org/10.1007/978-3-319-08867-9_49.

[4] Jackson, D. *Software Abstractions — Logic, Language, and Analysis*, revised ed. MIT Press, Cambridge, MA, USA, 2012. https://mitpress.mit.edu/books/software-abstractions-revised-edition.

[5] Lamport, L. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman, Amsterdam, The Netherlands, 2002. http://research.microsoft.com/users/lamport/tla/book.html.

[6] Leino, K. R. M. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic Programming and Automated Reasoning (LPAR-16), Dakar, Senegal, April 25–May 1, 2010* (2010), E. M. Clarke and A. Voronkov, Eds., vol. 6355 of *Lecture Notes in Computer Science*, Springer, Berlin, Germany, pp. 348–370. doi:10.1007/978-3-642-17511-4_20.

[7] LogTechEdu. JKU LIT Project LOGTECHEDU, July 2019. http://fmv.jku.at/logtechedu.

[8] Merz, S., and Vanzetto, H. Encoding TLA+ into Many-Sorted First-Order Logic. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016* (Cham, 2016), M. J. Butler, K.-D. Schewe, A. Mashkoor, and M. Biró, Eds., vol. 9675 of *Lecture Notes in Computer Science*, Springer International Publishing, pp. 54–69. doi:10.1007/978-3-319-33600-8_3.

[9] Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, Berlin, Germany, October 2017. http://isabelle.in.tum.de/doc/tutorial.pdf.

[10] Reichl, F.-X. The Integration of SMT Solvers into the RISCAL Model Checker. Master's thesis, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria, April 2020. https://www.risc.jku.at/publications/download/risc_6103/Thesis.pdf.

[11] RISCAL. The RISC Algorithm Language (RISCAL), March 2017. https://www.risc.jku.at/research/formal/software/RISCAL.

[12] Schreiner, W. The RISC Algorithm Language (RISCAL) — Tutorial and Reference Manual (Version 1.0). Technical report, RISC, Johannes Kepler University, Linz, Austria, March 2017. Available at [11].

[13] Schreiner, W. Validating Mathematical Theories and Algorithms with RISCAL. In *CICM 2018, 11th Conference on Intelligent Computer Mathematics, Hagenberg, Austria, August 13–17* (2018), F. Rabe, W. Farmer, G. Passmore, and A. Youssef, Eds., vol. 11006 of *Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence*, Springer, Berlin, pp. 248–254. doi:10.1007/978-3-319-96812-4_21.

[14] Schreiner, W. Logic and Semantic Technologies for Computer Science Education. In *Informatics'2019, 2019 IEEE 15th International Scientific Conference on Informatics* (Poprad, Slovakia, November 20–22, 2019), IEEE, pp. 7–12. To appear.

[15] Schreiner, W. Theorem and Algorithm Checking for Courses on Logic and Formal Methods. In *Post-Proceedings ThEdu'18, Theorem proving components for Educational software, Oxford, United Kingdom, July 18, 2018* (2019), P. Quaresma and W. Neuper, Eds., vol. 290 of *EPTCS*, pp. 56–75. doi:10.4204/EPTCS.290.5.

[16] Schreiner, W., Brunhuemer, A., and Fürst, C. Teaching the Formalization of Mathematical Theories and Algorithms via the Automatic Checking of Finite Models. In *Post-Proceedings ThEdu'17, Theorem proving components for Educational software, Gothenburg, Sweden, August 6, 2017* (2018), P. Quaresma and W. Neuper, Eds., vol. 267 of *EPTCS*, pp. 120–139. doi:10.4204/EPTCS.267.8.

[17] SemTech. SemTech — Semantic Technologies for Computer Science Education, January 2018. https://www.risc.jku.at/projects/SemTech.

[18] Steingartner, W., Eldojali, M. A. M., Radakovic, D., and Dostál, J. Software support for course in Semantics of programming languages. In *IEEE 14th International Scientific Conference on Informatics* (Poprad, Slovakia, November 14–16, 2017), pp. 359–364. doi:10.1109/INFORMATICS.2017.8327275.

[19] Steingartner, W., and Novitzká, V. Learning tools in course on semantics of programming languages. In *MMFT 2017 — Mathematical Modelling in Physics and Engineering* (Czestochowa, Poland, September 18–21, 2017), pp. 137–142. http://im.pcz.pl/konferencja/get.php?doc=MMFT2017_streszczenia_wykladow.pdf.

[20] Thiébaut, D. Automatic Evaluation of Computer Programs Using Moodle's Virtual Programming Lab (VPL) Plug-in. *Journal of Computing Sciences in Colleges 30*, 6 (June 2015), 145–151. http://dl.acm.org/citation.cfm?id=2753024.2753053.

Software Support for Visualizing of the Graph Algorithms in a Novel Approach in Educating of Young IT Experts

Mocinecová, Karina; Steingartner, William

Abstract: ***We present in this paper a software module which provides simulation and dynamic visualization (algorithm animation) of selected graph algorithms. The graphs as data structures and graph algorithms are a very important part of Data Structures and Algorithms undergraduate course in a computer science curriculum. The designed and prepared software module provides advanced functionality on graphs and algorithms, step-wise and overall mode of animation in both directions – forward and step-back visualization. Our software module can serve as a very useful teaching tool for students and programmers who need to simulate some algorithms on graphs as data structures.***

Index Terms: ***graph, graph algorithm, JavaFX, teaching software, university didactic, visualization***

1. Introduction

ALGORITHMS play a crucial *rôle* in the process of design programs. Furthermore, algorithms and data structures are essential topics in the undergraduate computer science curriculum. A study of data structures and algorithms can give a rise to writing more efficient programs [21]. Algorithms can be studied based on the mathematical and empirical analyses. Today, the visualization of algorithms is another and a very popular way to study algorithms and it is currently being investigated. An algorithm visualization can be defined as the use of images to convey some useful information about algorithms. That information can be a visual illustration of an algorithm's operation, of its performance on different kinds of inputs, or of its execution speed versus that of other algorithms for the same problem.

There are two main variations of algorithm visualization: static algorithm visualization and dynamic algorithm visualization (also known as algorithm animation).

An application of algorithm visualization can be found in research and education. Benefits for research are mostly that the visualization of an algorithm can uncover some unknown features of algorithms or can help improve the algorithm [28].

The application of algorithm visualization to education helps students in learning algorithms. Visualization and simulation can show basic principles of an algorithm, can help illustrate how a particular step of algorithm works. When a simulation allows interactively to change input data, then we can compare how an algorithm behaves on different input data.

Of course, it is true that students can learn and understand algorithms, also, without using a visualization and animation. But the reality is that many researchers and educators assume that students would learn an algorithm faster and more thoroughly using algorithm visualization [6], [7], [16], [18].

In this paper, we focus on the teaching tool for visualization and animation of graphs and selected graph algorithm. In section 2, we compare some well-known tools with similar functionalities that our software tool has and we explain the need for development of new tools. In section 3, we present our motivation for modernization of the teaching process and how we expect an increase of the attractiveness in the discussed topic for students. The section 4 sketches particular parts of design, deployment and application of our software tool. We conclude our paper with section 5 where we summarize our ideas and discuss future steps.

2. Related Work

A relatively wide range of free and easily accessible tools for visualizing graph algorithms are available. Before developing our application, we examined several similar solutions, of which we focused primarily on Graph Online [2], Data Structure Visualizations [5], and Algorithm Visualizer [1]. We tried to combine the positives of these applications, avoid negatives, and be inspired by the best functionalities they offer.

Manuscript received Apr 2020. This work was supported by the project KEGA 011TUKE-4/2020: "A development of the new semantic technologies in educating of young IT experts" and in the frame of the initiative project "Semantic Modeling of Component-Based Program Systems" under the bilateral program "Aktion Österreich – Slowakei, Wissenschafts- und Erziehungskooperation"

K. Mocinecová was a student of computer science at the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia (e-mail: karina.mocinecova@student.tuke.sk).

W. Steingartner (contact person) is an assistant professor at the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia (e-mail: william.steingartner@tuke.sk).

None of the related applications mentioned above allow visualization of all graph algorithms that are presented in the course for which our software was developed as a didactic tool. They also do not contain a detailed description of the algorithm steps performed in the way we have suggested. This was the main reason why we decided to develop an application that will be clear and easy to use even for beginners in the field of graphs and algorithms and will meet the requirements for teaching the selected topic in the course on Data Structures and Algorithms.

The Graph Online web application offers several options for creating a graph. The primary way is to draw vertices and edges interactively, which is what we focused on because it appealed to us the most. Drawing vertices, edges, and manipulating the graph are performed in different modes, between which it is necessary to switch. This style of drawing was an inspiration for us because it allows a user to create a graph according to his needs, but constantly switching e.g. between the mode for adding vertices and the mode for joining vertices was inconvenient for us.

Data Structure Visualizations and Algorithm Visualizer are tools that visualize algorithms on prepared graphs. They do not allow user to create a graph or modify the one that is displayed by default. The user is therefore limited to work with the graph that is offered. Data Structure Visualizations give users a choice of at least two graphs on which the algorithm is to be visualized, but it may be the case that none of the recommended graphs will meet the user's requirements.

The way the algorithms are visualized is very similar in all selected applications. It is mostly a color highlighting of graph objects that can be played automatically or manually stepped by the user. Furthermore, the Data Structure Visualizations application explains the steps performed more markedly, e.g. using a queue. The other two applications perform visualization only on the graph without any further explanation. Although the Algorithm Visualizer offers the code of a graph algorithm, during the visualization it does not highlight its parts that are currently being executed, and besides, shown code is written in a specific programming language like Java and not in pseudo-code, which can be considered quite limiting.

In our application, all the valuable functionalities offered by the mentioned tools were used. We also supplemented the application with our own ideas and suggestions to be in compliance with the requirements of the course for which it was created.

3. Motivation for Using Software Tools in the Teaching Process

In this section, we briefly present the motivation for integrating new technologies into the teaching process and how we apply them. We focus on extending a standard method of teaching with the innovated elements using the visualizing software – we present a contribution to teaching modernization which aim is innovation and increasing the attractiveness of the existing course on Data Structures and Algorithms located at the website https://kurzy.kpi.fei.tuke.sk/usaa/student/ along with the course content placed into the Moodle LMS.

3.1. The Course on Data Structures and Algorithms

In computer science education, many new practical skills must be taught. It is of prime importance to give students who learn system-software concepts a solid base of knowledge without any unnecessary details [9], [14]. In fact, software engineering does not mean only to write better programs [12], [25]. The course on Data Structures and Algorithms which is provided by the Faculty of Electrical Engineering and Informatics, Technical University of Košice, for the undergraduate degree in the study program Computer Science has been taught partly by conventional teaching methods which include lecturing and face-to-face interaction in a classroom and partly by using teaching software. The use of software is significant and plays a growing *rôle* in education, in particular in academic courses in computer science and mathematics [19]. Lecturing is provided by a teacher. The teacher delivers a content to students and learning level can be measured with the help of a written examination. Our main idea was to contribute to innovations based on presenting and visualizing behavior of selected graph algorithms more understandably and attractively. Research, experimental and development work were realized under the educational project "A development of the new semantic technologies in educating of young IT experts".

3.2. A Visualizing Tool in Teaching Process

The software is focused on improving and making teaching methods applied to the lecture and laboratory part of the course more attractive. Our goal was to apply modern software solutions in this course as a significant help in teaching lectures and laboratory exercises, to support the self-study and to motivate students to realize experiments with graphs. Presented software module will allow an illustrative and understandable use of selected graph algorithms with the main focus on dynamic visualization and showing particular steps of applying the algorithm shown in a pseudo-code. This process is modeled in one-to-one correspondence: each visual change of a

graph is expressed by a particular step of the algorithm in the pseudo-code. In this way, we achieve greater clarity in the applied procedures and principles in the course teaching.

Based on experience with the course on Data Structures and Algorithms, we identified points in which we can bring innovations and make this course more attractive for students and young IT experts. We revised the content of the course to reflect the current state of the art in the world (mostly oriented towards modern technologies). Some modules for visualizing selected algorithms have been developed and they are successfully used in teaching the course [23], [24]. We think that modern visualizing methods can significantly help and we want to apply them, but we do not want to avoid using standard methods [22]. Of course, we see those standard teaching methods – an explanation, algorithm demonstration on an example by a teacher and its practical implementation on laboratory exercise by students with the help of teacher where the *rôle* of the educator/teacher is indispensable and cannot be so simply omitted.

3.3. Results and Outcomes

We expect the practical outcomes of using the software tool in teaching computer science courses oriented to formal principles and software engineering with focusing on the proper implementation of graph algorithms in selected practical problems. One of the advantages can also be putting this software into practice for distance learning. Our software tool can serve as a modern interactive learning tool, as a support for new teachers in the course, and as a tool for IT experts using applications of graphs.

Proposed teaching software can help teachers (and educators) in better and illustrative form for students:

- during the lectures, the teacher can present examples directly and interactively; or at least use prepared examples depicted on screenshots;
- at laboratory exercises and seminars, examples can be explained step-wise with a possible change of input parameters and graph parameters to show the possible differences in algorithms' simulations.

Teaching software can be very useful for students especially in the following cases:

- during the laboratory work for interactive simulation of particular graph algorithm when changing the input parameters and graph configurations;
- in the phase of self-studying, applying the study-by-experiments method;
- in case of doing research or simulations when visual output is needed, especially when preparing own software solution for some graph problem.

The theoretical outcomes of using the software package can find their application in the field of further research on the issue of interactive and experiential teaching of theoretical principles of graph algorithms in computer science. We expect a raising of interest in simulations of algorithms and combining the theoretical principles with practical skills thank to simulations and visual representations, as well.

4. Software Tool

This paper describes a tool enabling visualization of graph algorithms such as depth-first search [26], breadth-first search [27], test for bipartite [20], finding graph components [17], Dijkstra's algorithm, Bellman-Ford algorithm and Floyd-Warshall algorithm to find the shortest path in a graph [4], isomorphism test [29], check of Eulerian and Hamiltonian graphs [8], [13], identification of minimum spanning tree using the algorithm of Kruskal and Jarník [3], and detection of a critical path [11]. An application was created as a teaching tool for Data Structures and Algorithms course and it was developed within a practical part of the master's thesis [15]. The primary purpose of the application was to provide the users with a comprehensible explanation of chosen graph algorithms, mainly by visualizing them. The key part is to visualize each step of the algorithm in a way that is clear and easily understandable. In the next sections, we describe the tool in more depth, namely the user interface and all the functionalities that the application offers. Next, we also specify its technical specification and using a UML diagram, we describe how the application was designed and developed.

4.1. User Interface and Functionalities

The created teaching tool is largely a single-frame application. In the process of designing the user interface, we focused on the availability of all crucial objects and functionalities that are necessary for the manipulation with the graphs and for the correct visualization of them. The application consists of the following basic parts:

- canvas, primarily dedicated to the manipulation of the graphs and controlling the visualization (Figure 1 – case *a*);
- list of implemented graph algorithms together with brief descriptions of their behavior (Figure 1 – case *b*);
- a side panel that displays the pseudo-code of the chosen graph algorithm together with listings of various intermediate data, which we get during traversing the graph (Figure 1 – case *c*).

(Figure 1 – see appendix)

After familiarization with the tool's user interface, we can proceed to the description of its particular functionalities. The creation of a graph and selecting an algorithm are necessary for running a visualization, which is the primary focus of this tool.

The first functionality is the creation of a graph. There are three implemented methods of how a graph can be created. We aimed to enable a user to create any graph, which can be done easily by drawing – that is the primary way of creating an arbitrary graph. By clicking the left button on the mouse anywhere on the canvas, the vertices are created with numbered labels. Already existing vertices can be connected by edges, which are created by subsequently double-clicking both the vertices the user wants to connect. In the graph theory, several types of edges are known, such as directed, undirected, weighted, unweighted, multiple, graph loops, etc. For the purpose of this application, we limited the types to create only directed and undirected edges, and also to weighted and unweighted edges. Therefore, the application does not support the creation of multiple edges and graph loops.

For the user who does not need to create a custom graph or only needs to demonstrate the visualization of the algorithms in a faster way, there is an option to choose a graph from examples, which is the second way of graph creation. The complete list of given sample graphs can be found in *Graph examples*. Any graph from the examples can be added to the canvas by a simple click on a button. Such a graph can be edited in the same way as a graph that was drawn. The user still can add and remove any vertices or edges and edit them.

The third option for creating a graph is to import it from an XML file. A specific structure of the XML format was defined for this application, which is shown in Listing 1.

Listing 1: XML file with a specific structure

```
<?xml version="1.0" encoding="UTF-8"?>
<GRAPH>
  <NODE>
    <ID>1</ID>
    <LABEL>4</LABEL>
  </NODE>
  <NODE>
    <ID>2</ID>
    <LABEL>2</LABEL>
  </NODE>
  <EDGE>
    <ID>1-2</ID>
    <DIRECTION>yes</DIRECTION>
    <FROM_ID>1</FROM_ID>
    <TO_ID>2</TO_ID>
  </EDGE>
  <WEIGHT>
    <ID>SpriteEdge1-2</ID>
    <EDGE_ID>1-2</EDGE_ID>
    <LABEL>17</LABEL>
  </WEIGHT>
  <NODE_NOTE>
    <ID>SpriteNode2</ID>
    <NODE_ID>2</NODE_ID>
    <LABEL>[17 ; 17]</LABEL>
  </NODE_NOTE>
  <NODE_NOTE>
    <ID>SpriteNode1</ID>
    <NODE_ID>1</NODE_ID>
    <LABEL>[0 ; 0]</LABEL>
  </NODE_NOTE>
</GRAPH>
```

The *GRAPH* element can consist of four elements: *NODE*, *EDGE*, *WEIGHT*, and *NODE_NOTE*. Each *NODE* must have a unique ID, and if we want the vertex label to be different from its *ID*, then a different *LABEL* has to be defined. For all edges, it is mandatory to define a unique *ID*, *DIRECTION* (yes/no), and ID of the two vertices that the edge connects (*FROM_ID* and *TO_ID*). It is also mandatory to specify a *WEIGHT* element for each edge. It must have a unique *ID*, information about the edge to which it is assigned (*EDGE_ID*), and if the edge is to be weighted, then the *LABEL* element is also required. The last element is *NODE_NOTE*. It represents a text that is displayed at the vertex in a graph (for example, the degree of a vertex). If we want to have this element defined in the XML file, then it must have a unique *ID*, information about a vertex to which it belongs (*NODE_ID*) and the text itself included in a *LABEL* element.

As we have already mentioned, the defined structure of the XML consists of mandatory (all elements highlighted in red) and optional elements. If any of the mandatory elements are missing or there is an error in the syntax of any element, be it optional or mandatory, no graph will be imported. The user is notified about such an error. After importing a correct XML file, a graph is generated and can be modified by the user in any way. Export of a graph to an XML file with a defined structure is possible for a graph created in any of the mentioned ways.

After a graph is created, there is a possibility to change some of its attributes additionally. A graph can be modified easily anytime, by changing vertices properties, edge properties, or the whole graph.

Modifying a vertex means to change the number of the label, mark a vertex as a starting vertex or ending vertex for an algorithm or delete the vertex entirely. These change options are available after a right-click on a vertex.

Any edge can be adjusted by changing its attributes, which depends on the type of the edge

– if it is directed or not and also if it is weighted or not. In a case that an edge is undirected or unweighted, the direction or weight of the edge can be set, and by that, it becomes a directed or weighted edge for which there is an option to change or remove its weight or direction anytime. In all cases, the edge can be also deleted. All of the mentioned options are displayed after a right-click in the middle of the edge.

There is also a possibility to change the whole graph globally by a right-click on canvas anywhere except on any of the objects of a graph. The change options include deleting all the weights of the edges, deleting all the directions of the edges, deleting all the node notes, deleting all the edges completely, and deleting all the vertices from the canvas. It is also possible to reset the graph to its original state, so any properties acquired during a visualization will be removed.

Another functionality of the learning aid is a selection of a graph algorithm to be visualized. The user can choose an algorithm from the list of implemented algorithms in the tool. Each provided algorithm is accompanied by a brief description of its behavior. A selection of an algorithm is followed by displaying the pseudo-code for the particular algorithm in the side panel. Also, useful intermediate data are provided there during traversing the graph. Therefore, each step of the algorithm is explained in detail, while the visualization is in progress. Providing an intermediated data precisely together with the pseudo-code in real-time was our primary focus as it makes this application a good tool to use in the process of learning and understanding the workings of various graph algorithms.

In a situation, when a graph is created and a graph algorithm is selected, the user can proceed to run a visualization. There are two ways of how visualization can be triggered. One is an automatic run of the visualization, which means each step of the visualization is played subsequently without any user input in a chosen time interval for every step. The interval can be set by the user before or during the run. The automatic run can be paused and resumed anytime during the visualization process. The second method of running a visualization is manual. The user can control the course of the algorithm visualization by clicking a button and therefore manually triggering the next or previous step in the process. All the buttons that are used to control the visualization are located above the canvas.

The use of either method to start a visualization results with the same functioning. An essential part of the visualization process (mainly for the learning purposes) is the displaying of each executed step by color-coding both the specific parts of the pseudo-code and the corresponding objects in the graph. For example, if the algorithm needs to find all neighbors of a current vertex, a pseudo-code line describing this is highlighted in one color, such as yellow, and all found neighbor vertices are also colored in yellow (Figure 2). On the top of the highlighted pseudo-code lines, a user is provided with an already mentioned explanation of the current step and also with intermediate data. The most emphasis was put on this feature as it is aimed to help students better understand and see how the graph algorithms work. We suppose this application would be helpful and easy to use, making a learning process of graph algorithms easier and faster.

(Figure 2 – see appendix)

4.2. Technical Specification

The software we created is a JavaFX application. JavaFX [10] is a platform designed, among other things, to create desktop Java applications that are supported by many operating systems, the most common of which are Microsoft Windows, macOS, or Linux. The primary reason why we chose the JavaFX platform was that we decided to implement the GraphStream library, mainly because it provides many interesting functionalities. It supports the creation of various types of graphs, editing them additionall and offering several options for styling each object in the graph, and much more. This library met most of our requirements, which were set before the development of the application when we already had an idea of what the visualization of the algorithms should look like. JavaFX and Java version 9.0.4 were used when creating the application.

The application is distributed as a self-contained executable JAR file that includes configuration and all necessary dependencies, which allows direct usage and does not require to have the proper Java version installed in advance since everything needed has been packaged into one file.

4.3. Implementation of a Tool

The simplified UML diagram (Figure 3) shows how the application was designed and developed. The figure illustrates only selected parts of the tool that are responsible for manipulating the graph and controlling the course of the algorithm visualization (the provided UML diagram presents a depth-first search algorithm; the way of implementation of the other graph algorithms is identical).

(Figure 3 – see appendix)

When the application is started, the *start()* method is called in the *App* class, which loads the *app.fxml* file and creates the *AppController* class. The methods in the *AppController* class are used to handle elements in a given *fxml* file. Moreover, the *DrawPanelFX* class is created at startup.

The *DrawPanelFX* class is important because it creates an interactive canvas to display and work with graphs. Especially, the *drawingGraph()* method is worth mentioning, due to its responsibility for creating vertices, edges, editing them additionally, and everything else that is allowed to do by clicking on the canvas.

We mentioned that the application enables a user to select a graph from a set of sample graphs, the preview of which is provided by the *AppController* class. Without calling the *drawingGraph()* method, only the *DrawPanelFX* class is used to display the graph, and therefore this class is also created in the *AppController*.

The presented UML diagram does not include methods that, after selecting an algorithm, display appropriate pseudocode and fields to list the performed steps of the chosen algorithm. Nevertheless, these methods are part of the class *AppController*.

Further, we move on to the description of the way the algorithms were implemented. Three classes were created for each graph algorithm – one main class which extends the *Algorithm* class and two more for an automatic run of the visualization and manual stepping. In the UML diagram, describing the implementation of the depth-first search algorithm, the three classes mentioned are *DFS*, *DFSStep*, and *DFSPlay*.

Pressing the *play* button above the canvas and calling the *play()* method creates an instance of the main class of the algorithm – *DFS*. Based on the selected type of visualization, in this case it is a method of an automatic run of the visualization, an appropriate class is created – *DFSPlay*. The *playDFS()* method ensures that the algorithm steps are played, for which a user sets a time interval. Because the *Algorithm* class extends the *Thread* class, it is possible to call the *sleep()* method in the *playDFS()* method to provide the desired pause between the steps. The *pause()* and *stop()* methods are used to interrupt or completely stop the visualization.

Another implemented visualization method is to manually step an algorithm by pressing the forward and backward buttons. The first time the forward button is pressed, the main class of the algorithm – *DFS* – is created, followed by the class *DFSStep*. The program continues by calling the *playBeforeStepping()* method, where the data structure *ArrayList* $<$*StepTemp*$>$ is filled with algorithm steps as *StepTemp* objects. Subsequently, by pressing the forward and backward buttons, a user goes through the pre-stored steps of the algorithm, thus the both methods *stepForward()* (*stepDFSForward()*) and *stepBackward()* (*stepDFSBackwards()*) are called in the background.

This is a brief overview of the classes and methods that are the most important and used in process of visualization. The application has many other classes, which, for example, display the confirmation and error alerts, provide export and import of the graphs as XML files, etc. As these are not such important functionalities, and even without them the visualization itself would eventually be possible, we decided not to add them to the simplified UML diagram of the application.

4.4. User Domain of a Teaching Tool

The main goal of the application is to provide users with a demonstrative and clear visualization of selected graph algorithms. It is not intended only for students of the course Data Structures and Algorithms, but for all those who are interested in this field. Our tool allows users to create any graphs in several ways, of which the process of drawing a graph interactively can certainly be considered mostly beneficial. Thus, the application gives users the space to create graphs according to their needs, on which it is possible to visualize any of the implemented graph algorithms. The created software can be used as a didactic aid in the teaching process, which clarifies and describes in detail the principles of presented graph algorithms, as well as a tool for experimenting with graphs and algorithms, which we hope is attractive for young IT experts and will be fully utilized by them.

5. CONCLUSION

Based on our research on simulation-based tools for the education of computer science students in the formal modeling, analysis, and reasoning of graphs and graph algorithms, we presented a software module that is ready to be used in the teaching process for the course on Data Structures and Algorithms. Our software module provides visual simulation and explanation of selected graph algorithms. We are convinced that increasing of programming skills in concrete topic can be supported also by visualizing and simulating methods. Thus, our software module can serve as a very suitable tool for studying the graphs and graph algorithms and their properties. We plan in our future work to design and prepare more visualizing tools that will be suitable for other topics in the field of Data Structures and Algorithms and to integrate them into one software package. We expect that this research significantly contributes to improving and sharing our theoretical and practical experience and knowledge in the domain of the solved problems.

ACKNOWLEDGMENTS

Authors would like to thank Ing. Rastislav Pištej for his valuable comments, observations and suggestions that contributed to the improvement of the final form of the developed application.

References

[1] *Algorithm Visualizer*. https://algorithm-visualizer.org/.

[2] *Graph Online*. https://graphonline.ru/en/.

[3] Karel Antoš. Minimum spanning tree problem. *14th Conference on Applied Mathematics, APLIMAT 2015*, pages 10–19, 2015.

[4] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. The MIT Press, 3 edition, 2009.

[5] Galles Galles. *Data Structure Visualizations*. University of San Francisco. https://www.cs.usfca.edu/~galles/visualization/Algorithms.html.

[6] Steven R. Hansen, Hari N. Narayanan, and Dan Schrimpsher. Helping learners visualize and comprehend algorithms. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*, 2, 01 2000.

[7] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13:259–290, 06 2002.

[8] Marián Klešč and Ján Plavka. *Mathematics 2 - Discrete mathematics and formal logic*. Technical University of Košice, 1st edition, 2015.

[9] Klemen Kloboves, Jurij Mihelič, Patricio Bulić, and Tomaž Dobravec. Fpga-based sic/xe processor and supporting toolchain. *International journal of engineering education*, 33(6(A)):1927–1939, 2017.

[10] Georgios Kouimtzis and Konstantinos Chertouras. Get rid of the numbers: Using javafx as an algorithm instruction tool. pages 340–341, 2009.

[11] TianZhi Li. A novel algorithm for critical paths. *2009 WRI World Congress on Computer Science and Information Engineering*, 1:226–229, 2009.

[12] Shaoying Liu, Kazuhiro Takahashi, Toshinori Hayashi, and Toshihiro Nakayama. Teaching formal methods in the context of software engineering. *SIGCSE Bulletin*, 41:17–23, 06 2009.

[13] Martin Mareš and Tomáš Valla. *Průvodce labyrintem algoritmů*. CZ.NIC, Praha, 1st edition, 2017.

[14] Jurij Mihelič and Tomaž Dobravec. SicSim: A simulator of the educational SIC/XE computer for a system-software course. *Computer Applications in Engineering Education*, 23(1):137–146, 2015.

[15] Karina Mocinecová. Extending of an existing software platform for support for teaching the course data structures and algorithms. Master's thesis, Technical University of Košice, Košice, 5 2020. (in Slovak).

[16] Davorka Radaković and Djordje Herceg. Towards a completely extensible dynamic geometry software with metadata. *Computer Languages, Systems & Structures*, 52:1 – 20, 2018.

[17] Md. Saidur Rahman. *Basic Graph Theory*. Springer Publishing Company, Incorporated, 1st edition, 2017.

[18] Purvi Saraiya, Clifford Shaffer, D. Mccrickard, and Chris North. Effective features of algorithm visualizations. *Proceedings of the SIGCSE Technical Symposium on Computer Science Education*, 36, 12 2003.

[19] Wolfgang Schreiner. Logic and semantic technologies for computer science education. In Anikó Szakál William Steingartner, Štefan Korečko, editor, *Informatics'2019, 2019 IEEE 15th International Scientific Conference on Informatics*, pages 7–12. IEEE, 2019.

[20] Jiří Sedláček. *Úvod do teorie grafů*. Academia, Praha, 1981.

[21] Clifford A. Shaffer. *Data Structures and Algorithm Analysis Edition 3.2 (Java Version)*. Dover Publications, 3 edition, 2013.

[22] Katarína Teplická, Miloš Matvija, and William Steingartner. *Innovative didactic methods in the teaching process at universities*. Technical University of Košice, 2020. (in Slovak).

[23] Slavomír Šimoňák. Using algorithm visualizations in computer science education. *Central European Journal of Computer Science*, 4(3):183–190, 2014.

[24] Slavomír Šimoňák and Martin Benej. Algomaster platform extension for improved usability. *Journal of Electrical and Electronics Engineering*, 10(1):27–30, 2017.

[25] John Wordsworth. Education in formal methods for software engineering. *Information and Software Technology*, 29(1):27–32, 1987.

[26] Jianbin Xiong, Zhenkun Li, Luqiao Fan, and Dongping Liu. Improved depth first algorithm and its application in information retrieval. *2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA)*, pages 69–73, 2010.

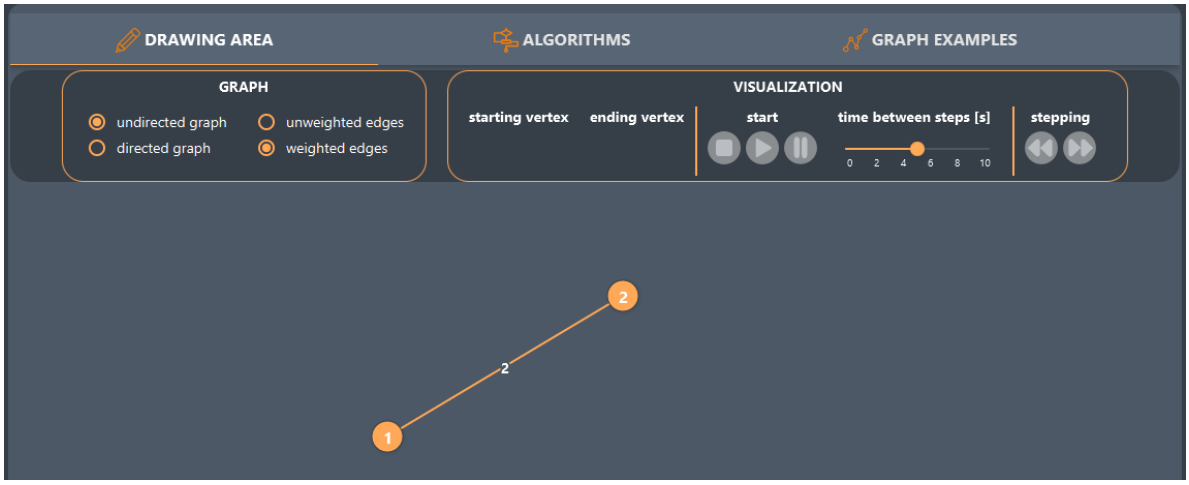
[27] Chunde Yang, Guohui Wei, Jun Tan, and Jingbo Xie. The breadth first search traversing algorithm of the graphs in dna computer. *2009 3rd International Conference on Bioinformatics and Biomedical Engineering*, pages 1–4, 2009.

[28] Wei Yu and Feng Su. On application of algorithm visualization in teaching software of algorithm. *2010 International Conference on Electrical and Control Engineering*, pages 2876–2879, 2010.

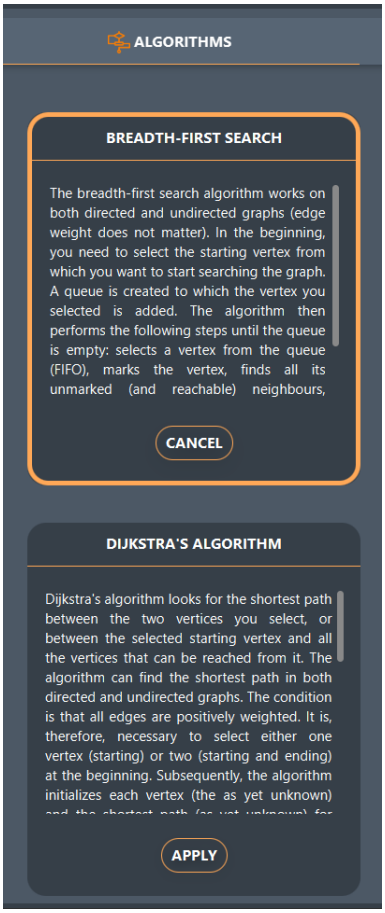
[29] Baida Zhang, Yuhua Tang, Junjie Wu, and Linqi Huang. A unique vertex deleting algorithm for graph isomorphism. *2011 International Symposium on Image and Data Fusion*, pages 1–4, 2011.

Karina Mocinecová obtained her bachelor's degree in 2018 at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics at Technical University in Košice, Slovakia. She graduated (Ing.) from the same department of the Technical University in Košice with a major in informatics in 2020.

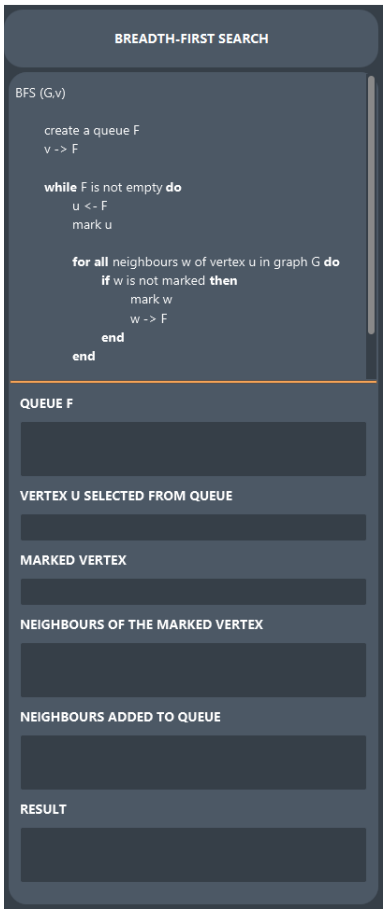
William Steingartner works as Assistant Professor of Computer Science at the Department of Computers and Informatics of the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. He defended his Ph.D. thesis "The *Rôle* of Toposes in Informatics" in 2008. His main fields of research are semantics of programming languages, category theory, compilers, data structures and recursion theory. He also works with software engineering.



(a) canvas and buttons for controlling the visualization



(b) part of the list of implemented algorithms



(c) a side panel

Fig. 1: Basic parts of the graphical interface of the application

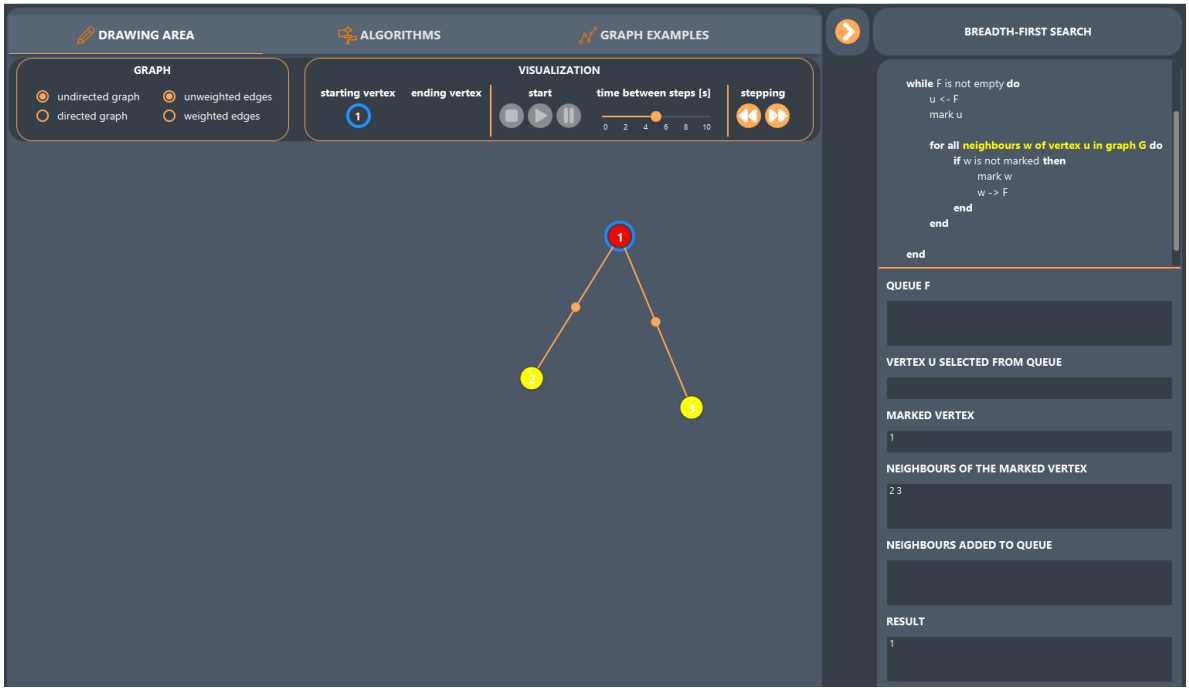


Fig. 2: Visualizing a breadth-first search algorithm

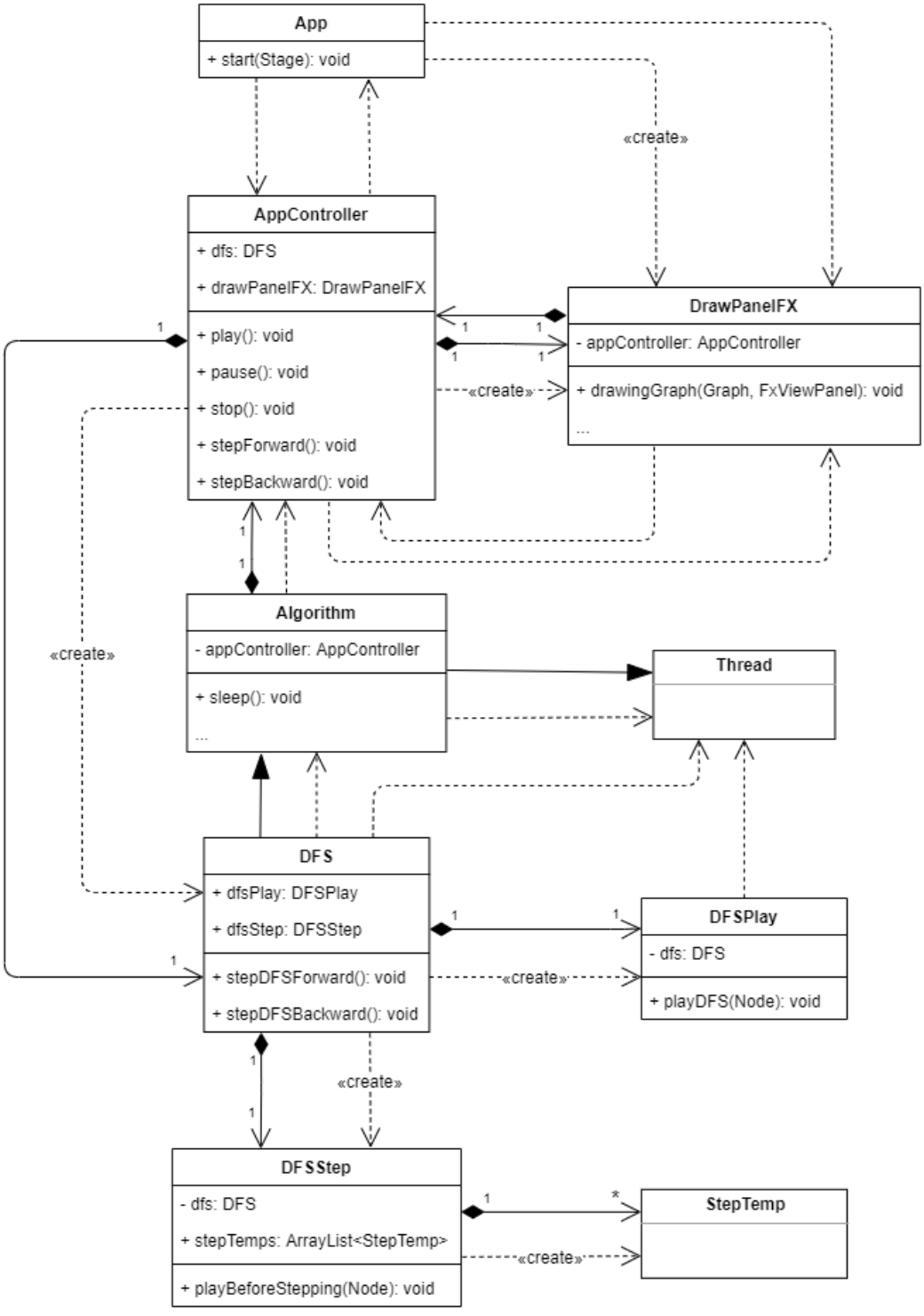


Fig. 3: Simplified UML diagram of the tool

Totally Bounded Metric Spaces, Their Model Theoretic Stability and Similarity Detecting Algorithms

Sági, Gábor and Al-Sabti, Karrar

Abstract: ***Many questions of theoretical computer science can be reduced to the following problem: let $\mathcal{X} = \langle X, \varrho \rangle$ be a metric space, let $A \subseteq X$ and let ε be a positive real number; for a given input $x \in X$ find $a \in A$ (if any) for which $\varrho(a, x) \leq \varepsilon$. This problem is called the similarity detecting problem of $(\mathcal{X}, A, \varepsilon)$. Usually, A (or sometimes X) is finite but huge, and the challenge is to represent the metric space in such a way computer algorithms may handle it efficiently.***

This paper consists of two main parts. The first part has a theoretical character: we investigate model theoretic properties of certain metric spaces. In Theorem 2.4 we show, that relational structures associated to totally bounded metric spaces have some stability properties in the model theoretic sense (all relevant definitions will be recalled in the paper). This result allows us to build some metric spaces from their finite subspaces.

The second part of the paper is application oriented. Based on the first part and on some results of [9] in the second part we propose a similarity detecting algorithm. We associate a finite dimensional Euclidean space $\mathcal{Y}$ to a totally bounded metric space $\mathcal{X}$ and an "almost isometry" $f : X \to Y$ which preserve distances modulo a controlled amount of inaccuracy. After that, instead of working with $\mathcal{X}$, we can work with $\mathcal{Y}$. The main result of this part is the description of the above method.

In the special case, when $\mathcal{X}$ itself is a large dimensional Euclidean space (with its usual Euclidean metric), our method can be considered as a kind of dimension reduction. In this special case we are analyzing the time complexity of our proposed algorithm, as well.

Index Terms: ***Totally bounded metric spaces, similarity detecting algorithms, dimension reduction.***

1. Introduction

The present work has a practical and a theoretical motivation. We start by the practical motivation.

Many questions of theoretical computer science can be reduced to questions about certain metric spaces, for further details we refer to [2], [6] and the references therein. Usually, these spaces are finite, but huge and the problem is how to handle these spaces by computer algorithms in a tractable way. This is the case, for example, if the distance function of the metric space measures "similarity" of two objects and the problem is to find the elements of a database which are similar enough to a given input. Related problems can be effectively solved if one is able to represent the metric space in a suitable way, for example, if one is able to embed the metric space into a finite dimensional Euclidean space (endowed with its usual metric) with a function which is an "almost isometry", or if one is able to embed a compact subset of a (finite dimensional) Euclidean space having large dimension into a considerably smaller dimensional Euclidean space by an "almost isometry".

For the theoretical background we briefly recall investigations initiated in [10] and continued in [9]. For metric spaces $\mathcal{X} = \langle X, \varrho \rangle$, $\mathcal{Y} = \langle Y, \sigma \rangle$ and a positive real number ε, a function $f : X \to Y$ is defined to be an ε-map iff for all $y \in Y$, the diameter of $f^{-1}(y)$ is at most ε. Thus, if ε is small, then f is almost injective. In Theorem 10 of [9] the first author gave a new proof for the following well-known fact: if $\mathcal{X}$ is totally bounded (for further explanation see Definition 2.1 and the sentence immediately after it), then for all ε there exists a finite number n and a continuous ε-map $f_\varepsilon : X \to \mathbb{R}^n$, where $\mathbb{R}^n$ is the usual n-dimensional Euclidean space endowed with the Euclidean metric. Such ε-maps still exist even if $\mathcal{X}$ has infinite covering dimension (in this case, n depends on ε, of course). Contrary to the previously known proofs (see e.g. Chapter 8 of [8]), the proof technique in [9] is effective in the sense, that it allows to establish estimations for n in terms of ε and structural properties of $\mathcal{X}$.

Now we turn to the theoretical part of this paper. There is a well known method (which will be recalled in Section 2 below) that associates a first order relational structure to a metric space. This

Manuscript received April 29, 2020. This work was supported by the Hungarian National Foundation for Scientific Research grant K129211.

G. Sági (contact person) Alfréd Rényi Institute of Mathematics, Reáltanoda u. 13-15, H-1053 Budapest, Hungary and Budapest University of Technology and Economics, Department of Algebra, Egry J. u. 1, H-1111 Budapest, Hungary (e-mail: sagi@renyi.hu).

K.Al-Sabti Budapest University of Technology and Economics, Department of Algebra, Egry J. u. 1, H-1111 Budapest, Hungary and University of Kufa, Faculty of Computer Science and Mathematics (e-mail: karrard.alsabti@uokufa.edu.iq).

method allows us to translate questions about metric spaces into questions about relational structures and these translated questions may be investigated by techniques of first order logic and model theory. From the results of [9] one can easily obtain that if $\mathcal{X}$ is a totally bounded metric space, then it's associated relational structure $\mathcal{A}(\mathcal{X})$ is Δ-stable in the model theoretic sense. We will reconstruct the proof of this fact below in Theorem 2.3. The main goal of section 2 is to investigate the converse, that is, in Theorem 2.4 we will show, that if $\mathcal{X}$ is dense in itself, ϱ is bounded, and $\mathcal{A}(\mathcal{X})$ is Δ-stable in a rather strong sense, then $\mathcal{X}$ is totally bounded.

On the basis of this background, in Section 3 we are proposing a method which can be used for similarity detecting, clustering and related problems. The structure of this paper is as follows. At the end of this section we are summing up our system of notation. Section 2 is devoted to theoretical investigations. Here the main result is Theorem 2.4 which provides a characterization for totally bounded metric spaces in terms of stability properties of their associated relational structures. In Section 3 we describe and analyze our similarity detecting algorithm with special emphasis for the case, when $\mathcal{X}$ itself is a large dimensional Euclidean space. Finally in Section 4 we provide some conclusion.

Notation

Our notation is mostly standard, but the following explanation may help.

Throughout $\mathbb{N}$ denotes the set of natural numbers. In addition, $\mathbb{R}$ and $\mathbb{R}^+$ denotes the set of real numbers, and the set of positive real numbers, respectively.

Let $\mathcal{X} = \langle X, \varrho \rangle$ be a metric space, $a \in X$ and let γ be a non-negative real number. As usual, the open γ-ball $B(\gamma, a)$ at a is the set

$$B(\gamma, a) = \{x \in X : \varrho(a, x) < \gamma\}.$$

If $\mathcal{E}$ is a Euclidean space, then the norm of any element x of $\mathcal{E}$ will be denoted by $||x||$ (the norm function of $\mathcal{E}$ and its usual metric are mutually definable from each other in the usual way).

2. Totally Bounded Metric Spaces and Model Tiheoretic Stability

Let $\mathcal{X} = \langle X, \varrho \rangle$ be a metric space. As it is well known, one can associate a relational structure $\mathcal{A}(\mathcal{X})$ to $\mathcal{X}$ in the following way. If d is a distance of $\mathcal{X}$, that is, $d \in ran(\varrho)$ then the binary relation R_d is defined to be

$$R_d = \{\langle a, b \rangle \in X^2 : \varrho(a, b) \leq d\}.$$

Thus, the relational structure

$$\mathcal{A}(\mathcal{X}) := \langle X, R_d \rangle_{d \in ran(\varrho)}$$

completely describes $\mathcal{X}$ in the sense, that $\mathcal{X}$ and $\mathcal{A}(\mathcal{X})$ are mutually definable from each other. Further, at the same time, $\mathcal{A}(\mathcal{X})$ can be treated as a model for an appropriate first order language.

Recall e.g. from [1], that the metric space $\mathcal{X}$ is defined to be *totally bounded* if and only if for all positive $\varepsilon \in \mathbb{R}$ there exists a finite family of ε-balls of $\mathcal{X}$ that covers X. For completeness, we recall the (rather standard) formal definitions below.

Definition 2.1: Let $\gamma \in \mathbb{R}^+$. A family $\{B_i : i \in I\}$ of γ-balls is defined to be a γ-net iff it covers X, that is,

$$X = \bigcup_{i \in I} B_i.$$

Using this terminology, a metric space $\mathcal{X}$ is defined to be totally bounded iff for all positive $\gamma \in \mathbb{R}$ there exists a finite γ-net in $\mathcal{X}$.

As it is well known, $\mathcal{X}$ is compact iff it's metric is totally bounded and complete (i.e. every Cauchy sequence is convergent in $\mathcal{X}$). For further details we refer to [1], as well. It is well known, that every finite metric space is compact.

Let $\mathcal{A} = \langle A, R_i \rangle_{i \in I}$ be any first order structure, let $X \subseteq Y \subseteq A$ be arbitrary (finite or infinite) and let Δ_1, Δ_2 be sets of first order formulas in the language of $\mathcal{A}$. As usual, $S(Y)$ denotes the set of types over Y and v denotes the unique free variable of formulas in elements of $S(Y)$. For further explanation for the notation we refer e.g. to [13]. According to Definition I.2.6 of [13], a type $p \in S(Y)$ is (Δ_1, Δ_2)-splitting over X iff there exist $\overline{b}, \overline{c} \in Y$ and $\varphi \in \Delta_2$ such that

$$tp_{\Delta_1}(\overline{b}/X) = tp_{\Delta_1}(\overline{c}/X)$$

but

$$\varphi(v, \overline{b}),\ \neg\varphi(v, \overline{c}) \in p.$$

This motivates the following "approximate version" of splitting in the context of metric spaces which we recall from [9] (see also [10]).

Definition 2.2: Let $\mathcal{X} = \langle X, \varrho \rangle$ be a metric space, let $A \subseteq B \subseteq X$ be arbitrary (finite or infinite), let p be a Δ-type over B in $\mathcal{A}(\mathcal{X})$ and let ε, δ be non-negative real numbers. Then we say, that p is (ε, δ)-splitting over A iff there exist $c_0, c_1 \in B$ such that for all $a \in A$ we have

$$|\varrho(a, c_0) - \varrho(a, c_1)| < \delta$$

but whenever b realizes p, we have

$$|\varrho(b, c_0) - \varrho(b, c_1)| \geq \varepsilon.$$

Keeping the notation introduced in the above definition, intuitively $p = tp_{\Delta}^{\mathcal{A}(\mathcal{X})}(b/B)$ is (ε, δ)-splitting over A if and only if there exists $c_0, c_1 \in A$ such that c_0 and c_1 are "indiscernible from the viewpoint of A modulo δ", but b "distinguishes

them modulo ε".

Jumping back to model theory, let κ be a cardinal. An increasing sequence of types $\langle p_i : i < \kappa \rangle$ is defined to be a splitting chain iff for all $i < \kappa$, p_{i+1} is splitting over the domain of p_i. Further, by Lemma I.2.7 of [13], if a first order theory T is λ-stable for some infinite cardinal λ, then in each model of T, the length of any splitting chain of types is smaller than λ. An easy compactness argument yields, that if $\mathcal{A}$ is a stable structure, and Δ is a finite set of formulas, then the length of each (Δ, Δ)-splitting chain of Δ-types in $\mathcal{A}$, is finite.

In the context of metric spaces, the following analogous result has been established in Theorem 5 of [9]: if $\mathcal{X}$ is a totally bounded metric space and Δ is the set of atomic formulas of the language of $\mathcal{A}(\mathcal{X})$ then for all $\varepsilon \in \mathbb{R}^+$ there exist $\delta \in \mathbb{R}^+$ and $N \in \mathbb{N}$ such that if $\langle p_i, i < m \rangle$ is a strictly increasing sequence of (ε, δ)-splitting Δ-types in $\mathcal{A}(\mathcal{X})$, then $m \leq N$; particularly, each (ε, δ)-splitting sequence of Δ-types in $\mathcal{A}(\mathcal{X})$ has finite length (in fact, N is a common upper bound for them). Thus, if $\mathcal{X}$ is totally bounded, then $\mathcal{A}(\mathcal{X})$ shows some stability properties. In fact, according to the next theorem, in this case, $\mathcal{A}(\mathcal{X})$ is Δ-stable. However, to state and prove the next theorem, we need to recall the following definition from [9]:

According to Definition 6 of [9], if $a \in X$ and $\varepsilon, \delta \in \mathbb{R}^+$ then $A \subseteq X$ is defined to be an (ε, δ)-basis for a iff for any $B \subseteq X - \{a\}$ with $A \subseteq B$, the type

$$tp^{\mathcal{X}}(a/B)$$

does not (ε, δ)-split over A.

Theorem 2.3: If $\mathcal{X} = \langle X, \varrho \rangle$ is a totally bounded metric space, then

(1) For all $\varepsilon \in \mathbb{R}^+$ there exist $\delta \in \mathbb{R}^+$ and a finite set $A \subseteq X$ such that A is an (ε, δ)-basis for all $a \in X$ (we emphasize, that A does not depend on a).

(2) $\mathcal{A}(\mathcal{X})$ is Δ-stable.

Proof. To show (1), assume $\mathcal{X} = \langle X, \varrho \rangle$ is a totally bounded metric space.Then (1) is the same, as Theorem 9 [9].

To show (2), for all $n \in \mathbb{N}^+$ choose δ_n and a finite A_n such that A_n is an $(\frac{1}{n}, \delta_n)$-basis for all $a \in X$ (according to the previous paragraph, such δ_n and A_n exist). Now let $\mathcal{A}$ be any elementary extension of $\mathcal{A}(\mathcal{X})$ and let $Y \subseteq A$ with $|Y| \leq 2^{\aleph_0}$. We shall show, that there are at most $2^{\aleph_0}$ many Δ-types over Y in $\mathcal{A}$. Enlarging Y if necessary, we may assume $A_n \subseteq Y$ holds for all $n \in \mathbb{N}$. Since $\mathcal{A}$ is an elementary extension of $\mathcal{A}(\mathcal{X})$, it follows, that for any $a \in A$ and $Y \subseteq B \subseteq A - \{a\}$, the type

$$tp^{\mathcal{A}}(a/B)$$

does not $(\frac{1}{n}, \delta_n)$-split over Y (because for a fixed n, $(\frac{1}{n}, \delta_n)$-splitting is first order expressible in the language of $\mathcal{A}(\mathcal{X})$). As $A^* := \cup_{n \in \mathbb{N}} A_n$ is countable, there are at most $2^{\aleph_0}$ many types over A^*. Hence, it is enough to show, that for $a, b \in A - Y$,

$$\text{if } tp_\Delta(a/A^*) = tp_\Delta(b/A^*) \text{ then } tp_\Delta(a/Y) = tp_\Delta(b/Y).$$

Assume, seeking a contradiction, that $tp_\Delta(a/A^*) = tp_\Delta(b/A^*)$, but $tp_\Delta(a/Y) \neq tp_\Delta(b/Y)$. Then, there exists $c \in Y$ such that $\varrho(a, c) \neq \varrho(b, c)$. Fix $n \in \mathbb{N}^+$ with $\frac{1}{n} < |\varrho(a, c) - \varrho(b, c)|$. But this is impossible, because A_n (and hence A^* as well) is an $(\frac{1}{n}, \delta_n)$ basis for c. This contradiction completes the proof. ∎

Now we turn to show a kind of weak converse of the above theorem, which is the main result of this section (and the main theoretical result of the paper).

Theorem 2.4: Let $\mathcal{X} = \langle X, \varrho \rangle$ be a metric space such that $\mathcal{X}$ is dense in itself and the range of ϱ is bounded. Then the following are equivalent.

(1) $\mathcal{X} = \langle X, \varrho \rangle$ is totally bounded;

(2) $\mathcal{A}(\mathcal{X})$ is Δ-stable in the following, strong sense: for all $\varepsilon \in \mathbb{R}^+$ there exist $\delta \in \mathbb{R}^+$ and a finite set $A_{\varepsilon,\delta} \subseteq X$ such that

$$(*) \quad A_{\varepsilon,\delta} \text{ is an } (\varepsilon, \delta)\text{-basis for all } a \in X.$$

Proof. First we note that (1) implies (2) by Theorem 2.3 (we also note, that in this direction we don't use the assumptions that $\mathcal{X}$ is dense in itself and the range of ϱ is bounded).

To prove the converse, assume (2). Let $\varepsilon \in \mathbb{R}^+$ be arbitrary; we shall show, that there exists a finite 3ε-net in $\mathcal{X}$. Choose δ and $A_{\varepsilon,\delta}$ that satisfy (2). By assumption, the range of ϱ is bounded, say C is an upper bound for it. Consider the real $[0, C]$ interval and its $|A_{\varepsilon,\delta}|^{th}$ power

$$\mathcal{Y} := [0, C]^{|A_{\varepsilon,\delta}|}$$

as a subspace of $\mathbb{R}^{|A_{\varepsilon,\delta}|}$. Clearly, $\mathcal{Y}$ is a compact space, hence it has a finite δ-net

$$\{B(u_i, \delta) : \ i \in I\}.$$

For $c \in X$ define $t(c) \in \mathbb{R}^{A_{\varepsilon,\delta}}$ to be the vector

$$t(c) = \langle \varrho(c, x) : x \in A_{\varepsilon,\delta} \rangle.$$

Let

$$I' = \{i \in I : (\exists a \in X) t(a) \in B(u_i, \delta)\}$$

and for all $i \in I'$ choose $a_i \in X$ so that

$$t(a_i) \in B(u_i, \delta).$$

Finally, let $B = \{a_i : i \in I'\}$. Clearly, B is finite. Further, for any $c \in X$ there exists $i \in I$ such that

$$t(c) \in B(u_i, \delta).$$

Then c witnesses $i \in I'$ and in addition,

$$||t(c) - t(a_i)||_2 < \delta.$$

It follows, that for all $a \in A_{\varepsilon,\delta}$ we have

$$|\varrho(a, c) - \varrho(a_i, c)| < \delta.$$

Summing up we have shown, that for all $c \in X$ there exists $b \in B$ such that for all $a \in A_{\varepsilon,\delta}$ we have

$$(**) \quad |\varrho(a, c) - \varrho(b, c)| < \delta.$$

We claim, that

$$\{B(b, 3\varepsilon) : b \in B\}$$

covers X, that is, it is a finite 3ε-net, as desired. To see this, let $c \in X$ be arbitrary. Then, there exists $b \in B$ such that $(**)$ holds. Further, as $\mathcal{X}$ is dense in itself, there exists $d \in X$ with $\varrho(c, d) < \varepsilon$. By $(*)$, $A_{\varepsilon,\delta}$ is an (ε, δ)-basis for d, hence

$$|\varrho(c, d) - \varrho(b, d)| < \varepsilon,$$

particularly, $\varrho(b, d) < 2\varepsilon$. But then,

$$\varrho(c, b) \leq \varrho(c, d) + \varrho(d, b) < 3\varepsilon,$$

that is, $c \in B(b, 3\varepsilon)$. Consequently,

$$\{B(b, 3\varepsilon) : b \in B\}$$

is a finite 3ε-net. As ε was arbitrary, this completes the proof. ∎

3. A Similarity Detecting Algorithm

In this section we provide an application inspired by the theoretical results presented in the previous section. This section is based on the investigations initiated in [11].

Let $\mathcal{X} = \langle X, \varrho \rangle$ be a metric space and let $A \subseteq X$ be a given set. We have the following intuitive picture in our mind: X is the set of all instances of an abstract data type and ϱ measures similarity between the elements of X: if $\varrho(x, y)$ is "small" for some $x, y \in X$ then we say, that x and y are "similar enough" to each other. More concretely, we fix $\varepsilon \in \mathbb{R}^+$ and consider it as an amount of inaccuracy one can tolerate. Then "x and y are similar enough" means $\varrho(x, y) < \varepsilon$.

More formally, the similarity detecting problem for $(\mathcal{X}, A, \varepsilon)$ is the following: given an input $x \in X$ find $a \in A$ such that $\varrho(x, a) < \varepsilon$. The problem is, that A may be huge and computing ϱ for two particular points may be slow.

As we mentioned, our goal in this section is to propose and analyze an algorithm that can be used to handle the above problem efficiently. To do so, we start by recalling the following notation.

If $\mathcal{X}$ is a totally bounded metric space, then $\nu(\mathcal{X}, \gamma)$ denotes the smallest cardinality κ for which there exists a κ-sized γ-net of $\mathcal{X}$.

For a given $\varepsilon \in \mathbb{R}^+$ let

$$N := 6 \cdot \nu(\mathcal{X}, \frac{\varepsilon}{30}) \cdot \nu(\mathcal{X}, \frac{\varepsilon}{12}).$$

Suppose $\mathcal{X}$ is a (finite or infinite) totally bounded metric space. Then, according to Theorem 10 of [9],

(1) if $\mathcal{X}$ does not contain isolated points, then there exist $n \leq N$ and an ε-map $f : X \to \mathbb{R}^n$ such that, for all $x, y \in X$ we have

$$||f(x) - f(y)|| \leq \sqrt{n}\varrho(x, y),$$

in particular, f is continuous.

(2) if $\mathcal{X}$ has countably many isolated points, then there exist $n \leq 1 + N$ and a continuous ε-map $f : X \to \mathbb{R}^n$.

(3) if $\mathcal{X}$ is compact, then there exist $n \leq 1 + N$ and a continuous ε-map $f : X \to \mathbb{R}^n$,

further the ε-map is effectively constructed in all cases above.

Now we can sketch our similarity detecting algorithm as follows (immediately after sketching the algorithm, we comment and further explain its crucial steps).

(Step 1) For a given $\varepsilon' \in \mathbb{R}^+$ find $n \in \mathbb{N}$ and a continuous ε'-map $f : X \to \mathbb{R}^n$;

(Step 2) compute $B := \{\langle f(a), a \rangle : a \in A\}$;

(Step 3) for an input $x \in X$ find $b =: \langle u, v \rangle \in B$ for which the usual Euclidean distance $||f(x) - u||$ is minimal;

(Step 4) if (for $b = \langle u, v \rangle$ computed in (Step 3) above) we have

$$||f(x) - u|| < \varepsilon',$$

then the output is v; otherwise there are no elements of A which are similar enough to x.

In (Step 1) above, n and f can be constructed in an algorithmic way, because in the proof of Theorem 10 of [9], the ε-maps have been constructed effectively for all ε. For (Step 2) we note, that

$$\{x : \exists y \langle x, y \rangle \in B\} = \{f(a) : a \in A\} \subseteq \mathbb{R}^n.$$

So, in (Step 3), instead of working with the original distance ϱ we are working with Euclidean distances that can be computed relatively quickly, provided that n is small enough.

A particularly important special case of the general similarity detecting problem is, when X is a subset of a large dimensional Euclidean space (and ϱ is the corresponding Euclidean distance), that is, if $X \subseteq \mathbb{R}^k$ for a large k. This special case will be called *dimension reduction*. We make the following remarks:

- Note, that (Step 1) and (Step 2) in the above

sketch are preparatory: they should be performed only once at the beginning; if we are searching similar objects many times, then the cost of (Step 3) will dominate.

• If the database A changes in time, then according to (Step 2) we can quickly modify B, as well.

Now we turn to study the dimension reduction problem, that is, the similarity detecting problem, when A is a subspace of a large dimensional Euclidean space $\mathcal{X} = \mathbb{R}^k$. Of course, the critical point in our algorithm is (Step 1) in which we have to find n and f. According to Theorem 10 of [9] they exist and f can be effectively constructed from n and from given $(\frac{\varepsilon}{30})$-nets and $(\frac{\varepsilon}{12})$-nets of A. We have to choose n as small as we can in order to make our algorithm more efficient. The rest of the present section is devoted to investigate the choice of n for dimension reduction.

Analyzing the proof of Theorem 10 of [9], one concludes, that n becomes smaller whenever one is able to find smaller $(\frac{\varepsilon}{30})$-nets and $(\frac{\varepsilon}{12})$-nets of A. More generally, for given $\delta \in \mathbb{R}^+$ and $k \in \mathbb{N}$ one has to find a k-sized δ-net of A (if such exists).

The natural approach would be, that for a given δ one tries to find δ-nets of A with as small cardinalities k as possible. However, this do not would be an efficient method, as such an approach would be equivalent to solve NP-hard problems of computational geometry, and cluster analysis; for further details in that direction we refer to [7] and the references therein. Instead, we propose to fix k and estimate δ for which there exists a k-sized δ-net in A; then consecutively increasing k we will obtain a decreasing sequence of the corresponding $\delta = \delta_k$ and we increase k until δ_k will be sufficiently small.

For a fixed k and a given finite $A \subseteq \mathbb{R}^m$ the well known k-center problem is to find a k-element subset $B \subseteq A$ such that

$$max_a min_b\{||b-a|| : a \in A\}$$

is as small as possible (for further details we refer e.g. to [5]). This problem is related to cluster analysis and to the inverse shortest path problem. According to [3], the k-center problem (already in the Euclidean plane $\mathbb{R}^2$) is known to be NP-complete as well (for more recent related investigations we refer to [12] and [14]). Hence, instead of exact solutions, it is more practical to search for suboptimal, approximate solutions. Indeed, there is a classical approximate solution for the k-center problem due to Gonzalez [4]; we will briefly recall this.

For a metric space $\mathcal{X} = \langle X, \varrho \rangle$ and $A \subseteq X, b \in X$ the standard definition of the distance of A and b is

$$\varrho(A,b) = inf\{\varrho(a,b) : a \in A\}.$$

Then the farthest path transversal sequence of a finite metric space is defined as follows: $a_0 \in X$ is arbitrary, and if a_j has already been defined for all $j < i$, then a_i is a point x in X for which

$$\varrho(\{a_j : j < i\}, x)$$

is maximal. Now we recall Gonzalez's idea presented in [4]: fix $k \in \mathbb{N}$ and let

$$r = \varrho(\{a_j : j < k\}, a_k).$$

Observe, that

(i) for any $i \neq j < k$ we have $\varrho(a_i, a_j) \geq r$

and

(ii) for any $x \in X$ there exists $i < k$ such that $\varrho(a_i, x) \leq r$.

By (ii), $\{B(a_i, r) : i < k\}$ is an r-net of size k. Further, Suppose $\{B(b_i, r') : i < k\}$ is another r'-net. Then, by the Pigeon-Hole principle, there would exist $i \neq j \leq k$ and $l < k$ such that $a_i, a_j \in B(b_l, r')$, therefore

$$r \leq \varrho(a_i, a_j) \leq 2 \cdot r'.$$

In another words, the minimal radius r' of a k-sized r'-net is at least $\frac{r}{2}$. Therefore r constructed above, is a 2-approximation of the minimal radius of a k-sized net.

Based on the above observations, our dimension reducing algorithm is built up from an initialization part and from a searching part; these may be summarized as follows (as before, comments and explanations will be provided immediately after describing these algorithms).

Initializing part.

Input: a finite set $A \subseteq \mathbb{R}^n$ and $\varepsilon \in \mathbb{R}^+$.

1. Choose an arbitrary $a_0 \in A$ and let $k = 1, \varepsilon' = 1 + \varepsilon$.
2. While $\varepsilon' > \varepsilon$ and $k < n$ Do
3. Let $r = max_{x \in A} \varrho(\{a_j : j < k\}, x)$.
4. Let $a_k \in A$ be such that $r = \varrho(\{a_j : j < k\}, a_k)$.
5. Let $\varepsilon' := 2r$.
6. Let $k = k + 1$.
7. End(Do).
8. Let $f : \mathbb{R}^n \to \mathbb{R}^k$ be the function that maps each $x \in \mathbb{R}^n$ onto

$$f(x) := \langle \varrho(x, a_0), ..., \varrho(x, a_{k-1}) \rangle.$$

9. Compute a list enumerating

$$B = \{\langle f(a), a \rangle : a \in A\}.$$

In order to keep notation simpler, we will denote the list enumerating B by B, as well.

Searching part.

Input: $x \in \mathbb{R}^n$.

1. let $m = 1$;
2. While $m \leq length(B)$ Do
3. let $\langle u, v\rangle$ be the m^{th} element of B;
4. let $d := ||f(x) - u||$;
5. if $d \leq \varepsilon/2$ then v is an output End(if);
6. if $d > \sqrt{k}\varepsilon$ then v is not an output End(if);
7. if $\varepsilon/2 < d \leq \sqrt{k}\varepsilon$ then
8. if $||x - v|| \leq \varepsilon$ then v is an output End(if);
9. if $||x - v|| > \varepsilon$ then v is not an output End(if);
10. End(if);
11. let $m := m + 1$;
12. End(Do).

We conclude this section by making some remarks on the Initializing and on the Searching parts.

Remarks on the Initializing part.

In step 1, ε' may be chosen to be an arbitrary real number greater than ε.

Suppose the algorithm has already computed $\{a_j : j < k\}$ for some k. Then, according to step 3, $\{B(r, a_j) : j < k\}$ is an r-net. Let $f_k : \mathbb{R}^n \to \mathbb{R}^k$ be the function that maps each $x \in \mathbb{R}^n$ onto

$$f_k(x) := \langle \varrho(x, a_0), ..., \varrho(x, a_{k-1})\rangle$$

(so, according to step 8, f is f_k for the last (largest) value of k). It follows from Lemma 1 of [9], that if $x, y \in A$ are such that $\varrho(x, y) > 2r$ then $||f_k(x) - f_k(y)|| > r$, or equivalently,

$$||f_k(x) - f_k(y)|| \leq r \text{ implies } \varrho(x, y) \leq 2r.$$

Thus, intuitively, the smaller is r, the "intermediate function" f_k is "more injective". According to step 5, ε' can be regarded as an estimation of "non-injectivity" of f_k and the algorithm is keep going whenever the value of ε' exceeds ε (the tolerable amount of inaccuracy given in the input). At the end we have

$$(*) \quad \text{if } ||f(x) - f(y)|| \leq \varepsilon/2 \text{ then } ||x - y|| \leq \varepsilon.$$

Further, by Lemma 1 of [9], for all k and $x, y \in \mathbb{R}^n$ we have

$$||f_k(x) - f_k(y)|| \leq \sqrt{k} \cdot ||x - y||,$$

particularly,

$$(**) \quad \text{if } ||f_k(x) - f_k(y)|| > \sqrt{k}\varepsilon \text{ then } ||x - y|| > \varepsilon.$$

So $(*)$ and $(**)$ can be summarized in the following three cases: let $x, y \in \mathbb{R}^n$ be arbitrary and let $d := ||f(x) - f(y)||$.

- if $d \leq \varepsilon/2$ then $||x - y|| \leq \varepsilon$;
- if $d > \sqrt{k}\varepsilon$ then $||x - y|| > \varepsilon$;
- if $\varepsilon/2 < d \leq \sqrt{k}\varepsilon$ then we have to compute $||x-y||$ in order to determine whether $||x-y|| \leq \varepsilon$. This is what we are doing in the Searching part.

As A is a finite set, the Initializing part always terminates. In fact, because of Step 2, it terminates after at most n many iterations of steps 3-6. If $k = n$ after Initializing, then this method is unable to reduce the dimension.

It is straightforward to see, that the number of steps of the Initializing part is proportional with $|A|^2$ in the worst case. The precise number of required steps strongly depends on ε and the structure of A, hence, at that level of generality we cannot improve the estimation for the time complexity of the Initializing part. However, we conjecture, that in particular situations, by a careful choice of ε, the number of required steps may be kept in a reasonably small level. We are planning to implement and test our algorithm on real life databases (such investigations are in progress at the moment).

Remarks on the Searching part.

It may happen, that there are several $\langle u, v_0\rangle, ..., \langle u, v_m\rangle \in B$ for which $||f(x) - u|| \leq \frac{\varepsilon}{2}$ (where x is the input). According to the choice of k in the Initializing part, we have $\varrho(v_i, v_j) \leq 2r \leq \varepsilon$ for all $i, j < m$, where $r = max_{x\in A}\varrho(\{a_j : j \leq k\}, x)$.

The number of steps in the Searching part is proportional with $|A|$, but when we compute $||f(x) - u||$, we are using the distance function of the k-dimensional Euclidean space. As k may be substantially smaller than n, this method may be more efficient than checking the elements of A step-by-step with the distance function of the n-dimensional Euclidean space. Further, because the Searching part is essentially a minimum-searching problem, it seems possible to accelerate Step 1 further by applying well known methods of algorithm theory or operation research.

4. Conclusion

In section 2 we investigated stability properties of the relational structures associated to totally bounded metric spaces. More concretely, in Theorem 2.3 we reconstructed a proof for the statement, that the relational structure associated to a totally bounded metric space is Δ-stable. Further, in Theorem 2.4 we proved a partial converse of the previous statement: a dense in itself metric space with a bounded metric is totally bounded iff its associated relational structure satisfies a strong stability condition.

Inspired by these theoretical results, in Section 3 we proposed a similarity detecting algorithm. The similarity between objects had been described by a metric ϱ. Our method is based on two parts: the Initializing and the Searching parts. In the special case, when the metric ϱ is the usual Euclidean distance of a large dimensional Euclidean space $\mathbb{R}^n$, our method can be considered as a way of reducing dimension. In this particular case we analyzed the time complexity of the Initializing and the Searching parts.

In the future we are planning to implement our algorithm and test it in real life databases.

REFERENCES

[1] Engelking, R., *General Topology*, Heldermann Verlag, Berlin, (1989).

[2] Faloutsos, C. and Lin, K., *FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets*, ACM., Vol. 24, No. 2, 163–174, (1995).

[3] R. J. FOWLER, M. S. PATERSON AND S. L. TANIMOTO, *Optimal packing and covering in the plane are NP-complete,* Information Processing Letters, Vol. 12, No. 3, 133-137, (1981).

[4] T. F. GONZALEZ, *Clustering to minimize the maximum intercluster distance,* Theoretical Computer Science, 38: 293-306, (1985).

[5] S. HAR-PELED, *Geometric Approximation Algorithms,* American Mathematical Society, Boston, US, (2011).

[6] Hjaltason, G.R. and Samet, H., *Contractive embedding methods for similarity searching in metric spaces*, Technical report, Computer Science Department, Center for Automation Research, Institute for Advanced Computer Studies, University of Maryland, (2000).

[7] V. Marianov and H. A. Eiselt editors, *Foundation of Location Analysis*, Springer Verlag, US, (2011).

[8] J. R. Munkres, *Topology*, Prentice Hall, US, (2000).

[9] G. SÁGI, *Almost injective Mappings of Totally Bounded Metric Spaces into Finite Dimensional Euclidean Spaces,* Advances in Pure Mathematics, 9. pp. 555-566 (2019).

[10] G. Sági and D. Nyiri, *On embeddings of finitw mwtric spaces*, in: the Proceedings of the 13th International Scientific Conference on Informatics (editors: V. Novitzka, S. Korečko and A. Szakál), pp. 227–231, IEEE, (2015).

[11] G. Sági and K. Al-Sabti, *Totally bounded metric spaces and similarity detecting algorithms*, in: the Proceedings of the 15th International Scientific Conference on Informatics (editors: W. Steingartner, S. Korečko and A. Szakál), pp. 338–342, IEEE, (2019).

[12] J. Satyabrata and P. Supantha,*Covering and packing of rectilinear subdivision,* WALCOM: algorithms and computation, 381–393, Lecture Notes in Comput. Sci., 11355, Springer, Cham, (2019).

[13] Shelah, S., *Classification Theory*, North–Holland, Amsterdam (1990).

[14] R. Zahed and T. M. Chan, *A clustering-based approach to kinetic closest pair,* Algorithmica Vol. 80, No. pp. 10, 2742–2756, (2018).

Classification Rules for Temporal Databases

Kvet, Michal and Matiaško, Karol

Abstract: ***Temporal database management is currently an inevitable part of information technology. Intelligence and robustness of the systems are enhanced by the decision making reflecting the current data images, as well as historical data and future plans. Many times, existing solutions do not provide sufficient power, because they cannot be unambiguously identified and classified. In this paper, we deal with temporal architectures highlighting granularity and propose own complex classification rules to determine individual access principles, technology, architecture, and security aspects. The classification consists of 14 evaluated aspects. In this paper, discussion about the undefined states and reliability provided by the consistency is done, as well.***

Index Terms: ***data integrity, data rating, temporal classification, temporal granularity, undefined states***

1. INTRODUCTION

THE importance of the data in the industrial environment forms the inevitable part for the control systems, management, decision making, and problem identification. Their significance even sharpens the aspect of intelligence of the current information systems. It is, therefore, non-suitable to deal only with currently valid data, but the processing must be shifted to the temporal database system level. Current database technology allows you to manage and evaluate the data during the whole life cycle. The important role just plays the efficiency of the whole process covering the storage principles, architecture, query management, and reliability. There are many technologies, structures, and access principles surrounding and covering databases. However, there is no complex classification resulting in the necessity to describe an environment step by step in the research, as well as in the industrial manner. This paper aims to fill this negative aspect and propose complex classification rules defining the structure, access principles, and operations.

Manuscript received May 11, 2020.

Michal Kvet, Karol Matiaško, Faculty of Management Science and Informatics, Zilinska univerzita v Ziline, Slovakia. E-mail: Michal.kvet@uniza.sk, Karol.matiasko@uniza.sk

Thanks to that, temporal technology can be described just with a few characteristics. This paper chronologically describes and evaluates temporal principles from history up to current trends. It tends to summarize knowledge and technology regarding temporality. We propose classification rules for temporal databases shifting the management from the conventional principles, where just current valid data are reflected, to the temporal aspect, modeling the whole time-spectrum and object states evolution.

The evolution of the temporal paradigm started just after the first releases of the conventional relational database systems. Conventional databases are characterized by storing only currently valid data, which means that historical images cannot be obtained the the definition. Thus, if any data tuple is updated, the original value is replaced by the newer version, and original ones are thrown away. Later, developers and researchers concluded, that historical images would provide relevant added value for the processing, management, and evaluation. Industry and machine parameter values can be optimized to reach better performance resulting in reducing costs, increasing performance, and reaching a better score.

The first temporal approaches were really simple and originating from the transaction structures. Each data change is logged before the processing itself using the two-phase protocol [2]. This log consists of the reference to the encapsulating transaction, executed operation, and UNDO and REDO image of the changed tuple. From the temporal point of view, if the log files are accessible, it is possible to reconstruct the image, as it was in the past, although it is a very complicated and demanding process. In the standard environment, all log files form a cyclically linked list and are sequentially rewritten, if they do not cover relevant data of the active transaction. The Archiver background process allows you to copy the log file into a separate storage repository before its overwritten [1]. Thanks to that, historical values can be obtained. The process of reconstruction requires access to the current image, as well as loading and parsing all the log files, which were created by any

transaction executed after the timepoint that we are looking for. The main disadvantage is reflected in efficiency, namely it is not possible to evaluate directly whether a particular log file consists of data modifying the required object or not. As a consequence, the process is too demanding and time and resource consuming. Technically, it is possible to start not only with the current data image, but any time point from the backup can be used, however, in that case, the whole backup must be loaded into the system, which is not possible to accept in the operative decision making.

Flashback is another technology allowing you to automate the reconstruction process based on the transaction, time, or System Change Number (SCN). It offers to mark objects or individual attributes, which are monitored over time in the historical spectrum. Thanks to that, the size of data to be parsed and loaded is lowered. Specific data structures are used to optimize the whole process, as well [1] [5] [6].

The limitation of the previously mentioned techniques is just the time spectrum and efficiency. It is too complicated and resource-consuming, as well as it requires too much time. Moreover, it can manage only currently valid data and history. No information about plans, future set parameters, and settings can be stored resulting in the necessity to develop a separate structure for the data valid in the future, forcing to manage additional rules to ensure reliability and consistency of the data and the system itself.

Several architectures for managing temporal data have been developed over the years. They are, mostly, based on the object granularity.

This paper aims to propose and describe models based on their architectures of the attribute and synchronization groups. Whereas several principles and techniques can be applied, it is necessary to define the classification rules to cover the temporality. Therefore, the main part of the paper proposes its complex techniques that participated from [6] [8] [10]. Section 2 deals with the temporal aspects from the granularity point of view. In section 3, undefined states and attribute values are highlighted defining storage and access principles. In section 4, transaction management is described. It is, namely, not suitable to use conventional principles of transaction management, whereas consistency in a temporal manner should be emphasized. Therefore, we propose our technique of collision management. Section 5 covers the classification itself.

2. *Temporal Architectures – Granularity Aspect*

Each object in the relational database can be clearly distinguished by a unique identifier – primary key (PK), which can consist of one or more attributes. If the time spectrum attributes are added to the identifier, the temporal model is created. In that case, each object is delimited either by the identifier itself (ID), but also by the time spectrum expressing mostly the validity interval. Thus, the object is characterized by the multiple states, but in different time points or intervals forming individual versions. The time spectrum can be modeled by the begin (BD) and endpoints (ED) of the validity or by just one attribute (begin point). In that case, each newer version of the object automatically delimits the previous one. Generally, time spectrum expresses validity, but the transaction interval expressing time of transaction approval can be used (BD2, ED2), as well. In principle, multiple time spectra can be present. The characteristics of the model reflect the number of time spectrum zones located in the system. The uni-temporal model deals with one spectrum, the bi-temporal approach is characterized by two spheres to be handled. In general, the multi-temporal model can be identified. Data model principles are shown in fig. 1. Label ID defines a set of attributes defining the object itself. It uses the same principle as in a conventional database. Validity is the most often used temporal spectrum forming a uni-temporal table. It is defined by the BD and ED reflecting validity interval. Thus, the object state temporal identification (primary key – PK) is a composite consisting of three attribute sets – ID, BD, and ED. Note, that ID is itself a set and can consist of several attributes.

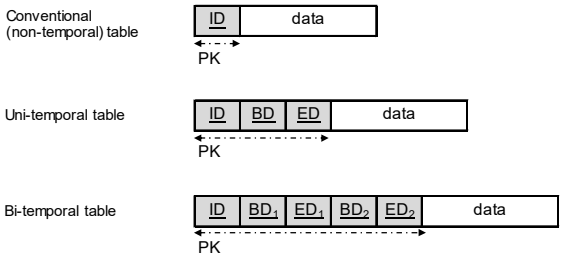


Figure 1.Object architecture [10]

If the main processed granularity is shifted to the column, attribute-oriented model is created. When using the previously described model – object-level – each change forces the system to create a completely new state. Thus, if some column values are not changed, original values are copied to newer states uncovering the storage and performance efficiency. Moreover, identification of real change requires to manipulate with two consecutive states. The attribute system encloses each attribute by its validity. Therefore, the complex data image arises as the projection of individual attributes

and their validity in the defined time interval. Thanks to that, the size of the whole structure is optimized, whereas only veritable change is present and stored. On the other hand, if several attributes are changed at once, each change forces the transaction manager to add a new row to the temporal layer. The architecture of the system is shown in fig. 2. It contains three layers dividing current and inactive data into separate levels. The first layer consists of only currently valid data, which can be queried directly outside the temporal system. Temporal evaluation and change management are covered by the hearth of the system – second layer with a temporal table. It manages all the changes and interconnects them to individual layers and objects. Non-actual values (historical and future plans) are separated in the third layer. Temporal transaction manager, as an extension of the background process set, covers the whole process and ensures consistency, as the transformation from the current data layer into historical or vice versa, from the future plans into the current states. Performance definition and comparison to other architectures can be found in [9].

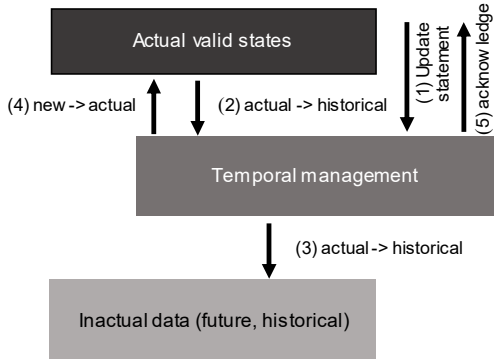


Figure 2.Column architecture [9]

The limitation of the attribute granularity is just the case of multiple attributes are updated synchronously. In that case, each attribute requires one insert operation in the temporal layer. Therefore, we introduce our system with synchronization groups. The data model is shown in fig. 3. Attribute reference is replaced by the expression data_val, which can express the attribute itself but can be linked to the temporal group covering multiple attributes. Thus, the attribute is not referenced directly, but in the temporal layer, an object reference to the synchronization group is located. Thanks to that, performance is ensured. Physical implementation and architecture can be found in [8] [14]. In this paper, we introduce an improved version of the management and data model.

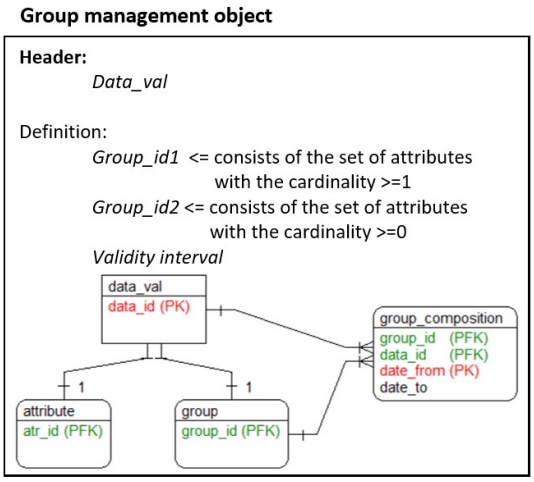


Figure 3.Group architecture

Another processing level is formed by the spatial aspect of the processing. In that case, however, it is not necessary to change the identification model of the data tuple, just new attributes defining spatial position are added forming Spatio-temporal architecture [15]. The database itself can be located either in the on-premise, but the cloud technology is widespread now. Current temporal trends are defined in [13]. Big data architecture is described in [16].

Our proposed architectures delimit the data by the temporal validity intervals. Many times, however, large objects (LOBs) are not covered by the temporal change monitoring. Temporal data archive characteristics and architecture dealing with the multimedia can be found in [17].

3. UNDEFINED STATES

As already mentioned in the previous sections, there can be several models and multiple approaches for dealing with the states themselves. If each object must be defined at any time point, an undefined state must be stated explicitly. The point is, how to model and express the undefined state. Generally, the undefined state does not automatically mean, that all of the attributes are unknown, respectively do not hold relevant data. Each object definition consists of the list of attributes, which can be undefined, but the object from the outer view can be considered as (partially) valid. There can be also a list of attribute sets, which decompose the object state resulting in a complete invalid object state.

Data to be stored in the database can be produced by various sources, data streams, or sensors. A typical problem is just the uncertainty of the communication channel, like wireless connection, etc. For these purposes, the input database connector is extended with a reliability module to cover the data correctness. If some data portion, attribute, or a whole object state is not defined, it must be clearly stated. The first representation is NULL expression, that the

values are not present. In our solution, we do not use such notation for several reasons. First of all, such value does not hold sufficient information value. What does it mean in practice, that the object, group, or just one attribute is NULL? Does it mean, that the system cannot obtain the value? Or that the input value cannot be evaluated as reliable? Or the value came to the system with latency? Moreover, NULL values are not part of the index structures, forcing the system to scan all data blocks sequentially [11] [12]. Shifting to the temporal spectrum, the problem is even sharper. From the NULL value in the time notation, we cannot even distinguish that the event has not occurred yet. Therefore, in many literatures and papers, MaxValueTime representation is used, physically modeled by the maximal date to be processed (31/12/9999). Once again, when managing data, a particular representation must be extracted from the real timepoint in the future. Based on the previous analysis, we concluded that it is necessary to develop our own solution for dealing with the undefined values. The principle is based on the memory object for each data type or the object definition. If the undefined value is present, a pointer to the memory object is used instead. Based on the definition, such a pointer can be part of the sorting and index itself. Undefined object definition is always present in the instance memory, it is created during the mounting process of the instance, managed by the background processes and released during the instance drop (closing operation). It is supervised by the System Monitor background process and it is part of the core. Thus, if corrupted or there is an attempt to drop it, the whole instance is automatically and immediately closed. Object state can be routed to the object part of the memory object (Undef_obj), the attribute is pointed to a particular position based on the data type. If the synchronization group is present, it is operated on the same principles as the undefined object itself. The architecture is composed of the attribute definition and object definition separately.

If the time validity interval is defined explicitly, a virtual calendar must be created for the defined data precision in time, by which the undefined time frames can be identified.

4. *Consistency Management*

Transaction in the temporal system is defined in the same manner as in the conventional approach – atomicity, consistency, isolation, and durability. When dealing with the consistency, the time spectrum must be handled, as well. It means, that each referenced object must cover the time interval of the child record in the hierarchy. Naturally, such an object can be defined by multiple states. There cannot be, however, time period during which the parent object is undefined or non-reliable. In terms of transactions, data collisions must be taken care of, too. As mentioned, each object is defined by no more than one valid state anytime, with emphasis on versioning and state corrections, as well. It requires a temporal manager to be extended to cover such functionality. In our proposed solution, we define four collision rules to ensure the reliability and consistency of the system.

1. Complete-reject automates the collision management by refusing the new state, which is in collision with the existing state.
2. Complete-approve is characterized by accepting the new plan, thus the existing state validity is shortened.
3. The partial-approve method is similar, in that case, the validity of the new state is shortened to limit the collision. This rule can be extended by the parameter shifting, which can hold value True or False. If True is selected, the end point of the new state validity is shifted to the right, so the original validity duration remains unchanged. Vice versa, if the False option is used, the original validity end point is applied.

The whole management is secured by our three-phase Commit protocol shown in fig. 5. BOT denotes Begin of Transaction, EOT expresses End of Transaction.

The following rules apply to this method:

- The transaction cannot change the data in the database until it is confirmed (before reaching RC point).
- The transaction cannot change the data in the database unless all conflicts between the time intervals of individual object images - states in time have been resolved.
- A transaction cannot be committed if transaction rules have also been applied that does not allow the modification of the validity period of existing states.

The advantage of this method is that in the event of an accident before reaching the RC point, none of the changes have been recorded in the database and therefore no changes are required. Another advantage is that if a collision situation occurs that does not allow the transaction to continue (e.g. Restricted rule is used), there is no need to modify the database as no changes have been applied yet.

If an accident occurs after sending an RC signal, then the solution is based on rolling back the transaction.

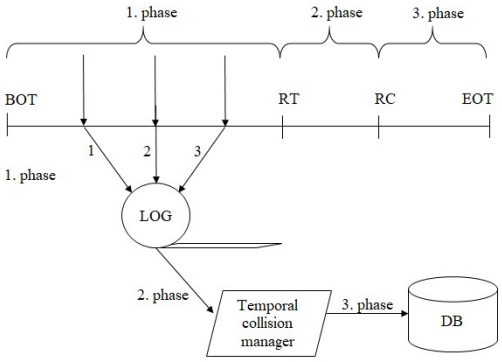


Figure 5.Three-phase Commit

5. CLASSIFICATION

Gradual extension of temporal approaches and time-oriented database systems brought the need for formalization and categorization of individual types. There was no organized form for managing types in terms of modeled granularity or system architecture, so in 2015 we decided to create normative classification rules [9] originating from [7], which consist of three types - α / β / γ /, where:

- **α** represents the type of database system used.
- **β** characterizes the type of temporal structure.
- **γ** denotes the type of online transaction processing (OLTP).

As part of the development of new temporal techniques and the definition of robust tools, to which we also contributed significantly through our scientific and experimental activities, it was necessary to extend the categorization to include additional attributes. Besides, the above definition has also been slightly modified - in the new system, we distinguish between non-timed (conventional) models and static attributes that were originally considered identical in terms of time management.

In this paper, we propose a complex extension and new classification rules. The definition consists of the following parts **A / B / C / D / E / F / G / H / I / K / M / O / P / R:**

A – database system type

- **N** - no database system support (e.g. file system).
- **R** - relational DBS.
- **H** - object-relational DBS.
- **O** - object-oriented DBS.
- **X** - DBS storing data in the form of XML documents.
- **N** - non-relational DBS.
- **x** - the unspecified type of DBS used.

B – Temporal dimensions

- **0 - S** - static data (codebooks, configuration parameters, constants, etc.)
- **0 - C** - conventional (non-timed) data - objects in which changes are not tracked over time.
- **1 - U** - uni-temporal data (records are limited by the validity).
- **1 - T** - uni-temporal data (records are delimited by the timestamp of insertion or status update - transactional validity without recording of the state validity limit itself).
- **2 - B** - bi-temporal data bounded by the validity and reliability expressed by the transactional validity.
- **2 - O** - bi-temporal data bounded by the validity and synchronization timestamp. This modeling method is used in systems with asynchronous processing in offline mode followed by synchronization with the central system (fig. 6). There are different times for the local system and the reflection in the central database node (synchronization timepoint).

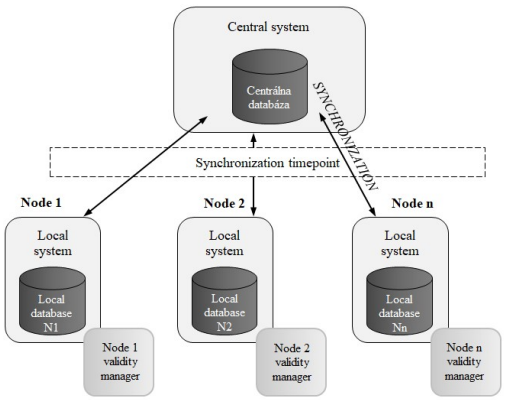


Figure 6. The architecture of the synchronization tool

- **3 - F** - multi-temporal data with three dimensions (validity, transactional validity, and timestamp of the transformation of the states valid in the future to the current ones).
- **3 - O** - multi-temporal data with three dimensions (validity, transactional validity, and synchronization timestamp).
- **4 - K** - multi-temporal data with four dimensions (validity, transactional validity, timestamp of the transformation of states valid in the future to the current and timestamp of synchronization).
- **x - M (x)** - multi-temporal access to data, where "x" represents the number of dimensions.

C – the type of granularity

- **O** - object granularity (any change in the temporal attribute causes a completely new state to be created; effective, if all individual changes are synchronized, they occur at

uniform times).

- **C** - column granularity (each change is made at the attribute level; the complete state of the object is created by composing individual images of the attributes).
- **G** - granularity at the sync group level.

D - representation of time on the axis - time spectra that are processed and evaluated in the temporal system (several options can be used):

- H - historical states.
- C - currently valid states.
- F - states of objects which validity begins in the future (the solution must include an automated projection of these plans into a set of current states).
- E - use the Epsilon (Ɛ) principle. Only significant changes in the values of individual attributes, respectively attribute groups. The value difference must be greater than Epsilon, otherwise, the validity of the original state will be prolonged (or retained).

E - Online transaction processing (OLTP)

- N - no transaction processing support.
- L - OLTP with logs.
- O - OLTP at the level of temporal objects.
- A - OLTP at the level of temporal attributes of the object.

F - definition of the way of the collision of states:

- R - restriction.
- P - partial validity.
- C - complete validity.
- W - notification (the solution is left to the user). Ignoring collisions can cause inconsistent states and database integrity violations.

G - definition of indices (structure, type, and characteristics of indices) [3] [4] [11]

H - index localization [8] [11]:

- **T** - data is stored in the same tablespace as the data itself.
- **S** - data is stored in another tablespace, optionally with a different data block size:
  - by the same size (8kB by default) inherited property of the main database structure,
  - 2, 4, 16, 32, 64kB data block.
- **D** - indexes are distributed to individual DDBS nodes.

I - represents the use of an own Flower Index Approach (FIA) to eliminate the impact of using the Table Access Full (TAF) method.

If no suitable index for the query is present, the system must use the TAF method to scan all blocks associated with the table regardless of the data, individual block holds. As a result, inappropriate, fragmented, and even empty blocks are loaded to the memory for the evaluation. To remove such an impact, we have proposed our own FIA approach. It uses the main index technique, that the leaf layer of the B+tree index (most often used database index) consists of the direct pointers to the data in the database (ROWID). ROWID is a unique row identifier and contains the address to the database file, block, and references the position of the row inside it. Our approach, however, uses only block level for the processing. Thus, only relevant blocks are loaded. If no ROWID points to the block, it is clear, that it does not hold data of the table. If the industrial environment, individual parameters, and object values are changed dynamically, the size of the row can vary resulting in creating fragmentations inside the block and the file itself. Our proposed solution, therefore, limits the impact of the fragmentation, no structure rebuilding is necessary to be executed. Thanks to that, resource extensive method can be omitted and performance does not degrade.

Individual records are located in the appropriate database blocks using pointers, so it is not necessary to sequentially check all the blocks associated with the table. The index so defined is permanently available in the server instance memory (as long as the STATUS instance is OPEN):

- **Y** - yes - FIA access can be used.
- **N** - no - use of FIA access is prohibited.

The principle of the proposed method is shown in fig. 7. Notice, that several ROWID values point to the same data block, which is then marked as already processed. After the loading process of the block into the memory, it is completely scanned to locate data.

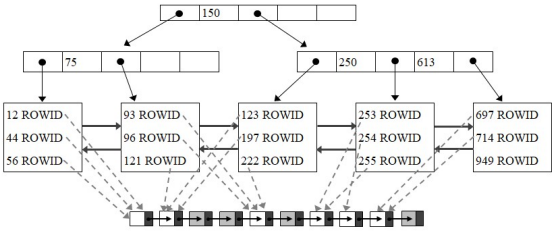


Figure 7.FIA index architecture

K - Expression of security rules with emphasis on audits:

- **A** - audit at the level of whole database objects (e.g. whole tables) = database audit.
- **T** - Trigger audit.
- **F** - processing of attributes = level audits =

fine-grained audit.

M - the architecture of the database system itself:

- **S** - Single instance.
- **R** - Real Application Cluster.
- **S** - Streams.
- **D** - Data Guard.
- **C** - Cloud.

O - selected integrity rule applied in time to reference the integrity and composition of objects (ISA hierarchy):

- **S** - the strict condition of coverage - the object in the role of "child" must be completely covered by the valid status of its "parent", respectively ancestor in the hierarchy. Thus, foreign keys are not only checked at the existing level of the object, to which the record is referenced. Individual time intervals of validity are monitored and evaluated, as well.
- **R** - relaxed coverage condition - reference integrity ignores validity intervals completely.

P - state classification

- **Exact** - each object must be defined by exactly one state at any time. In this case, undefined object states are also managed explicitly.
- **Max** - this approach is defined by a relaxed rule, just correctly defined states are recorded, undefined and incorrect states are calculated automatically, however, they are not stored in the database.

6. CONCLUSIONS

The conventional database approach does not bring sufficient power for intelligent information systems or management systems in the industry, whereas only current valid data are modeled, monitored, and evaluated. The temporal paradigm has brought the wide possibilities to manage data over the whole life spectrum of each object with emphasis on the information value. Thanks to that, it is easily and effectively possible to deal with decision making, creating prognoses, and complex analytics. In this paper, we deal with temporal architectures, we bring our own 3 level architecture of the column temporal granularity, in which each attribute is bordered by the time frame expressing modeled and processed time interval, mostly validity. Such architecture aims to limit the efficiency of object architecture and remove the necessity for storing the same data values many times. Another proposed temporal system interconnects object and column granularity by defining the temporal group. In this case, if multiple attribute values are changed during the same event, the synchronization group can be created and managed as the one composition, not individual attributes separately.

This paper deals also with the management of undefined states, the principle of modeling, and identifying such states. We prefer our object located in the instance memory, to which object states can point. Thanks to that, such values can be indexed and effectively-identified.

Individual states must be covered by the transaction rules focusing on integrity and consistency. Several rules are defined to limit the collisions of the states. The relational paradigm of the temporal system requires no more than one valid state for each object any time, therefore collisions must be dynamically identified and removed. Principles are based on removing new or existing state or by changing their validity intervals.

The core part of this paper is delimited by the classification rules of the temporal databases. In the past, there was no complex classification of the system and was very complicated to identify real principles, data flow, and management. Therefore, we proposed basic classification is 2015. In this paper, we extend the principles to cover temporality complexly. Transaction management, architecture, volatility, dimensions, collisions, index management and location, security, or integrity. We propose 14 layers defining each temporal system.

In the past, our emphasis will be on the data distribution in the temporal systems, data loading regarding block size, indices, and fragmentations. After the system and architecture development, new classifiers will be proposed to cover a distributed environment.

ACKNOWLEDGMENT

This article was created in the framework of the National project IT Academy – Education for the 21st Century, which is supported by the European Social Fund and the European Regional Development Fund in the framework of the Operational Programme Human Resources.

The work is also supported by the project VEGA 1/0089/19 Data analysis methods and decisions support tools for service systems supporting electric vehicles and the Grant system of the University of Zilina.

REFERENCES

[1] Ahsan, K., Vijay, P., "Temporal Databases: Information Systems", Booktango. 2014.

[2] Ashdown, L., Kyte T., "Oracle database concepts", Oracle Press, 2015.

[3] Avilés, G., et al., "Spatio-temporal modeling of financial maps from a joint multidimensional scaling-geostatistical perspective". In Expert Systems with Applications. 60, 280-293, 2016.

[4] Doroudian, M., et al., “Multilayered database intrusion detection system for detecting malicious behaviours in big data transaction”, IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), 2016.
[5] Erlandsson, M., et al., “Spatial and temporal variations of base cation release from chemical weathering a hisscope scale”. In Chemical Geology. 441, 1-13, 2016.
[6] Johnston, T., “Bi-temporal data – Theory and Practice”, Morgan Kaufmann, 2014.
[7] Johnston, T., Weis, R., “Managing Time in Relational Databases”, Morgan Kaufmann, 2010.
[8] Kvet, M., Matiaško, K., “Temporal Data Group Management”, IEEE conference IDT, 5.7. – 7.7.2017, 218-226, 2017.
[9] Kvet, M., Matiaško, K., “Transaction Management in Temporal System”, IEEE conference CISTI 2014, 18.6. – 21.6.2014., 868-873, 2014.
[10] Kvet, M., Matiaško, K., “Uni-temporal modelling extension at the object vs. attribute level”, IEEE conference UKSim 2014, 20.11 – 22. 11.2014, , 6-11, 2014.
[11] Kuhn, D., Alapati, S., Padfield, B., “Expert Oracle Indexing Access Paths”, Apress, 2016.
[12] Kumar, N., “Efficient data deduplication for big data storage systems”, In Advances in Intelligent Systems and Computing,714, 351-371, 2019.
[13] Lan, L., Shi, R., Wang, B., Zhang, L., Shi, J.: “A Lightweight Time Series Main-Memory Database for IoT Real-Time Services”, Lecture Notes in Computer Science, Volume 11894 LNCS, 2020, pp. 220 – 236.
[14] Li, S., Qin, Z., Song, H., “A Temporal-Spatial Method for Group Detection”, Locating and Tracking, In IEEE Access, 4. 2016.
[15] Moreira, J., Duarte, J., Dias, P.: “Modeling and representing real-world spatio-temporal data in databases”, Leibniz International Proceedings in Informatics, LIPIcs, Volume 142, 2019
[16] Ochs, A. R., et al.: “Databases to Efficiently Manage Medium Sized, Low Velocity, Multidimensional Data in Tissue Engineering”, Journal of visualized experiments JoVE, Issue 153, 2019.
[17] Saany, S. I. A., Rahman, M. N. A., Nasir, A. F.: Temporal based multimedia data archive, International Journal of Recent Technology and Engineering, Volume 7, Issue 5, 2019.

Evaluation of Static Analysis Methods of Python Programs

Gulabovska, Hristina; Porkoláb, Zoltán

Abstract: ***Static analysis is a method for detecting code smells and possible software bugs by examining the source code without executing the program. While we have considerable experiences for programming languages with static type system, especially for C, C++, and Java, languages with dynamic behavior requires different approaches. Python is an important programming language with a dynamic type system, used in many emerging areas, including data science, machine learning, and web applications. In this work we overview static analysis methods currently applied for Python, investigate their advantages and shortages, and highlight the restrictions of current tools and suggest further research directions to tackle these problems. We report our experiences applying static analysis methods on an open source Python software system where we found numerous issues confirmed by the developers. Based on these findings, we suggest refined configuration settings on static analysis tools.***

Index Terms: ***static analysis, symbolic execution, Python***

1. INTRODUCTION

PYTHON is one of the most rapidly emerging programming languages [38]. Being flexible and expressive, it is very popular to implement on Machine Learning and Cloud based systems among others. The popularity is partially derived from its dynamic behavior: Python is a dynamically typed programming language, i.e. a variable is just a value binded to a certain (variable) name, the value has a type but not the variable. Namely, one can assign a new value with a possibly different type to an existing variable. This would be a compile time error in statically typed languages such as Pascal, C, or Java, but allowed in Python. At the same time, Python is strongly typed, as operations may fail on an object when the operation is not defined on the actual type of the value held [39]. More dynamic features, such as calling methods dynamically, declaring dynamic attributes (using `getattr`), and others also increase the expressiveness and usability of the language.

Manuscript received ... This work was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002)

Hristina Gulabovska is a MSc Computer Science student in the Eötvös Loránd University (ELTE), Budapest, Hungary (e-mail: hristina@gulab.me).

Zoltán Porkoláb is an Associate Professor of the Eötvös Loránd University (ELTE), Budapest, Hungary (e-mail: gsd@elte.hu).

However, such dynamic behavior might be an obstacle when we try to validate the software systems written in Python. For languages with static type system, various static analysis methods exist and are used either as commercial [33], [36] or free, open source tools [6]–[8], [18]. Although there are promising projects for Python (see Section 3), they are significantly less expressive than their counterparts for C, C++ or Java languages.

In the recent years we experienced rapid development of static analysis tools and methods. Besides proprietary tools we found a fine number of open source projects with growing developers' communities [16], [43]. Static analysis is an important aspect in modern software development, addressed by various academic and industrial researches, and projects such as Intellectual outputs No. O1 and O2 of the Erasmus+ Key Action 2 (Strategic partnership for higher education) project No.2017-1-SK01-KA203-035402: "Focusing Education on Composability, Comprehensibility and Correctness of Working Software" [37], [40].

In this paper, we investigate the possible research directions towards more powerful static analysis tools for the Python programming language. In Section 2 we evaluate the applicable analysis methods based on their strength and weaknesses. In Section 3 we briefly overview the most important tools and research directions currently available for analyzing Python systems. We use two specific tools to compare the *Abstract Syntax Tree* (AST) based methods and the symbolic/concolic execution in Section 4. In Section 5 we evaluate static analysis on an open source software using the feedback from the developers of the software to confirm true and false positive results. Our paper concludes with Section 6.

2. STATIC ANALYSIS METHODS FOR SOFTWARE SYSTEMS

To verify the correctness of a software system we can choose between methods. The most common solution is to write test cases, either using white box or black box approach. Although testing is essential for modern software development, it is a costly and slow approach which efficiency greatly depends on the test coverage we achieve. The earlier a bug is detected, the lower the cost of the fix is [5], testing is not ideal in this aspect.

Alternatively, we might turn to analyzer tools that apply various validation methods to find po-

tential or actual misuses in the software. Dynamic analysis tools run the program in a special environment where they can detect incorrect, erroneous execution. However, tools such as *Valgrind* [27], or *Google Address sanitizer* [35] which work in run-time, evaluate the correctness of only those parts of the system which have actually been executed. Such *dynamic analysis methods* therefore require carefully selected input data, and they easily can miss certain corner cases.

In the case of static analysis, we do not run the software. Instead, the input of the static analyzers are the source code, and we apply various methods to find dangerous constructs without running the program. Static analysis is a popular method for finding bugs and code smells [4] as they do not depend on the selection of input data while they can (at least theoretically) provide full coverage of the code.

Most static methods apply heuristics, which means that sometimes they may *underestimate* or *overestimate* the program behavior. In practice, this means static analysis tools sometimes do not report existing issues what is called *false negative*, and sometimes they report correct code erroneously as a problem, which is called as *false positive*. Therefore, all reports need to be reviewed by a professional who has to decide whether the report stands. However, if the tool produces a large number of false positives, the necessary review process requires large efforts by the developers, meanwhile they also lose their trust in the analyser tool. Therefore, when in doubt, many tools rather choose to drop findings to minimize false alarms.

2.1. Pattern Matching

In this method the source code is first converted to a canonical format, and we match regular expression to every line in the source and reports each match. Although, this method seems to be very simple, its huge advantage is working on non-complete source, even if it cannot compile. Additional advantage is the low level of false positives, as well-written regular expressions have easy to predict results. Early versions of CppCheck [25] used pattern matching to find issues in C and C++ programs.

At the same time this method has several disadvantages too. As regular expressions are context free grammars, we are restricted to find issues based on information in the close locality of the problem. Thus, we could not use type information, name and overload resolution, and cannot follow function calls. As a summary, we can consider pattern matching based approaches as easy entry-level methods [26].

2.2. AST Matchers

To provide the necessary context information to the analysis, we can use the *Abstract Syntax Tree* (AST). AST is a usual internal data structure used by the front-end phase of the compilers [1]. Basically, the AST is a lossless representation of the program, frequently also decorated with type information and connections between declarations and their usage. This representation is suitable for catching errors that the simple pattern matching is unable to detect. Such AST based checks are usually relatively fast. Some rules can even be implemented using a single traversal of the AST. That makes it possible to implement such checks as editor plug-ins. Tools, such as the Clang Tidy [8] uses AST matching for most of its checks.

While the AST matcher method is more powerful than a simple pattern matching, it has some disadvantages too. To build up a valid AST requires a complete, syntactically correct source file. To resolve external module dependences we need some additional information not represented in the source itself, such as include path for C/C++ programs, CLASSPATH for Java or BASE_DIR in Python. That usually means, we have to integrate the static analysis tool into the build system which can be painful. Another shortage of the AST matcher method is that it cannot reason about the possible program states which can be dependent on input values, function parameters.

2.3. Symbolic execution

When executing *abstract interpretation* [11] the tool reasons about the possible values of variables at a certain program point. *Symbolic execution* [19], [20] is a path-sensitive abstract interpretation method. During symbolic execution we interpret the source code, but instead of using the exact (unknown) run-time values of the variables we use symbolic values and gradually build up constraints on their possible values. Memory locations and their connections are represented by a sophisticated hierarchical memory model [42]. A constraint solver can reason about these values and is used to exclude unviable execution paths. Most of the high-end proprietary analysis tools, such as CodeSonar [18], Klocwork [33], and Coverity [36], as well as open source products such as the Clang Static Analyzer [7], and Infer [6] use this method.

Symbolic execution is the most powerful method for static analysis as it makes profit from the program structure, the type system, the data flow information and is able to follow function calls. However, there is a price for this. To represent the internal state of the analysis, the analyzer uses a data structure called the *exploded graph* [31]. This graph is exponential in the number of control branches (conditions) of the program. This

could be critical, especially for loops, which are represented as unrolled set of conditions and statements. This factor makes symbolic execution also the most resource expensive (in memory and execution time) method.

2.4. Concolic execution

An interesting mixture of symbolic and concrete execution is called a *concolic* execution [34] and it targets this problem. The main idea is that we use concrete values for execution driven by symbolic execution. We start the execution with an arbitrary input value, we maintain both the symbolic execution state and a storage for the concrete values. Whenever the concrete execution takes a branch, the symbolic execution is directed toward the same branch. Then the constraint solver is used to negate the path conditions thus to choose a new concrete value to cover the other branch. [2].

The advantage of concolic execution is that the operations of the program state can be executed on concrete values, thus it could be implemented in more simple way and using less resources, while the SAT solver still helps to follow all the possible execution paths. Especially for languages such as Python, where the interpreter could evaluate the analyzed program making possible to call unmodeled external methods or using third party modules this approach is seriously extending the power of symbolic execution.

3. Analysis tools for Python

Compiler relies on static analysis to generate its warnings during compilation time. However, its primary task is to translate source code, and not to execute a full scale and costly static analysis. The Python compiler misses catching a number of common bugs and errors, therefore already existing third-party Python static analysis tools are aiming to cover the catches missed by the actual compiler. Among the most common actual Python static analysis tools, the following could be listed as the most reliable: PyLint [22], Pyflakes [28], flake8 [10], Frosted [12], Pycodestyle [32], and Mypy [21]. These tools are open-sourced, and some of them are still explicitly said to be in an experimental stage. They are using the AST method in order to statically evaluate the potential bugs and errors of the source code.

Currently, PyLint is seen as the most popular Python static analysis tool, which is free and capable of not only catching logical errors, but also warns regarding the specific coding standards. In PyLint, there is a possibility to write custom rules, too. There are three types of possible custom rules: Raw checkers (analyzing each module as a raw file stream), Token checkers (using list of tokens representing the source code) and AST checkers. Most of the checkers are working on the Abstract Syntax Tree (AST) which is provided by the astroid [23] library. Adding to the reliability of PyLint, it is worth mentioning that it is trusted by many big companies, such as Google [17], which is mostly relying on PyLint for the static analysis of its Python code-base. There is also a number of popular IDEs and frameworks using PyLint for in-time static analysis of the Python code, some of which are: PyCharm, VSCode, Django with PyLint, Eclipse with PyDev etc.

Beside the present static analysis tools, there are several Python tools (mostly in experimental status) which are related to symbolic execution and SMT (Satisfiability Modulo Theories) such as: PyExZ3 [3], PySym [14], PySMT [24], and mini-mc [41], etc. Most of them are using the Z3-solver [15].

During the research and comparison of the AST and symbolic execution methods for Python static analysis in this paper, two tools were used. PyLint, as the currently most reliable representation of evaluating the AST, and, for the symbolic execution part, Z3-solver and mini-mc symbolic model checker, which help to explore the symbolic evaluation of the source code.

One of the more critical common bugs in Python that was not caught by PyLint during the research, nor the Python compiler itself, was the “Closure bug”. Closure in Python is an important concept that allows the function object to remember the values in enclosing scopes even if they are not present in memory. At the same time, it is prone to bugs which as shown in the code example on Listing 1 is very often hardly caught even during runtime.

```
1 def greet(greet_word, name):
2     print(greet_word, name)
3 greeters = list()
4 names = ["Kiki", "Riki", "Joe"]
5 for name in names:
6     greeters.append(lambda x: greet(x,
      name))
7 for greeter in greeters:
8     greeter("Hi ")
```

Listing 1: The closure bug

We may expect this code to print:

```
1 Hi Kiki
2 Hi Riki
3 Hi Joe
```

But instead it prints:

```
1 Hi Joe
2 Hi Joe
3 Hi Joe
```

The closure bug is one of the trickiest issue without actually causing a run time error. In our earlier researches we found that only PyLint is able to catch this problem, reporting a Warning “Cell variable name defined in loop” which is not necessary a clear message for the developers about the specific error they made.

4. Comparison of AST-based and symbolic execution methods

In this section, we compare the power of the AST-based method to the symbolic execution method. We selected two representative tools for the two methods, run tests with them, and analyzed the results. Our goal is not only to show which methods can report more real errors, *true positives*, but also which are better avoiding to report *false positives* – code snippets that are correct but falsely reported as a suspicious code segment.

Most of the present Python tools use the AST method for static analysis of the Python source code. We have selected PyLint [22] as one of the most widely used and powerful static analysis tool for Python, which is using the AST method based on the asteroid [23] library.

For the symbolic execution method, we have chosen mini-mc, an experimental symbolic execution implementation [41], using the Z3's Python interface. The mini-mc tool is implementing the fork-explore-check idea when evaluating symbolic values. The Python VM tries to convert them into boolean values at all branches to intercept the conversion and replace it with a `fork` statement. In practice, mini-mc processes all reachable program paths, forking new processes to evaluate the false branch. It also detects unreachable conditions where the evaluation stops. We selected mini-mc for its simplicity and demonstrative power.

Analyzing the static analysis methods, we noticed that symbolic execution might be better approach for static analyzing of Python considering its dynamically typed characteristics. Therefore, we composed a few examples to see step by step the symbolic evaluation and then compare if PyLint as an AST based static analyzer or mini-mc as a symbolic execution method could catch better the errors during the analysis, and exclude false positives in unreachable paths.

```
1  #!/usr/bin/env python3
2  from mc import *
3  import os
4  import time
5
6  def func(arg):
7    if(1==arg):
8      print("branch11",os.getpid())
9      x=1
10   else:
11     print("branch12",os.getpid())
12     x=0
13   if(1==arg):
14     print("branch21",os.getpid())
15     y=5/x
16   else:
17     time.sleep(3)
18     print("branch22",os.getpid())
19     y=4/(x+1)
20 arg = BitVec("arg",32)
21 func(arg)
```

Listing 2: Usage example of mini-mc.

Listing 2 is the very first Python code example that we used to run a symbolic model checker, and as it is seen, this program should not report an error since both if-conditions (in line 7 and line 13) could be true at the same time and their bodies could be executed without errors. With this example we mostly demonstrate the execution of the mini-mc symbolic model checker. As it is seen on Listing 3 the program was executed in a quasi-parallel way. At every branch statement the program forks a new process, the process id and the logical assumption is written to the output. (The use of time.sleep(3) on line 17 is to emphasize the non-deterministic evaluation order of the branches). When the engine detects unsatisfied condition, that is also printed as `unreachable`.

```
1  [7088] assume (arg == 1)
2  [7090] assume ¬(arg == 1)
3  [7090] unreachable
4  [7088] assume (arg == 1)
5  [7088] assume (arg == 1)
6  branch11 7088
7  branch11 7088
8  branch21 7088
9  [7090] exit
10 [7089] assume ¬(arg == 1)
11 [7089] assume (arg == 1)
12 [7089] unreachable
13 branch12 7089
14 [7089] assume ¬(arg == 1)
15 [7091] assume ¬(arg == 1)
16 branch12 7089
17 branch22 7091
18 [7091] exit
19 [7089] exit
20 [7088] exit
```

Listing 3: Mini-mc result executing Listing 2.

The Python example on Listing 4 should point out the power of symbolic execution over the AST based approaches. The program defines the variable `z` in the true-branch of the first if-condition and uses it in the true-branch of the second if-condition. As the two if-conditions (line 2 and line 5) could not be true at the same time, if `1!=arg` then the first if-condition body would not be executed but the second if-condition body is executed. In this case, however, the program becomes faulty since the variable z was not introduced yet.

```
1 def func(arg):
2   if(1==arg):
3     print("branch11",os.getpid())
4     z=1
5   if(1!=arg):
6     print("branch21",os.getpid())
7     x=z
8 arg = BitVec("arg",32)
9 func(arg)
```

Listing 4: Local variable may be referenced before assignment. PyLint does not report.

PyLint, as an AST based static analyzer, does not execute a full path sensitive analysis, therefore it is unable to recognize that the assignment `x=z` will be executed only in those cases when statement `z=1` is not. In the same time, PyLint detects that the variable `z` is defined in one of the

branches of the if-condition in line 2 and supposes that it will be used only in this case in line 7. At the end, conservatively, do not report error to minimize possible false positives.

When the mini-mc was run the *Unbound error* was detected (Listing 5) showing the benefits of symbolic execution over the AST based methods for Python as a dynamic language.

```
1  [6324] assume (arg == 1)
2  [6324] assume (arg != 1)
3  [6324] unreachable
4  branch11 6324
5  [6324] assume (arg == 1)
6  [6326] assume ¬(arg != 1)
7  branch11 6324
8  [6326] exit
9  [6325] assume ¬(arg == 1)
10 [6327] assume ¬(arg != 1)
11 [6327] unreachable
12 [6327] exit
13 [6325] assume ¬(arg == 1)
14 [6325] assume (arg != 1)
15 branch21 6325
16 Traceback (most recent call last):
17   File "./example4.py", line 17, in <module>
18     func(arg)
19   File "./example4.py", line 14, in func
20     x=z
21 UnboundLocalError: local variable 'z'
       referenced before assignment
22 [6325] exit
23 [6324] exit
```

Listing 5: Mini-mc output on the Listing 4 detecting the undefined variable error.

On Listing 6 we changed the code in order to check the behavior of symbolic execution when instead of concrete values, intervals are used in the if conditions.

```
1 def func(arg):
2   if(1<arg):
3     print("branch11",os.getpid())
4     z=1
5   if(2<arg):
6     print("branch21",os.getpid())
7     x=z
```

Listing 6: Using intervals in conditions.

In order to enter the second if-condition one has to also enter the first if-condition and the unbounded error should not be reported.

```
1  [5243] assume (arg > 1)
2  [5243] assume (arg > 2)
3  [5243] assume (arg > 1)
4  [5245] assume ¬(arg > 2)
5  branch11 5243
6  branch21 5243
7  branch11 5243
8  [5245] exit
9  [5244] assume ¬(arg > 1)
10 [5244] assume (arg > 2)
11 [5244] unreachable
12 [5244] assume ¬(arg > 1)
13 [5246] assume ¬(arg > 2)
14 [5246] exit
15 [5244] exit
16 [5243] exit
```

Listing 7: Execution result of Listing 6.

The results on Listing 7 shows that the symbolic execution was working just fine when we used intervals.

There are certain situations, however, when symbolic/concolic execution may cause unreasonable false positives. This is derived from the nature of concolic execution we discussed in Section 2-D. The evaluation is partially driven by the SAT solver, that is the engine to encounter all execution paths, but the concrete execution of the statements inside the branches is using a concrete value chosen by the constraint.

In the example on Listing 8 we execute a function with an unknown argument `arg`. There is a very small possibility, that `arg` is 42, which could cause `ZeroDivisionError`. PyLint static analyzer does not report such an error, as this would be most likely a false positive.

```
1 def func(arg):
2   if arg == 41:
3     print("branch21",os.getpid())
4   else:
5     print("branch22",os.getpid())
6     z = arg - 42
7     z = 99 / z
```

Listing 8: Concolic execution example

On Listing 9 it could be seen that the symbolic execution inaccurately reports the possibility of `ZeroDivisionError` for the `else` branch.

```
1  [15532] assume (arg == 41)
2  branch21 15532
3  [15533] assume ¬(arg == 41)
4  branch22 15533
5  42
6  Traceback (most recent call last):
7    File "example11.py", line 23, in <module>
8      func(arg)
9    File "example11.py", line 17, in func
10     z = 99 / z
11 ZeroDivisionError: division by zero
12 [15533] exit
13 [15532] exit
```

Listing 9: Symbolic execution with mini-mc produces false positive report on Listing 8.

The reason is, that the concolic execution accidentally takes 42 as the sample value for `arg` when `arg != 41`. This makes the otherwise unlikely situation of dividing by zero unavoidable. Although such unlucky situations may be infrequent in every day development, this false positive could be extremely annoying.

Nevertheless, at the moment concolic execution seems to be the most powerful static analysis method for dynamic languages such as Python, but this example shows that it is far from perfect and that there is room to improve it.

5. Empirical results

For the empirical results on Python static analysis in practice, PyLint was tested on an open source software system CodeChecker. CodeCheker is an analyzer tool, defect database and viewer extension for the Clang Static Analyzer and Clang-Tidy – written mainly in Python 3 as a cooperation between Eötvös Loránd University,

Budapest and Ericsson Hungary Ltd. [9]. Two main reasons why CodeChecker was chosen is: first the expectation that it is high quality software as it is used daily by companies like Google, Mozilla and others, and second, the availability of the developers to confirm our findings. Since at this point of the test, we were specially focused to find directly logical errors and where not interested in the warnings or code styling reports and messages, PyLint was run over the CodeChecker source code with only error reports switched-on.

Several real and particularly interesting error reports were gathered as a result, and in this discussion, we will elaborate the reasons behind them, as well as, summarize the reasoning of their informative value. Moreover, we discuss the possibilities of the detected true positive errors being false positives in other cases (and vice versa), in order to notice if switching a specific error report off and further customizing the static analyzer setup might introduce possible improvements to the power of the static analyzer.

Apart from the empirical results being done for testing and research purposes, we are glad to hear that the developers found this immediately useful enough to start fixing the reported bugs. The pull request to fix most of these issues can be found at [13].

5.1. True positives

There were seven particularly interesting and discussion worthy true positive error reports found during the test run. To begin with, the report "Explicit return in `__init__`" with error code E0101 was confirmed as a true positive by the CodeChecker developers. The `__init__` method in Python is a special one, and is called when allocating memory for a new object. It is not used for creating new objects, only allocating and initializing, so it should not return any value. However, in Python very often it happens that the developers `return None` in the `__init__` method. This does not distinctly cause a run-time error, but the return value is meaningless.

The next one, "Class 'traceback' has no 'print stack' " member with error code `E1101`, is a usual runtime error which occurs when an object is accessed for a non-existent member. Although this was confirmed as a true positive error in the CodeChecker code base, it could be possible, in other cases, and especially if reported by an AST-based static analysis tool, that such an error report is false positive. Due to the dynamically typed nature of Python, this report could refer to object members which are created dynamically, and thus misleadingly predict a run-time error. In effect, this error can be seen reported in our case as well, in the false positive reports on CodeChecker with message Instance of 'BuildAction' has no 'original command' member. There was an object called 'BuildAction' in the code base, which members are dynamically set, and this same report there appears as a false positive.

Afterwards, the following three different true positive reports had the same error code. These are worth to analyze separately due to the fact that each of them being initiated by a slightly different reason. In addition, having the same error code for moderately divergent true positive reports, questions the clarity of the information in the error reports. As mentioned before, the format of the description of the error reports, as well as their clarity, is essential, since it can significantly influence the time and cost of evaluating the reports by the developers, which is presently an unavoidable final step in the complete process of the static analysis. One of these three issues report the message `"Undefined variable 'traceback' "`. It is a very typical error and happens simply because developers forget to import a specific module. In a similar way, it suggests that the `sys` module was not imported. However, other than the obvious case that this can happen, plainly by forgetting to import a specific module, this can also happen, for example, when developers actually do not forget to import the module, but they do it in a bad location in the source code. Namely, it occurs that several lines of imports are put in a try-catch block, including the crucial sys module, too. If for some reason, the try block fails in the middle of execution of the number of import statements, the rest of the statements are being left not executed. Thus, the program continues with the sys module not imported. Of course, this will be a run time error. The third reports of these has the error message "Undefined variable 'msg' ", and it is interesting to observe, because if very simple and naïve. Typing mistakes are really common errors in the everyday industry world. Again, the dynamic nature of Python makes such an error complex enough to get unnoticed by the compiler, and simple typing mistake which might seem trivial to us, leads to a real run-time error, by accessing an undefined variable.

Last true positive error under this discussion is the one with error code E0102 reporting "method is already defined in line 320". Redefining functions, classes and methods in Python is allowed since they are also values which can be stored as variable and you can rebind them and pass around as you would do in C++ when using function pointers. Functions in Python are first-class objects and they are defined dynamically, when we are defining a new function with a function name of an already previously defined function, basically that function name is now just bound to the new function object. Nevertheless, this is still very often confusing for the developers, therefore PyLint reports it as a possible logical error. By

looking deeper into the nature of this problem, and due to many known programming conventions of writing unique function names, which is not a special exception while programming in Python, too, one can easily predict that this error report would usually appear as a true positive and definitely needs a very good reason to decide to switch it off while customizing the static analyzer.

5.2. False positives

In the six confirmed false positives, the PyLint error with code `E0611` which for example occurred with message: "`No name 'spawn' in module 'distutils' `" is worth to begin the discussion with. Due to the improvements in Python 3.3+ over Python 2.7, this error is currently, by many developers, initially marked as a false positive, since the rule is written for a behavior in Python 2.7 which is now improved in Python 3.3+. In the simplest case, creating a simple empty `__init__.py` file in the directory of the module, solves the problem of this error. The `__init__.py` file is needed to be able to treat directories which contain it, as packages. According to the documentation, this prevents directories with common name, to unintentionally hide called modules that occur later on the module search path. In Python 2.7 these `__init__.py` files are a must to be able to import from packages, thus PyLint positively reports an error here. However, Python 3.3+ introduced namespace packages which do not need an ordinary list for their `__path__` attribute and there is no `parent/__init__.py` file. Since there might be multiple parent directories found during an import search, not necessarily physically located next to each other, Python creates namespace package for the top-level parent package whenever it or its sub packages are imported. A detailed description can be found in [29], [30]. Knowing this, we could notice that this particular error could be a possible true positive only if it is run on a Python 2.7 version, otherwise when Python 3.3+ is used, it would be the best to disable this error by customizing the setup of Pyint static analyzer. Another false positive error is the one with error code `E1101` which also occurred in the true positive findings. As mentioned in the discussion in 5-A True positives, due to the dynamic nature of the Python language, such an error report with message "`Instance of 'BuildAction' has no 'original command' member`" can appear as a false positive, since attributes of an object can be set dynamically during run-time. The number of true positive findings on this report is greater than the false positive, therefore we could not simply switch off this error report in order to lower the false positive cases. We experienced that altogether, Pylint reported slightly more true positives than false positives on the test run. This suggests that to find real issues in a well-designed and widely used open source software system, even the current AST based approaches are strong enough to contribute to the quality of the Python software systems.

6. Conclusion

Static analysis methods evaluate software systems without running them, applying various heuristics on the source code to detect possible code vulnerabilities. Current tools provide less support for programming languages with dynamically type system, such as Python. Although there exist some – mainly AST based – tools, their capacity to detect Python-specific errors are yet unsatisfactory. Symbolic execution techniques may open new directions for supporting Python static analysis. The dynamic behavior of the language is better covered by methods, where the change of the program state is emulated. With the help of SAT solvers, we can provide full coverage of the program path (under the resource constraints). We have shown that even the simplest symbolic execution tools can find issues otherwise not detected by AST based tools. Especially concolic execution is one of the most promising method directions.

Symbolic execution-based analyzer tools for Python can use powerful SAT solvers, such as Z3, but currently, they model neither the type of information nor the most important modules of the Python language. This is a serious restriction, and further research should target that area. Nevertheless, the otherwise powerful concolic execution can also cause unwanted false positives. Based on our experiments, we suggest using both an AST based tool and a symbolic execution tool to maximize the true positives and minimize the reported false positives.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018.

[3] Thomas Ball and Jakub Daniel. Deconstructing dynamic symbolic execution. *Proceedings of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering*, (MSR-TR-2015-95), January 2015.

[4] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.

[5] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, January 2001.

[6] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.

[7] Clang SA. Clang Static Analyzer, 2019. https://clang-analyzer.llvm.org/.
[8] Clang Tidy. Clang-Tidy, 2019. https://clang.llvm.org/extra/clang-tidy/ (last accessed: 28-02-2019).
[9] CodeChecker. Codechecker home page, 2019. https://github.com/Ericsson/codechecker (last accessed: 28-08-2019).
[10] Ian Cordasco. Flake8, 2016. http://flake8.pycqa.org/en/latest/.
[11] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.
[12] Timothy Crosley. Frosted, 2014. https://pypi.org/project/frosted/.
[13] Márton Csordás. Fix pylint errors #2448 – pull request on github.com, 2019. https://github.com/Ericsson/codechecker/pull/2448.
[14] Björn I. Dahlgren. Pysym, 2016. https://pythonhosted.org/pysym/.
[15] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963, pages 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
[16] Artem Dergachev. Clang Static Analyzer: A Checker Developer's Guide, 2016. https://github.com/haoNoQ/clang-analyzer-guide (last accessed: 28-08-2019).
[17] Google. Google python style guide, 2018. http://google.github.io/styleguide/pyguide.html.
[18] GrammaTech. CodeSonar, 2019. https://www.grammatech.com/products/\codesonar (last accessed: 28-02-2019).
[19] Hari Hampapuram, Yue Yang, and Manuvir Das. Symbolic path simulation in path-sensitive dataflow analysis. *SIGSOFT Softw. Eng. Notes*, 31(1):52–58, September 2005.
[20] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.
[21] Jukka Lehtosalo. Mypy, 2016. https://mypy.readthedocs.io/en/latest/.
[22] Logilab. Pylint, 2003. http://pylint.pycqa.org/en/latest/.
[23] Logilab. Astroid, 2019. https://astroid.readthedocs.io/en/latest/.
[24] Yuri Malheiros. pysmt, 2016. https://pysmt.readthedocs.io/en/latest/tutorials.html.
[25] Daniel Marjamäki. CppCheck: a tool for static C/C++ code analysis, 2013.
[26] Martin Moene. Search with cppcheck. *Overload Journal*, 120, 2014.
[27] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.
[28] PyCQA. Pyflakes, 2014. https://pypi.org/project/pyflakes/.
[29] Python Software Foundation. Python 2 packages documentation, 2019. https://docs.python.org/2/tutorial/modules.html#packages.
[30] Python Software Foundation. Python 3 packages documentation, 2019. https://docs.python.org/3/tutorial/modules.html#packages.
[31] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.
[32] Johann Rocholl. Pycodestyle, 2006. http://pycodestyle.pycqa.org/en/latest/.
[33] Roguewave. Klocwork, 2019. https://www.roguewave.com/products-services/klocwork (last accessed: 28-02-2019).
[34] Koushik Sen. Concolic testing. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 571–572, New York, NY, USA, 2007. ACM.
[35] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.
[36] Synopsys. Coverity, 2019. https://scan.coverity.com/ (last accessed: 28-02-2019).
[37] Csaba Szabó. Programme of the winter school of project no.2017-1-sk01-ka203-035402: "focusing education on composability, comprehensibility and correctness of working software", 2018. accessed 02-July-2019.
[38] Tiobe. TIOBE programming community index, july 2019, 2019. accessed 02-July-2019.
[39] G. van Rossum. Python tutorial. Technical Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, May 1995.
[40] Š Korečko. Interactive approach to coloured petri nets teaching. Technical Report IK-TR3, Eötvös Loránd University, Faculty of Informatics, Budapest, May 2018.
[41] Xi Wang. A mini symbolic execution engine, 2015. http://kqueue.org/blog/2015/05/26/mini-mc/.
[42] Zhongxing Xu, Ted Kremenek, and Jian Zhang. A memory model for static analysis of C programs. In *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I*, ISoLA'10, pages 535–548, Berlin, Heidelberg, 2010. Springer-Verlag.
[43] Anna Zaks and Jordan Rose. Building a checker in 24 hours, 2012. https://www.youtube.com/watch?v=kdxlsP5QVPw.

Hristina Gulabovska is a MSc Computer Science student in the last semester of her studies at the Eötvös Loránd University (ELTE), Budapest, specializing in Software and Service Architectures.

Zoltán Porkoláb received his doctoral degree in Computer Science from the Eötvös Loránd University (ELTE), Budapest in 2004. He is an Associate Professor of the Department of Programming Languages and Compilers at the Faculty of Informatics, ELTE, Budapest, Hungary. At the same time, he holds Principal C++ Developer position at Ericsson Hungary Ltd.

Advanced Code Comprehension using Version Control Information

Tibor Brunner; Zoltán Porkoláb

Abstract: ***Version control systems were originally designed to handle the development process of a source code and synchronize team work by resolving conflicts. While this purpose remains the primary target, they proved to be useful for other purposes including the help of code comprehension. Advanced code comprehension process requires utilization of the full knowledge portfolio of the software system. In this paper we investigate how a version control information of the project can be utilized for extending our apprehension of large legacy systems providing a better understanding of the software under examination. We show that some of the hidden structural connections between the elements of the program can be revealed most easily by the development history of the system. An industrial level implementation of the method using git version control information has been implemented as an open source extension of the CodeCompass software comprehension framework.***

Index Terms: ***code comprehension, version control, git, software technology***

1. Introduction

It is a well-known fact, that the largest cost factor of a software product for its whole lifetime is the maintenance cost. One of the reasons is that, prior to any maintenance activity – new feature development, bug fixing, etc., – programmers first have to locate the place where the change applies, then, have to understand the actual code, and, finally, have to explore the connections to other parts of the software to decide *how* to interact in order to avoid a regression. All these activities require an adequate understanding of the code in question and of its environment. Most of these activities are currently impossible to automate, therefore the developers should spend their expensive time to carry out these actions.

Therefore, it is not a surprise that code comprehension is a key factor of modern software development, exhaustively researched by both the industry and academy. Various scientific and industrial papers have been published on the topic in conferences, e.g. in the series of International Conference of Program Comprehension, and in the Intellectual outputs No. O1 and O2 of the Erasmus+ Key Action 2 (Strategic partnership for higher education) project No.2017-1-SK01-KA203-035402: “Focusing Education on Composability, Comprehensibility and Correctness of Working Software” [16], [31] among others.

Most of the comprehension approaches are based on the source code. That is a logical approach as the actual software might have already diverged from the original specification and the documentation may also be outdated. Therefore, typical comprehension tools analyze the source code, support fast navigation, feature location, and reveal the internal structure of the software. However, not all the internal connections within the system can be detected by analysing the source. Virtual function calls on polymorphic objects, pointers, references, closures are among the program constructs where static analysis has limitations.

Code comprehension may not be restricted to existing code bases. Important architectural information can be gained from the build system, like relations between libraries, binaries and source files [32], [33]. Even more interesting structural connections can be revealed from the history of the project development. We can identify which files were added or changed together, how these changes are related to certain commit messages and which lines were added/removed/changed frequently.

CodeCompass is an open source code comprehension framework [29] developed by Ericsson and Eötvös Loránd University, Budapest, to help the code comprehension process of large legacy systems. The tool is based on the LLVM/Clang compiler infrastructure [5], [14], and has been designed to be extremely scalable, seamlessly working with many million lines of code. Fast search options help locating the requested feature by text search. Once the feature has been located, precise information on language elements for variables, inheritance and aggregation relationships of types, and call points of functions are provided by the LLVM/Clang infrastructure. Easy navigation possibilities and a wide range of visualizations extend far more than the usual class and function call diagrams and help the user in more complete comprehension. To make the compre-

This work was supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002)

T. Brunner (contact person) is a PhD student at the Faculty of Programming Languages and Compilers, Eötvös Loránd University, Hungary (e-mail: bruntib@caesar.elte.hu).

Z. Porkoláb is an Associate Professor at the Faculty of Programming Languages and Compilers, Eötvös Loránd University, Hungary (e-mail: gsd@caesar.elte.hu).

hension more extensive, CodeCompass utilizes a full portfolio of available information including build commands but also utilizes version control information, if available; git commit and branching history, blame view are also visualized.

In this paper we investigate the role of the version control information for code comprehension purposes. We overview the main categories of the existing comprehension software using specific tools as example in Section 2. In that section we also review current research directions regarding the version control systems. Section 3 describes the architecture of CodeCompass in the context of how we collect and use version control information. In Section 4 we present why it is important to include version control information in a code comprehension tool. In Section 5 we show CodeCompass support for the version control information to reveal hidden connections between otherwise unrelated code segments. Section 6 discusses the most common use cases where version control could be used for code comprehension. Our paper concludes in Section 7.

2. Related Work

Code comprehension became a hot research topic recently, with dedicated user communities and, proprietary and open source tools.

2.1. Code Comprehension Tools

On the software market, there are several tools which aim some kind of source code comprehension. Some of them uses static analysis, others examine also the dynamic behavior of the parsed program. These tools can be divided into different archetypes based on their architectures and their main principles. On the one hand there are tools having server-client architecture. Generally, these tools parse the project and store all necessary information in a database. Clients (usually web-based) are served from the database. These tools can be integrated into the workflow as nightly CI runs. This way the developers can always browse and analyze the whole, large, legacy codebase. Also, there are client-heavy applications where smaller part of the code base is parsed. This is the use case for most of the IDE based editors where the frequent modification of the source requires quick update of the database with analyzed results. In this section we present the most popular tools used in industrial environment from each category.

Woboq Code Browser [6] is a web-based code browser for C and C++ languages. This tool has extensive features which aim for fast browsing of a software project. The user can quickly find the files and named entities by a search field which provides code completion for easy usability. The navigation in the code base is enabled through a web page consisting of static HTML files. These files are generated during a parsing process. The advantage of this approach is that the web client will be fast since no “on the fly” computation is needed on the server side while browsing. Also, hovering the mouse on a specific function, class, variable, macro, etc. can show the properties of that element. For example, in case of functions one can see its signature, place of its definition, and place of usages. For classes, one can check the size of its objects, the class layout, and offset of its members and the inheritance diagram. For variables, one can inspect their type and locations where they are written or read.

OpenGrok [11] is a fast source code search and cross reference engine. Opposed to Woboq, this tool doesn’t perform deep language analysis, therefore it is not able to provide semantic information about the particular entities. In addition to text search there is possibility to find symbols or definitions separately. The lack of semantic analysis allows Ctags to support several (41) programming languages. Also, an advantage of this approach is that it is possible to incrementally update index database. OpenGrok also gives opportunity to gather information from version control systems like Mercurial, SVN, Git, etc. It has the ability to search not only in the content of source files but in their history as well. Since most of these version control systems (VCS) provide search functionalities in the project history (including commit messages and source files), OpenGrok can forward these queries to the given VCS. However, there are no extra visualizations in order to display the “blame view” so the developer could understand what other relevant changes happened in other files in the same commit. The branches of the history are invisible too. CodeCompass intends to support these use-cases.

Understand [13] is not only a code browsing tool, but also a complete IDE. Its great advantage is that the source code can be edited and the changes of the analysis can be seen immediately. Besides code browsing functions already mentioned for previous tools, Understand provides a lot of metrics and reports. Some of these are the lines of code (total/average/maximum globally or per class), number of coupled/base/derived classes, lack of cohesion [19], McCabe complexity [24] and many others. *Treemap* is a common representation method for all metrics. It is a nested rectangular view where nesting represents the hierarchy of elements, and the color and size dimensions represent the metric chosen by the user. For large code bases, the inspection of the architecture is necessary. Visual representation is one of the most helpful ways of displaying such structures. Understand can show dependency diagrams based on various relations such as function call hierarchy, class inheritance, file dependency,

file inclusion/import. Users can also create their custom diagram type via the API provided by the tool.

CodeSurfer [8] is similar to Understand in the sense that it is also a thick client, static analysis application. Its target is understanding C/C++ or x86 machine code projects. CodeSurfer accomplishes deep language analysis which provides detailed information about the software behavior. For example, it implements pointer analysis to check which pointers may point to a given variable, lists the statements which depend on a selected statement by impact analysis, and uses dataflow analysis to pinpoint where a variable was assigned its value, etc.

Development tools. The aforementioned tools are mainly designed for code comprehension. Another application area of static analysis is writing the code itself. This is a very different way of working in many aspects, which requires a slightly different tool set. Maybe the most widespread IDEs are *NetBeans* [2] and *Eclipse* [9] primarily for Java projects, and *QtCreator* [12] mainly for C++ projects. The recent open source tools tend to be pluginable so their functions can easily be extended according to special needs and domain specific tasks. The greatest benefit of these tools is the ability of incremental parsing, which means the real-time re-analysis of small deviations in the source code. The *Visual Studio* [15] IDE has a rich interface for code comprehension features, like go to definitions and all references among others.

2.2. Version Control Information Usage

Version control information is used for various software research areas. Most frequently, in the center of these researches is the connection between commit actions and software quality, and the cost of maintenance. Naturally, this is used as the prediction of distribution of software bugs. In [26], the authors developed a regression model that accurately predicts the likelihood of post-release defects for new entities. Similarly, in his PhD thesis [17] the author describes the connection between code maintenance activities as it is reflected by version control information and the deterioration of the code quality. In a related paper [18] the authors show that a connection between version control operations and maintainability really exists, in spite of the fact that the data is coming from different sources.

Apart from software quality, other researches target the developer's team. The authors of [22] utilize version control information for mining and visualizing networks of software developers. They detect similarities among developers based on common file changes, and construct the network of collaborating developers. The authors show that this approach performs well in revealing the structure of development teams and improving the modularity in visualizations of developer networks.

Unfortunately, information retrieved from version control systems has its limitations. As an example, attempts to predict code quality or developer efficiency cannot be achieved according to a research described in [25].

A frequent problem with information mining from version control systems is that they store only atomic information. In [23], the authors suggest a set of heuristics for grouping change-sets files that frequently change together. The results show that the approach is able to find sequences of changed-files.

Version control information can be used not only for extracting data from the source code for code comprehension purposes, but it is also a frequent research target for analyzing comments – either structured, or natural language text – in order to get additional information about the system [30].

As we have seen, the information retrieved from the version control systems is used for various other purposes than the original intention for the original source control. However, we are not aware of any earlier published research that is aiming the code comprehension problem using version control information.

3. The CodeCompass Architecture

CodeCompass provides a read-only, searchable and navigable snapshot of the source code, rendered in both textual and graphical formats. CodeCompass is built with a traditional server-client architecture as depicted in Figure 1. The server application provides a Thrift [3] interface to clients over HTTP transport. The primary client that comes pre-packaged with the tool is a web browser based single-page HTML application written in HTML and JavaScript.

Since the interface is specified in the Thrift interface definition language, additional client applications (such as a command line client or an IDE plugin) can be easily written in more than 15 other languages supported by Thrift (including C/C++, Java, Python etc.). An experimental Eclipse plugin is already implemented.

A parsed snapshot of the source code is called a *workspace*. A workspace is physically stored as a relational database instance and additional files created during the *parsing process*. The parsing process consists of running different *parser plugins* on the source code. The most important parser plugins are:

A **search parser** iterates recursively over all files in the source folder and uses Apache Lucene [1] to collect all words from the source code. These words are stored in a search index, with their exact location (file and position).

The **C/C++ parser** iterates over a JSON compilation database containing build actions, using

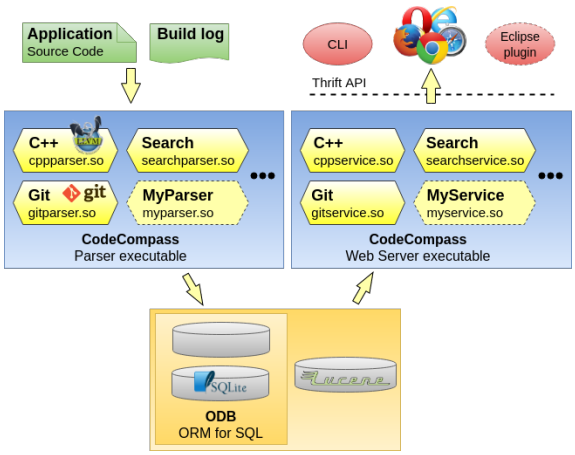


Fig. 1: CodeCompass architecture

the LLVM/Clang parser [5] and stores the position and type information of specific AST nodes in the database. This database will be used by the *C/C++ language service* to answer Thrift calls regarding C/C++ source code.

Among other parser plugins the **Git parser** reads the version control information from the source tree (in the `.git` directories) and stores it into the project database.

CodeCompass has an extensible architecture, so new parser plugins can be written easily in C/C++ language. Parser plugins can be added to the system as shared objects.

On the **webserver**, Thrift calls are served by so-called *service plugins*. A service plugin implements one or more Thrift services and serves client requests based on information stored in a workspace. A Thrift service is a (remotely callable) collection of methods and type definitions. All Thrift services have one implementation with the exception of the *language service*, which is implemented for C/C++, Java and Python. The *language service* is distinct in the sense that it provides a basic code navigation functionality for the languages it is implemented for. To put it simply, if this interface is implemented for a language, a user will be able to click and query information about symbols in the source code view of a file written in the given language.

The **Web-based user interface** is organized into a static *top area*, extensible *accordion modules* on the left and also extensible *center modules* on middle-right.

The source code and different visualizations are shown in the center, while navigation trees and lists, such as file tree, search results, list of static analysis (CodeChecker) bugs, browsing history, code metrics, and version control navigation is shown on the left. New center modules and accordion panels can be added by developers. The top area shows the search toolbar, the currently opened file, the workspace selector, simple navigation history (breadcrumbs) and a generic menu for user guides.

The version control related functionality is available from both the center modules, selecting the *Team* menu item on any code parts, and from the *revision control* module from the *accordion* part.

4. Importance of Version Control in Code Comprehension

The original goal of version control systems was to handle the development process of the source code and to synchronize the team work by resolving conflicts in case of colliding code modifications. However, many times the source code management is joined with some issue tracker in order to provide a platform for reporting bugs or collecting feature requests. GitHub [20], GitLab [21] and Bitbucket [28] are the most popular code storage and issue tracker platforms.

The unit of a code modification in Git is *commit*. Among others a Git commit consists of the set of changed lines, the committer, and the date of modification. A hash value combined from these identifies the commit itself. Each commit has one or more parents to which the change is related. This relationship between commits defines the *commit history*. When a new feature is being introduced then the development of this feature happens on a different branch, i.e. a commit has another child node. This way the modifications of differences don't interfere until the feature is ready. As soon as it is complete, the branch can be *merged* to the main software branch resulting two parent commits for the merge node.

Git supports multiple development workflows. Guidelines are available that describe how to organize the joint work among team members. One way of working is keeping the commit history in one line. In this formation every team member is pushing commits to the single master branch. Of course, some review sessions precede this action. Gerrit [10] is such a review handler tool that supports this workflow. When a new feature or bug fix has been implemented, it is pushed as one commit to the top of Git history, so the steps of the new feature's solution cannot be broken down to smaller commits. Another workflow is creating a new branch for a new feature or bug fix, implementing the solution in several commits and sending a merge request to the master branch through the version control system. The merge command will join the feature branch to the master with a special *merge commit* which thus has two parents.

In either configuration there is a way for grouping a set of modifications belonging together. From the code comprehension point of view this grouping has an important role, since this way

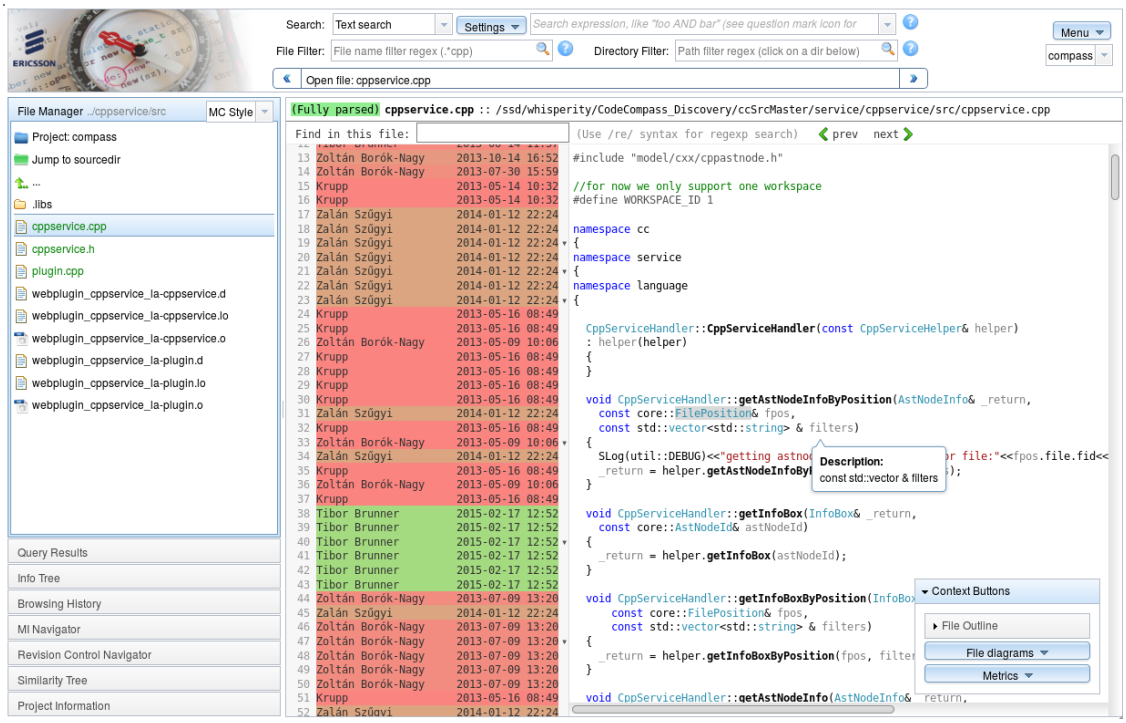


Fig. 2: Git blame view

we can direct our attention on specific code fragments which are enough to consider in their own. Highlighting these parts, the huge code base is narrowing down to a few lines what makes comprehension much easier.

In a well-organized project it should be easy to orientate. Some languages like Java even enforce the organization of modules in an equivalent directory structure. Or in C/C++, there is a common practice to separate the interface and the implementation of a module. The interface is located in a header file conventionally under an `include` directory and the implementation goes to the corresponding `src` folder with the same name except for the file extension (.h vs. .c or .cpp). However, when it comes to a feature development, the introduction of a new feature may touch the files of multiple modules located at distant parts of the code base. In the commit history of CodeCompass project we found that there were 226 non-merge commits (i.e. commits with one parent) which contain code modification or addition, and in 104 of these there was a common modification of an interface and its implementation file together. Besides these in the 226 commits, on average more than two files were edited which are not a common modification of an interface and an implementation.

The presentation of the files modified together is a helpful visualization in order to help users to see which other files should be inspected together while comprehending a code part. In the next section we discuss what kind of visualizations CodeCompass provides regarding Git version control system to support this process.

5. Version Control Support in CodeCompass

CodeCompass supports the code comprehension via various visualizations to present the different views of git information related to the project. As a starting point, one can initiate the **blame view** on any source code. Git blame view shows line-by-line the latest changes (commits) to a given source file as seen on Figure 2. The background color of the committer also holds information: the commit that happened recently are colored lighter green, while older changes are darker red. This view is excellent to review why certain lines were added to a source file.

Clicking on the committers' name of the blame view CodeCompass brings us to the **commit information**. This contains the exact date and the message of the comment. Here we still can inspect the committed code.

CodeCompass can also show Git commits in a filterable list ordered by the time of the commit. This search facility can be used to list changes made by a person or to filter commits by relevant words in the commit message.

Many times, when we are reading the source code and find a fragment which is interesting in some ways or seems suspicious, we would also like to find what other parts of the current file have been modified at the same time. These colors are thus visual aid for determining which modifications

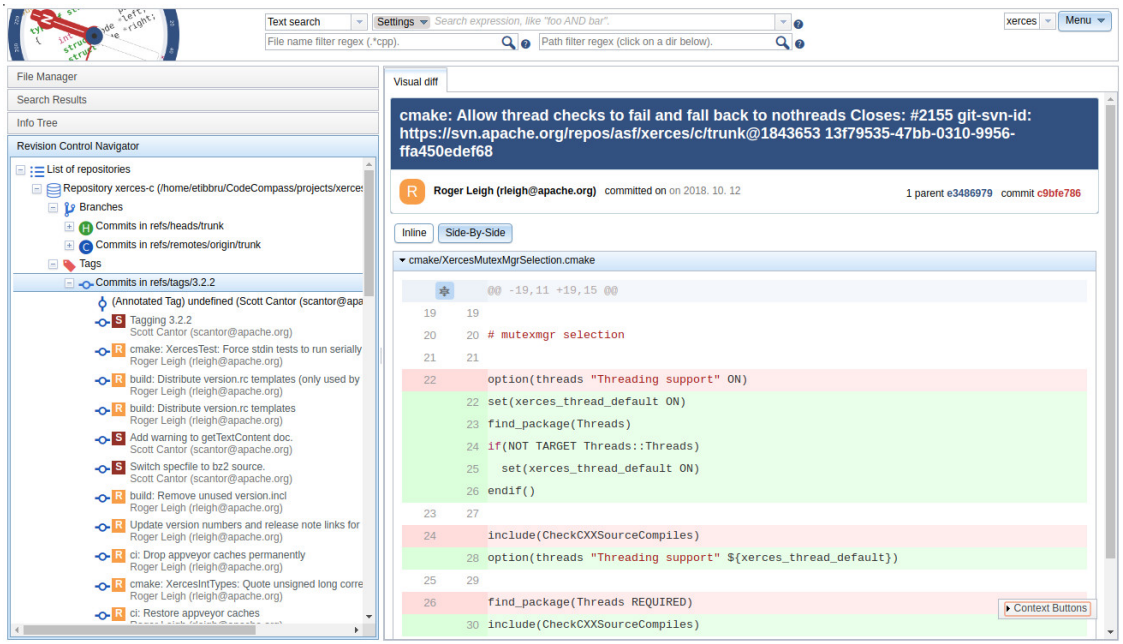


Fig. 3: Git branch view

belong to the introduction of the same new feature or bug fix.

Usually it is a project level decision whether the explanatory comments about the reasons of a modification should be incorporated in the source code or should be described in the commit message. The advantage of writing these comments in the source code is that this way these comments will be version controlled and also inseparable from the code. However, some information belongs to the commit message of the modification (e.g. the issue which this commit solves, some links to external pages or earlier commits, etc.). In CodeCompass we would like to present this information to the user too, immediately at the currently displayed file and line.

CodeCompass implements **branch view** that presents which commits have been developed on a separate branch and thus belong together. On Figure 3 we see a typical branch view.

In case of a software project, the evolution of the program may also carry useful information. The organization of the development process is also up to the project members. One of the most common structures is to maintain a *master* branch which always contains the latest version of the project. When a new feature is created or a bug fix is being introduced, then these are developed on a separate, so called *feature branch*. This branch contains one or more commits which make up the change together. When the introduced feature is stable on the feature branch, then the changes are merged to the master. The bigger a new feature is, the more suitable it is to separate it on consecutive commits.

All of the visualizations are based on a similar graphical appearance of existing revision control tools, to minimize the cognitive effort for the developers when using revision control related information in CodeCompass.

Such use of revision control information of large legacy projects can reveal hidden connections in large software systems and help the complete comprehension of these projects.

6. Important Use Cases

In this section we show use cases where version control information helps to understand the connection between different parts of the system, thus provides essential information for both comprehending and maintaining it.

6.1. Configuration

Let us suppose that we are maintaining a foundation library, like a library for logging. It is common that such a library can be configured without a modification of the code, e.g. via environment variables or configuration files. Such libraries are evolving by time and new features are added to the existing ones, also, they may be target of bug fixes and other maintenance works.

How can we add a new feature (or feature parameters) to the existing system? First, we have to modify the structure of the configuration file. Depending on the technical implementation, this requires the (de-)serialization of the configuration file and/or the schema description of it. Also, the new features should be implemented in the heart of the library.

In any well-structured system, these three actions should be executed in separate parts of the code. The reader of the configuration file is a well-defined sub-library called either in the startup phase of the library, or during static initialization. The schema description is usually a different file, e.g. in an XSD format, also physically separated from the rest of the system. Finally, the actual code to execute the feature is placed somewhere inside the logical structure of the library.

It is easy to see, that there exist neither direct relations between the files (e.g. they are not in some common subdirectory), nor there are implicit language connections, like function calls, or commonly used global variables to bind these participants. The developer has no other chance to detect these connections than reveal the version control information to connect them: they were created, checked-in and maintained together during the lifetime of the system.

6.2. Plug-ins

Large systems linking against fixed libraries (either statically linked or using shared objects/DLLs) are sometimes proved to be too rigid. It is more flexible to apply advanced component-oriented approaches, like plug-ins [27]. With plug-ins it is possible to apply new code to an existing system without any additional compiling/linking action or even other configuration settings. Applications of plug-ins include:

- enabling third-party developers to create features to extend static applications, like audio or video decoders, graphical components, etc.
- adding features to existing large systems.
- reducing the size of the system elimination unused features from linking statically to the program.
- separating software components with incompatible licenses.

The plug-inable infrastructure, by its nature, avoids any direct connection between the plug-ins and the components working with them, otherwise the linking phase could not be avoided. However, this means that no explicit information is held in the system about these implicit dependencies. When a new plug-in is applied or an existing one has been modified it could be challenging to detect the components that require interest.

In those cases, the version control information is the main source for the maintainer. As plug-ins and their client code are introduced together into the system, the maintainer can detect the corresponding components.

6.3. Databases/Network Connections

Hidden connections between software components in large systems can be manifested in persistent data. One component of the system may write a record to the database and other(s) may retrieve this information without introducing any explicit connection between these modules. Similar situation happens when data is sent over a network connection. Let's recognize, that even the use of some common data type is not necessarily occurring between the communicating entities.

The reader and writer side of the communicating partners can use differently *named* data types with the same structure. It is also not uncommon, that – for efficiency reasons – even the structure of the reader and writer buffers are different. In such situations the version control information could be a useful source to reveal the connections.

6.4. A Real World Example

Xerces-C++ is a popular open source library for parsing, validating, serializing and manipulating XML documents [4] written in a portable subset of C++ and makes it easy to give applications the ability to read and write XML data.

The library is prepared for annotation support back in 2003. The modification added a new struct `PerfMapElem` to the header file `ElemStack.hpp`. If one start to investigate what the purpose of this feature addition is and how it has been implemented it is a natural way to search all the usages of this type. However, `PerfMapElem` is used only in 3 other files. The usages include defining template parameter of a vector-like container as well as declaring (pointer(s) to) individual objects.

However, it is less obvious that, among others, this modification removed the `fElemStack` field from the `DGXMLScanner` class and replaced by a heap allocated vector `ValueVectorOf<PrefMapElem*>`. This information can be retrieved easily from the version control information.

Locating the definition of the `PerfMapElem` in `ElemStack.hpp`, we can use the *blame view* (see Figure 2) to find the corresponding commit `f2cf1160` where this definition has been introduced. Clicking on the left bar of the view, we are automatically navigating to the *change view* where all the changes included by this commit is represented. This view also reveals that the new feature modified 12 files instead of the originally supposed 4. It would be otherwise a very hard investigation to detect these connections without the version control information.

7. Conclusion

Code comprehension is an important research area to support better understanding of large industrial software systems to reduce maintenance costs. Tools supporting comprehension are mainly based on the static analysis of the source code. However, software systems may contain hidden

relationships between their components. Connections between configuration files and their application in the code, and similar are hard to detect based on the source. At the same time, those connections are likely reflected by the software development process, which can be retrieved from the version control information.

CodeCompass is an open source code comprehension framework which is intended to collect the whole information portfolio of the system under investigation. This includes not only the internal structure revealed from the source code, but also additional information, including the git version control information.

The blame view shows the last committers line by line for the source lines visually expressing the "age" of the code. From here, the developer easily can navigate to the commit information, to check the commit message and the other files affected by the commit. One can also compare the code of different comments. Finally, we can inspect the development of the project by the traditional branch view. The applied visualizations inherit the graphical interface of the usual version control tools to make them familiar for the developers.

All these possibilities make the comprehension more complete and help to increase the code quality in the further development process. The full implementation is available as an open source project at [7].

REFERENCES

[1] Apache Lucene. https://lucene.apache.org/core/.
[2] Apache NetBeans. https://netbeans.org.
[3] Apache Thrift. https://Thrift.apache.org.
[4] Apache Xerces-C validating XML parser. https://github.com/apache/xerces-c.
[5] Clang: a C language family frontend for LLVM. https://clang.llvm.org/.
[6] Code Browser by Woboq for C and C++. https://woboq.com/codebrowser.html.
[7] CodeCompass website. https://github.com/Ericsson/CodeCompass.
[8] CodeSurfer. https://www.grammatech.com/products/codesurfer.
[9] Eclipse. https://www.eclipse.org/ide/.
[10] Gerrit Code review home page. https://www.gerritcodereview.com/.
[11] OpenGrok. https://opengrok.github.io/OpenGrok.
[12] Qt Creator. https://www.qt.io/.
[13] SciTools: Understand. https://scitools.com.
[14] The LLVM Compiler Infrastructure. https://llvm.org/.
[15] Visual Studio. https://visualstudio.microsoft.com.
[16] Tibor Brunner. Codecompass: an extensible code comprehhension framework. Technical Report IK-TR1, Eötvös Loránd University, Faculty of Informatics, Budapest, May 2018.
[17] Csaba Faragó. Maintainability of source code and its connection to version control history metrics, phd thesis. Technical report, Department of Software Engineering, University of Szeged, Szeged, May 2016.
[18] Csaba Faragó, Péter Hegedűs, Ádám Zoltán Végh, and Rudolf Ferenc. Connection between version control operations and quality change of the source code. *Acta Cybernetica*, 21(4):585–607, 2014.
[19] Brian Henderson-Sellers. *Object-oriented Metrics: Measures of Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
[20] GitHub Inc. home page. https://github.com.
[21] GitLab Inc. home page. https://gitlab.com.
[22] Andrejs Jermakovics, Alberto Sillitti, and Giancarlo Succi. Mining and visualizing developer networks from version control systems. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, page 24–31, New York, NY, USA, 2011. Association for Computing Machinery.
[23] Huzefa Kagdi, Shehnaaz Yusuf, and Jonathan I. Maletic. Mining sequences of changed-files from version histories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, page 47–53, New York, NY, USA, 2006. Association for Computing Machinery.
[24] McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.
[25] Keir Mierle, Kevin Laven, Sam Roweis, and Greg Wilson. Mining student cvs repositories for performance indicators. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, page 1–5, New York, NY, USA, 2005. Association for Computing Machinery.
[26] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 452–461, New York, NY, USA, 2006. Association for Computing Machinery.
[27] Oscar Nierstrasz, Simon J. Gibbs, and Dennis Tsichritzis. Component-oriented software development. *Commun. ACM*, 35(9):160–165, 1992.
[28] Atlassian Corporation Plc. Bitbucket home page. https://bitbucket.org.
[29] Zoltán Porkoláb, Tibor Brunner, Dániel Krupp, and Márton Csordás. Codecompass: An open software comprehension framework for industrial usage. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 361–369, New York, NY, USA, 2018. ACM.
[30] Y. Shinyama, Y. Arahori, and K. Gondow. Analyzing code comments to boost program comprehension. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 325–334, 2018.
[31] Csaba Szabó. Programme of the winter school of project no.2017-1-sk01-ka203-035402: "focusing education on composability, comprehensibility and correctness of working software", 2018. accessed 02-July-2019.
[32] Richárd Szalay, Zoltán Porkoláb, and Dániel Krupp. Measuring mangled name ambiguity in large c/c++ projects. In *Zoran Budimac (Ed), Proceedings of the Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications. Belgade, Serbia 2017.*, 2017.
[33] Richárd Szalay, Zoltán Porkoláb, and Dániel Krupp. Towards better symbol resolution for C/C++ programs: A cluster-based solution. In *IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 101–110. IEEE, 2017.

Tibor Brunner is working on his PhD at the Eötvös Loránd University (ELTE), Budapest, Hungary. He is an expert of code comprehension and static analysis. He is also teaching C and C++ programming languages for BSc students.

Zoltán Porkoláb received his doctoral degree in Computer Science from the Eötvös Loránd University (ELTE), Budapest, Hungary in 2004. He is an Associate Professor at ELTE, and at the same time, he holds Principal C++ Developer position at Ericsson Hungary Ltd.

The Automated Creation of Logical Constructions from Natural Language Sentences

Bilanová, Zuzana; Štancel, Martin

Abstract: ***This paper describes the theoretical design and implementation of a prototype semantic machine of transparent intensional logic. Procedural semantics of this logic causes that it can be used in the field of logical analysis of natural language. This higher-order logic works with partial functions of the intensional typed λ-calculus what allows us to remove the semantic ambiguities of natural language. The analysis of sentences in transparent intentional logic is performed in three steps - type analysis, construction synthesis, and type control. The semantic machine described in this paper makes it possible to analyze sentences in natural language in the first two of the above steps, for the mentioned logical system. Since transparent intensional logic examines the meaning of sentences in natural language, there was no need to focus on the syntactic level of analysis. The parsing is performed using an external analyzer, which has been carefully selected after comparison with competing analyzers. At the end of the article, the possibilities of the presented solution are compared with the already existing semantic machines processing the meaning of natural language sentences into intensional constructions.***

Index Terms: ***intensional construction, semantic machine, syntactic analyzer***

1. INTRODUCTION

PAVEL Tichý created **transparent intensional logic** (TIL) in 1961 [20]. TIL is based on an extended variant of the object-oriented λ-calculus builds on the ramified hierarchy of types. Tichý's object base (defined to analyze natural language expressions) consists of a set of individuals ι, a set of truth values o, a set of time points τ, and a set of possible worlds ω [19]. The intensions are objects of type $((\alpha\tau)\omega)$, which means function from some possible world in time point for any type α.

The meaning of natural language expression is represented by a construction. Construction is an abstract and algorithmically structured procedure. There are four basic kinds of **constructions** [8]:

- variable '*x*' – *v*-constructs a construction (based on its valuation *v*),
- trivialization '^0*X*' – constructs X without any change,
- closure '$\lambda x X$' – constructs a function,
- composition '[*XX*]' – constructs the application of a function to its arguments.

The logical system TIL is unconventional in that it is not completely formally specified by syntax, semantics and proof system. Montague intensional logic (MIL) [18] is the competitive logic for TIL. MIL has several drawbacks that TIL had overcome, more at [2]. TIL's extraordinary expressive power makes it a suitable basis for the implementation of a semantic machine.

2. SYNTACTIC ANALYSIS

To successfully implement a semantic machine, it is necessary to first perform a syntactic analysis of sentences [10] and then examine the meaning of a sentence using semantic analysis [9]. TIL principles are applied at the semantic level, so there is no need to implement own parser. An existing tool used for parsing was selected after comparing several efficient parsers.

Natural Language Toolkit (NLTK) [3] contains a set of libraries that allow text classification, tokenization, and grammatical analysis. An active community of developers makes it an up-to-date tool with significant usability potential. The disadvantage of NLTK is that it is intended more for beginners in the field of semantic analysis and is also not suitable for processing large amounts of data. **TextBlob** [16] can be considered an extension of NLTK, but its use is even more intuitive, due to its simple interface. Like NLTK, it is slow and unsuitable for larger projects. **Apache OpenNLP** [15] can be used not only for the standard spectrum of text data analysis but also for creating text corpora for generators and conversational interfaces. Unfortunately, it is not a modern solution and therefore does not have the necessary

Manuscript received May, 2020.

This work was partially supported by the Faculty of Electrical Engineering and Informatics at the Technical University of Košice under contract No. FEI-2020-70: "Behavioral model of component systems based on coalgebras and linear logic" and by the project KEGA under contract No. 011TUKE-4/2020: "A development of the new semantic technologies in educating of young IT experts".

Z. Bilanová (contact person) is an assistant professor at the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia (e-mail: zuzana.bilanova@tuke.sk).

M. Štancel works at the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia (e-mail: martin.stancel@tuke.sk).

development support. **SpaCy** [11] can be seen as a successor to NLTK. Unlike NLTK, it is also suitable for more complex business applications. SpaCy is not only the fastest parser but also a complex tool that has only one API for all types of parsing. The disadvantage of SpaCy is that it is primarily intended for text analysis in the field of market research. A Python-based library called **Textacy** [6] is based on SpaCy and is as powerful as its predecessor. However, Textacy focuses on additional tasks that are performed before and after parsing (processing of various types of documents, downloadable datasets with data intended for further processing, calculations for textual statistics, etc.).

The library **Stanford Core NLP** (SCN) [17] was used to create our semantic machine. SCN is a multi-purpose multiplatform parser, whose biggest advantage is its scalability. Several of its components can be integrated with NLTK, but unlike it, it can process large amounts of data very quickly (after SpaCy, this is the fastest of the mentioned solutions). In addition to text processing and generation, it also allows us to work with conversational interfaces and can extract an extraordinary amount of information from the analyzed text. As part of syntactic analysis, it provides data obtained from grammar analysis, statistics, and deep learning. The SCN was created by an elite research institution and is therefore not intended for commercial purposes but as a basis for further research and experimentation.

2.1. Stanford CoreNLP

Based on the previous analysis, there are several reasons why the SCN was selected as an external tool generating input to semantic machine TIL: a freely available solution designed for academic purposes, support for a large number of natural languages, extensive project documentation, active community of developers creating regular updates, high flexibility and scalability of the system, integration of various types of text analysis, visualization analysis outputs, high analysis speed even with a large amount of input data.

The use of SCN is demonstrated in Fig.1, where the output is in json format.

2.2. SCN dependencies

Since it was decided to use SCN, it is necessary to understand data in its output. For the implementation of the semantic machine, it is important to focus on *tokens* and *basicDependencies*. There are some selected dependencies explained [5] (later used in chaper 3-A):

- conj - conjunction,
- cop - copula,
- nsubj - nominal subject,
- det - determiner,


```
"tokens":[{
        "after":" ",
        "before":"",
        "characterOffsetBegin":0,
        "characterOffsetEnd":4,
        "index":1,
        "lemma":"John",
        "originalText":"John",
        "pos":"NNP",
        "word":"John"},
      { "after":" ",
        "before":" ",
        "characterOffsetBegin":5,
        "characterOffsetEnd":7,
        "index":2,
        "lemma":"be",
        "originalText":"is",
        "pos":"VBZ",
        "word":"is"},
      { "after":" ",
        "before":" ",
        "characterOffsetBegin":8,
        "characterOffsetEnd":10,
        "index":3,
        "lemma":"a",
        "originalText":"an",
        "pos":"DT",
        "word":"an"},
      { "after":" ",
        "before":" ",
        "characterOffsetBegin":11,
        "characterOffsetEnd":19,
        "index":4,
        "lemma":"american",
        "originalText":"American",
        "pos":"JJ",
        "word":"American"},
      { "after":"",
        "before":" ",
        "characterOffsetBegin":20,
        "characterOffsetEnd":26,
        "index":5,
        "lemma":"singer",
        "originalText":"singer",
        "pos":"NN",
        "word":"singer"},
      { "after":"",
        "before":"",
        "characterOffsetBegin":26,
        "characterOffsetEnd":27,
        "index":6,
        "lemma":".",
        "originalText":".",
        "pos":".",
        "word":"."
      }
    ]
```

```
"basicDependencies":[
      {
        "dep":"ROOT",
        "dependent":5,
        "dependentGloss":"singer",
        "governor":0,
        "governorGloss":"ROOT"
      },
      {
        "dep":"nsubj",
        "dependent":1,
        "dependentGloss":"John",
        "governor":5,
        "governorGloss":"singer"
      },
      {
        "dep":"cop",
        "dependent":2,
        "dependentGloss":"is",
        "governor":5,
        "governorGloss":"singer"
      },
      {
        "dep":"det",
        "dependent":3,
        "dependentGloss":"an",
        "governor":5,
        "governorGloss":"singer"
      },
      {
        "dep":"amod",
        "dependent":4,
        "dependentGloss":"American",
        "governor":5,
        "governorGloss":"singer"
      },
      {
        "dep":"punct",
        "dependent":6,
        "dependentGloss":".",
        "governor":5,
        "governorGloss":"singer"
      }
    ]
```


Fig. 1: Syntactic analysis of the sentence *„Eleanor was the queen"* generated from Stanford CoreNLP, more about *tokens*, *basicDependencies* and *enhancedDependencies* at [17].

- dobj - direct object,
- neg - negation modifier,
- amod - adjectival modifier,
- advmod - adverb modifier,
- advcl - adverbial clause modifier,
- aux - auxiliary,
- poss - possession modifier,
- mark - marker,
- nmod - nominal modifier,
- nsubjpass - passive nominal subject.

3. SEMANTIC MACHINE OF TRANSPARENT INTENSIONAL LOGIC

After syntactic analysis, semantic analysis of sentences is the second step of the natural language processing. Working with TIL, the output of the semantic analysis is a sentence represented as TIL construction.

Until now, it was created only one implementation of the TIL - its author is Aleš Horák [12]. His semantic machine is based on the **Normal**

Translation Algorithm (NTA). NTA enables the automated processing of texts in the Czech language and conversion into their logical representation to cover the most frequent linguistic phenomena. NTA consists of two steps - in the first, a parser of Czech sentences is created, which (in the second step) are semantically represented as TIL constructions. The NTA parser, called **SYNT** [13], generates outputs in the form of a tree of grammatical dependencies from input sentences. An evaluation is performed from the tree of grammatical dependencies, in which it proceeds from the leaves towards the root, while in the inner nodes the synthesis of values obtained from the child nodes is gradually performed. The leaves are evaluated using the **Lexicon** [14] tool. SYNT can be considered as the only existing semantic TIL machine.

TIL-Script [4] is a multiplatform declarative programming language based on TIL syntax. Table 1 contains selected TIL types and logical objects rewritten in TIL-Script.

TABLE 1: TIL syntax elements and their equivalents in TIL-Script

TIL	TIL-Script	Description
ι	Indiv	individual
o	Bool	truth value
ω	World	possible word
τ	Time / Real	time / real number
$\wedge, \vee$	And / Or	conjunction / disjunction
$\supset, \equiv$	Implies / Equiv	implication / equivalence
$\neg$	Not	negation
$\forall, \exists$	ForAll / Exist	universal / existential quantifier

In TIL is natural language processing performed in three consecutive steps [7] (shown in Figure 2):

1) type analysis - to each object of the sentence is assigned its type,
2) creation of construction - construction represents the meaning of a sentence,
3) type control - verifying if the previous step was correctly done (voluntary step).

The right side of the Figure 2 shown input processing. The input is firstly sent to SCN which provides syntactic analysis and return analyzed sentence in form of json. Then the semantic machine in the module *provide_type_analysis* makes a type analysis and saves all the recognized types into a list. The list should be passed to the module *create_construction*. After construction is successfully created, it is printed on the output of the semantic machine and the necessary data are supplied to the module *provide_type_analysis* (this part is a subject of future research).

3.1. Type analysis

The implementation of TIL semantic machine starts with the type analysis. The analyzed types extend basic atomic TIL types o, ι, τ, ω. The list of

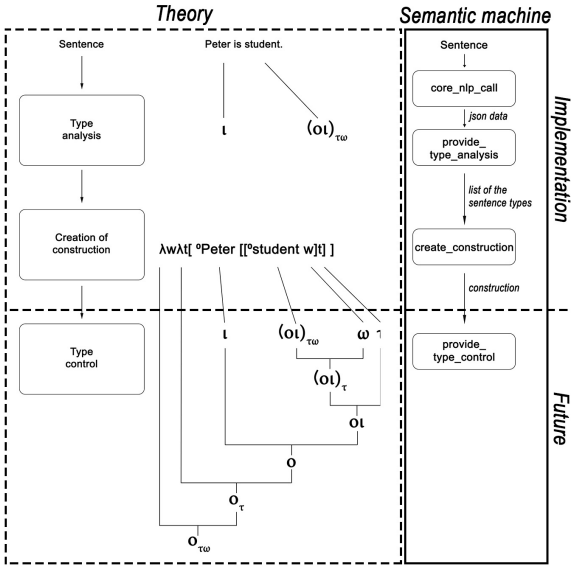


Fig. 2: Semantic analysis of sentences in TIL [1].

types used in the provide_type_analysis module and how to identify them via the SCN output is as follows:

Individual ι - basic type in TIL. The individual represents a specific entity that always starts with a capital. SCN marks it as *NNP* - a proper noun, singular (Figure 3).

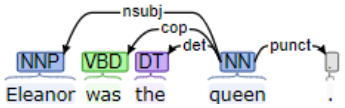


Fig. 3: Assignment of individual type.

Truth value o - basic type in TIL. In the sentence *"The queen was old."* word *"old"* represents truth value because it defines the state of the *"queen"* in some world and time. The truth value can be obtained as an independent component of the nominal subject *nsubj* (Figure 4).

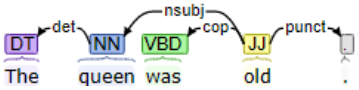


Fig. 4: Assignment of truth value type.

Attribute $((\iota\iota)\tau)\omega$ - an extended type in TIL. There is a need to find a dependency *case*, where an individual is the dependent component and the indicator *'s* represents the independent component. Then the dependency *nmod:poss* can be found - the dependent component is the individual and the independent component is the attribute which is owned by the individual. If there is no individual in the sentence, it is necessary to find *nmod:poss* dependency directly (Figure 5).

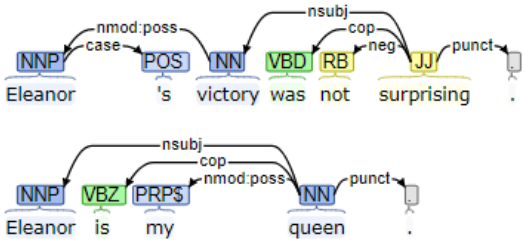


Fig. 5: Assignment of attribute type.

Property of the object $((o\iota)\tau)\omega$ - an extended type in TIL. It is necessary to find the *nsubj* relationship similar to the type proposition. The object to which the type is assigned in this case will not be the whole sentence, but only its verb phrase (Figure 6).

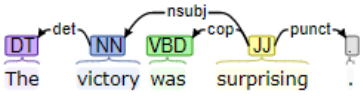


Fig. 6: Assignment of property of object type.

Property of the property $((((o\iota)\tau)\omega)((o\iota)\tau)\omega)$ - an extended type in TIL. To determine this type, there should be a dependency *amod*, where the dependent component presents property of property and the independent component is property assigned to the object (Figure 7).

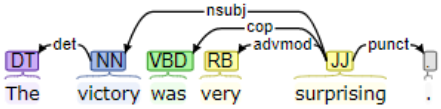


Fig. 7: Assignment of type property of property.

Binary relation between individuals $(((o\iota)\iota)\tau)\omega$ - an extended type in TIL. To determine this type, there should be a dependency *nsubj* or *nsubjpass*, where dependent component is individual and the independent component is a relation. Then dependency *dobj* with another individual as the dependent component and relation as the independent component should be found (Figure 8).

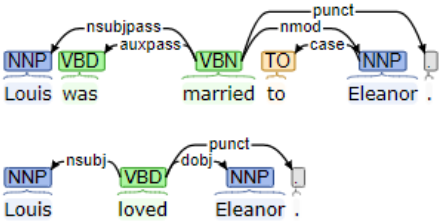


Fig. 8: Assignment of binary relation type.

Binary relation in general $(((o\iota)\iota)\tau)\omega$ - an extended type in TIL. A variant of the previous type, where the subject and object is not an individual, but any type (Figure 9).

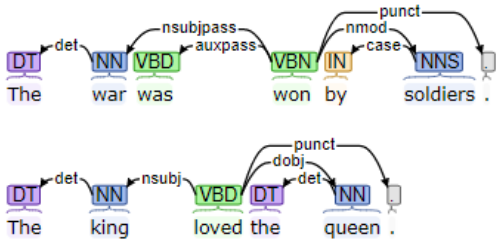


Fig. 9: Assignment of binary relation type.

Value τ - an extended type in TIL. An expression has this type when it is a cardinal number *CD*. Value can be expressed by the special nouns as *weight, height, length, temperature, depth, width, distance, speed, count, number...* (Figure 10).

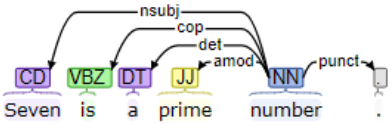


Fig. 10: Assignment of value type.

Property of the value $((\tau)\tau)\omega$ - an extended type in TIL. After identifying the value type in the sentence, the dependency *nsubj* can be found, where the dependent component is the value and the independent component is the property (Figure 11).

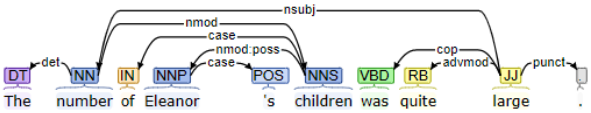


Fig. 11: Assignment of property of the value type.

All of the above-mentioned types have been implemented in TIL semantic machine. Correct type analysis is a prerequisite for the creation of construction, which is discussed in the next chapter.

3.2. Creation of constructions

In Figure 1 is a dependency *root* which is in every analyzed sentence and the creation of construction always starts from it. After the type analysis is finished, a list of all found types of lexical units is made. This list helps to determine what type is assigned to the word at the root of the sentence.

After identifying the type of the root of the sentence, it is determined with which words the root is related - this is a key property for building the composition of the construction. In the sentence

"Eleanor was the queen.", The root of the sentence is the word *"queen"*, which is a type property of the object. This type is intensional, from which it is possible to determine where the closure is in the sentence. Subsequently, a related lexical object *"Eleanor"* is found, which is of the individual type. The resulting construction generated by the TIL semantic machine looks like this:

$$\lambda w \lambda t\ [[[\text{\^{}0queen}\ w]\ t]\ \text{\^{}0Eleanor}] \quad (1)$$

The collisions arose during automated type checking when the same word seemed to belong to several categories of types. For example, in the sentence *"Eleanor had 10 children."*, the word 10 was identified as a number and should be a value type. This is not true, the sentence says that a particular individual has a certain property. Between *"Eleanor"* and *"children"* is the nsubj relationship, which is characteristic of property of the object type and based on the relationship between *"10"* and *"children"* it is obvious that *"10"* is a property of the property type. Therefore, a basic type check is performed between the lexical units that are related to each other, which excludes the possibility of inadequate type analysis.

The last step that needed to be processed in the synthesis of constructions was the use of logical connectives. The dependency *conj* was used to identify the $\wedge$, $\vee$, which are always in the root node. Processing connectives allows us to create semantic representations of very complex sentences - e.g. the sentence *"Louis VII was the husband of Eleanor of Aquitaine, Constance of Castile and Adele of Champagne."* is represented by the following construction:

$$\lambda w \lambda t[\text{\^{}0Louis\_VII}\ [[\text{\^{}0Eleanor\_of\_Aquitaine}\ [[\text{\^{}0husband}\ w]\ t]]\text{AND}\ [\text{\^{}0Constance\_of\_Castile}\ [[\text{\^{}0husband}\ w]\ t]]\text{AND}\ [[\text{\^{}0Adele\_of\_Champagne}\ [[\text{\^{}0husband}\ w]\ t]]]]] \quad (2)$$

Mark and *advcl* relationships were used to identify the implication $\Rightarrow$ - there must be a *mark* dependency between the dependent component and the word "if" and there must be an *advcl* relationship between the same dependent component and the time, consequence, conditional or purpose clause. The semantic representation of the sentence *"If Eleanor is the wife of Louis, then she is the queen of France."* is represented by the following construction:

$$\lambda w \lambda t[[\text{\^{}0Eleanor}\ [[\text{\^{}0wife\_of\_Louis}\ w]\ t]]\ \text{IMPLICATION}\ [\text{\^{}0Eleanor}\ [[\text{\^{}0queen\_of\_France}\ w]\ t]]] \quad (3)$$

The last logical object that needed to be processed when creating the structures was the negations in the sentences. A negation is in a sentence if there is *neg* dependency between any two words. The semantic representation of the sentence *"Eleanor was not a queen of Spain."* is represented by the following construction:

$$\lambda w \lambda t[^0\text{Eleanor NOT}\ [[^0\text{queen\_of\_Spain}\ w]\ t]] \quad (4)$$

The constructions 1, 2, 3 and 4 are examples of outputs generated by the TIL semantic machine.

4. Conclusion

In this paper, we have presented a functional implementation of the TIL semantic machine. Worldwide, this is only the second implementation of a semantic machine based on intensional logical principles. It can therefore only be compared with the Synt system (see chapter 3).

For example, the sentence *"If Eleanor is the wife of Louis, then she is the queen of France."* is analyzed in the Synt system as

$$[\backslash w[\backslash t[\text{Implies}[\text{'Eleanor 'wife\_of\_Louis@wt}]\ [\text{'Eleanor 'queen\_of\_France@wt}]]] \quad (5)$$

and in our TIL semantic machine as in equation 3.

Based on this comparison, it is clear that both semantic machines generate the same output, even though they use the different syntax (the output generated from Synt is written in TIL-Script).

However, the presented solution also contains certain shortcomings related to the fact that it is not a full implementation of the TIL potential, but only a prototype. This means that while it is possible to analyze most of the Czech language corpus through the Synt system, the use of our TIL semantic machine is partially limited because its development is still ongoing. Our TIL semantic machine has one indisputable advantage over the implementation of Synt - its source natural language is English, which gives this tool the potential for application in international research.

References

[1] B. Bednár, Z. Bilanová, and A. Pekár. The automated translation of natural language sentences into intensional logic at the type analysis and construction synthesis levels. In *Acta Electrotechnica et Informatica*, volume 19, pages 3–7, 2019.

[2] Z. Bilanová and M. Uchnár. Comparison of the approaches of montague and tichý within a logical analysis of an english sentence. *POSTER 2017*, pages 1–6, 2017.

[3] S. Bird, E. Klein, and E. Loper. *Natural Language Processing with Python - Analyzing Text with the Natural Language Toolkit*. O'Reilly Media, 2009.

[4] N. Ciprich, M. Duží, and M. Košinár. Til-script : Functional programming based on transparent intensional logic. *RASLAN 2007, P. Sojka, A. Horák, (Eds.)*, pages 37–42, 2007.

[5] M. C. de Marneffe and Ch. D. Manning. Stanford Typed Dependencies Manual, 2008.

[6] B. DeWilde. Textacy: Nlp, before and after spacy, 2019.

[7] M. Duží. Til as the Logic of Communication in a Multi-Agent System. *Research in Computing Science, special issue Advances in Natural Language Processing and Applications*, 3:27–40, 2008.

[8] M. Duží and P. Materna. Logical Form. *G. Sica, Essays on the Foundations of Mathematics ad Logic*, pages 115–153, 2005.
[9] M. Duží and P. Materna. *TIL jako procedurální logika, Průvodce zvídavého čtenáře Transparentní intensionální logikou*. aleph Bratislava, Slovak Republic, 2012.
[10] L. T. F. Gamut. *Logic, Language, and Meaning, Volume 2: Intensional Logic and Logical Grammar*. University of Chicago Press, 1990.
[11] S. Guido and S. Muler. *Introduction to Machine Learning with Python - A Guide for Data Scientists*. O'Reilly Media, 2016.
[12] A. Horák. *The Normal Translation Algorithm in Transparent Intensional Logic for Czech*. PhD thesis, Faculty of Informatics, Masaryk University, Brno, 11 2001.
[13] A. Horák and V. Kadlec. New meta-grammar constructs in czech language parser synt. *Text, Speech and Dialogue: 8th International Conference*, pages 85–92, 2005.
[14] A. Horák and K. Pala. Lexicons in til and verb valency frames. *Proceedings of the International Conference on Communications in Computing*, pages 255–261, 2004.
[15] G. S. Ingersoll, T. S. Morton, and A. L. Farris. *Taming Text - How to Find, Organize, and Manipulate It*. O'Reilly Media, 2012.
[16] S. Loria. Textblob: Simplified text processing, 2018.
[17] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.
[18] R. Montague. The Proper Treatment of Quantification in Ordinary English. In *Approaches to Natural Language*, 1970.
[19] J. Raclavský. On Partiality and Tichý's Transparent Intensional Logic. *Magyar Filozófiai Szemle*, 54:120–128, 2010.
[20] P. Tichý. The Foundations of Frege's Logic. In *De Gruyter, Berlin and New York*, 1988.

Zuzana Bilanová graduated from Technical University in Košice with a major in informatics and is currently working as an assistant professor. She has been focusing on logical analysis of natural language sentences, nontraditional logic systems in computer science, and the theory of types.

Martin Štancel graduated from Technical University in Košice with a major in informatics and is currently in the second year of the doctoral study. He has been focusing on machine learning and he has also been interested in creating the semantic machine of transparent intensional logic.

Automated Hardening of a Linux Web Server

Olenčin, Michal and Perháč, Ján

Abstract: ***This article is dedicated to the design and implementation of automated hardening a Linux web server after the clean installation. The article deals with the analysis of the cybersecurity, focus, and scope of the configuration. Based on that, was made a design of an automated security configuration with a detailed description. The main accomplishment of the work is a configuration script. The script is compared and evaluated with existing solutions and it achieved the best rating in security audits.***

Index Terms: ***cybersecurity, Linux security, Web security***

1. INTRODUCTION

THE PAPER is oriented on designing an automated hardening of a Linux web server and subsequently on implementing and evaluation of the designed solution. At first, a motivation for creating this paper is specified, afterward the current state of cybersecurity is analyzed. Then a focus of security and analysis is specified. The results of the analysis are taken into account in proposing a design of our configuration according to a wide range of resources.

The output of the implementation is an open-source script [12], [13] of an automated configuration implemented according to the proposed design. Finally, an evaluation of the implemented configuration was performed. That evaluation consists of a functional test, a compatibility test of the webserver configuration, and security audits of default and designed configurations. Also security audit by the Lynis tool was performed and also a score of existing open-source solutions measured and compared.

Our recent research in the field of computer security was focused on intrusion detection systems [10], [14], its formal description, and developing their models [11], [15] and securing the Linux web servers. In this paper, we are focused on the results of our recent work with the automated securing of the Linux webserver. We start with a brief description of the current state in the area. We compare existing solutions, and analyze technological possibilities. Then, we describe the details of the designs and our implementation. in the final section, we evaluate our implementation. We compare it with other popular solutions in the field by performing standard security audits.

Manuscript received May 2, 2020.

This work was supported by the Faculty of Electrical Engineering and Informatics at the Technical University of Košice under contract No. FEI-2020-70: "Behavioral model of component systems based on coalgebras and linear logic", and by the project KEGA 011TUKE-4/2020: "A development of the new semantic technologies in educating of young IT experts".

M. Olenčin (contact person) is an MSc student at the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia (e-mail: michal@olencin.com).

J. Perháč (contact person) is an assistant professor at the Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia (e-mail: jan.perhac@tuke.sk).

2. MOTIVATION

The cybersecurity and protection of sensitive data are frequently discussed topics today. There is a growing interest in this area mainly due to the increase in cyberattacks and theft of sensitive data.

After installation of a web server, a basic configuration is used and it is necessary to configure it as needed. The basic server setup includes many security issues that need to be corrected by an appropriate configuration for security reasons. The basic configurations are customized to maximize support and make development easier. The configuration process can be automated using programs written in a scripting language. Creating an automated configuration makes the process of configuration quicker and more effective.

Currently, there are very few solutions dealing with the automation of a Linux web server configuration in the desired complexity and with adequate protection. The main goal of this work is to create a solution on an appropriate level of protection and complexity by creating an automated configuration of a Linux web server.

3. THE CURRENT STATE OF CYBERSECURITY

We have defined the current state of cybersecurity by summarizing the most interesting results from following reports:

- *"Cisco 2018 Annual Cybersecurity Report"* by Cisco [3],
- *"Executive Summary - 2018 Internet Security Threat Report"* by Symantec [18],
- *"Cloud Security Report"* by Alert Logic [9].

The most interesting results e.g. occurrences of malicious programs dedicated to various areas increased in the years from 2015 to 2017 as following:

- cryptocurrencies by 8500%,
- IOT devices by 600%,

- ransomware attacks by 46%,
- malware on mobile devices by 54%.

3.1. Used technologies

In the first step, we have analyzed the usage of current technologies. We have taken into account the following criteria:

- a Linux distribution,
- a web service,
- database service,
- programming language.

Our main goal was to create a configuration for a wide group of potential users, therefore our main criterion was the popularity of the technology.

Because of long-term support and the high market share, we have chosen an Ubuntu Server as the supported distribution. Version Ubuntu Server 16.04.5 64-bit has been chosen, because of its stability. Ubuntu has 37.8% market share altogether. If the Debian distribution had been added as a supported distribution, the solution would have covered 61.1% of the whole market.

We have chosen an Apache webserver as a support web server, because of its 45.5% market share in its latest version Apache 2.4. The choice of the Nginx as a supported web server was also taken into account, but based on the balance between quantity and quality, Apache won. Both are comparable in performance, flexibility, and reliability [7]. If the Nginx web server had been added as a support web server, the solution would have covered ¾ of the market share.

As the support database service, MariaDB was chosen in its latest version of a database server, specifically MariaDB 10.3. The selections of MySQL and PostgreSQL as the support database server were also taken into account. PostgreSQL is robust, has a complex configuration, but is less powerful compared to others. MariaDB is a fork of MySQL, which is 5% more powerful [6] than MySQL. Also, its popularity tends to increase. But both, MySQL and PostgreSQL, are efficient and reliable database services and their support could be added in the future.

PHP was chosen as the support programming language in the latest version, because of its dominance on the market with 78.9%.

Results are summarized in the table 1 *"Result of selected security focus"* covering a wide range of usability.

TABLE 1: Result of selected security focus

Scope	Focus
Linux distribution	Ubuntu Server 16.04.5 64-bit
Web service	Apache 2.4
Database service	MariaDB 10.3
Programming language	PHP 7.2

3.2. Existing Solutions

There are several automated Web Linux server configuration solutions. Unfortunately, most solutions are not complex and only address the configuration of the Apache web service itself, or address a small configuration range. The solutions with the required complexity currently rarely exist.

Among the proprietary license solutions, the Lynis Enterprise solution is available [4]. Unfortunately, this solution is too complex for common needs and requires the purchase of a license, which is a disadvantage. This solution offers security audit, system security, vulnerability management, ample support for Unix operating systems, macOS and many more.

Among the open-source solutions, the *"JShielder"* [17] and the *"Ubuntu hardening"* [16] solutions are available. Both solutions are not overwriting existing configurations and do not use the latest versions of services. Ubuntu hardening solution also includes a smaller range of configurations compared to the JShielder solution.

Each of the mentioned solutions is implemented as a bash script.

While analyzing existing solutions, deficiencies have been identified that proposed solution seeks to eliminate. The proposed solution compared to existing solutions includes:

- configuration backup,
- use of the latest versions of services,
- it overwrites existing configurations,

with open-source availability, with the required complexity and configuration range for basic needs.

4. Design of our solution

The design of our solution is proposed according to the CIS Ubuntu Linux 16.04 LTS Benchmark [1], the CIS Distribution Independent Linux Benchmark [5], the CIS Apache HTTP Server 2.4 Benchmark [2], the book *"Practical Apache, PHP-FPM & Nginx Reverse Proxy: How to Build a Secure, Fast and Powerful Webserver from scratch"* [8], the configuration instructions from the non-profit organization Mozilla, the proposed Lynis configuration, the proposed Tiger configuration and the recommended set of rules for security2 module from the non-profit organization OWASP.

The design is targeted to achieve the highest possible security for common web servers and includes:

- Installation and hardening of the Apache web server and also security2, SSL, evasive, headers, include, http2 and reqtimeout modules.
- Installation and hardening of the MariaDB database server.

- Installation and hardening of the PHP programming language.
- Installation and hardening of the OpenSSH server.
- Security configuration of an operation system:
  - Hardening of directory permissions.
  - Hardening of the file system.
  - Hardening of network protocols.
  - Hardening of the kernel.
  - Hardening of passwords.
  - Hardening of the firewall.
  - Hardening of other areas.

The design of configuration for the Apache webserver includes adding sources with the latest stable version of a package manager, turning off unnecessary modules, limiting leaked information through web server response, setting up the service to run under a separate user, configuring security modules, request limits, logging level, permission rights for the service files, etc.

Among unnecessary modules that are `webdav`, `status`, `autoindex`, `proxy`, `userdir` and `info`.

Among security modules that are `security2`, `ssl`, `evasive`, `headers`, `include`, `http2` and `reqtimeout`.

The tables 2 and 3 presents our configuration of the Apache in detail.

TABLE 2: Configuration of the Apache

Parameter	Value
DocumentRoot	/var/www/html/
FileETag	None
Timeout	10
LimitRequestBody	102400
LimitRequestFieldsize	1024
LimitRequestline	512
LogLevel	notice core:info
Protocols	h2 http/1.1
ServerTokens	Prod
TraceEnable	off

TABLE 3: Configuration of the SSL module

Parameter	Value
SSLCipherSuite	*Based on SSL generator*
SSLCompression	off
SSLHonorCipherOrder	on
SSLInsecure-Renegotiation	off
SSLProtocol	all -SSLv3 -TLSv1 -TLSv1.1
SSLSessionTickets	off
SSLStaplingCache	"shmcb:/var/run/ocsp(128000)"
SSLStapling-ResponderTimeout	5
SSLStaplingReturn-ResponderErrors	off
SSLUseStapling	on

Next, the configuration of the MariaDB database server includes adding a source with the latest stable version of a package manager, running the built-in secure installation, preventing access to services outside the local network and access to local files through the database and configuring the port. The MariaDB database server does not require complex hardening configuration in general.

The table 4 presents our configuration of the MariaDB in detail.

TABLE 4: Configuration of the MariaDB

Parameter	Value
bind-address	127.0.0.1
general-log	0
local-infile	0
port	*Depends on input*

Furthermore, the configuration of the PHP programming language includes also adding a source with the latest stable version of a package manager, disabling a debug and error messages, also disabling dangerous functions, limiting leaked information through web service, configuring session, cookies, and the limit of access to system files, etc.

Among dangerous functions that are for example process control functions, functions of the POSIX standard, program execution functions, functions that provide sensitive information and much more.

The table 5 presents our configuration of the PHP in detail.

TABLE 5: Configuration of the PHP

Parameter	Value
allow_url_fopen	0
allow_url_include	0
disable_functions	*Depends on design of configuration*
display_errors	0
display_startup_errors	0
expose_php	0
html_errors	0
log_errors	1
open_basedir	/var/www/html
post_max_size	100K
session.use_strict_mode	1
session.cookie_httponly	1
session.cookie_secure	1

Configuration of the OpenSSH (version 7.2) server includes installation of the package with a model for server, limiting leaked information through services, port configuration, configuring user authorization, logging level, authorization message, cryptography algorithms, enabling only public key authentication and much more. Upon public key authentication, i.e. a key pair authentication is important to have a private key secured against theft. For maximum security of private key can be used universal 2nd factor *(U2F)*, one-time password *(OTP)* or another form of multi-factor authentication *(MFA)* can be used.

Tables 6, and 7 presents our configuration of the OpenSSH in detail.

Operation system configuration consists of the following:

TABLE 6: Configuration of the OpenSSH authentication

Parameter	Value
ChallengeResponseAuthentication	no
GSSAPIAuthentication	no
HostbasedAuthentication	no
PasswordAuthentication	no
PubkeyAuthentication	yes
RhostsRSAAuthentication	no
RSAAuthentication	no

TABLE 7: Configuration of the OpenSSH

Parameter	Value
Banner	/etc/issue.net
AllowAgentForwarding	no
AllowGroups	ssh
AllowTcpForwarding	no
Ciphers	*Depends on design*
ClientAliveInterval	300
Compression	no
DebianBanner	no
IgnoreRhosts	yes
KexAlgorithms	*Depends on design*
LoginGraceTime	60
LogLevel	VERBOSE
MACs	*Depends on design*
MaxAuthTries	2
MaxSessions	2
PermitRootLogin	no
Port	*Depends on input*
Protocol	2
TCPKeepAlive	no
UseDNS	no
UsePrivilegeSeparation	SANDBOX
X11Forwarding	no

- Hardening of directory permissions includes the configuration of sticky bit on all world-writable directories, set of basic permissions rights and configuring the permissions for the sensitive system files.
- The file system is in this way configured so that it includes disabling some unnecessary files systems, configuring USB devices and mounting partitions.
  - Among unnecessary files systems are the following: `cramfs`, `squashfs`, `freevxfs`, `jffs2`, `hfs`, `hfsplus` and `hfsplus`.
- The hardening of network protocols includes disabling some unnecessary network protocols.
  - Among unnecessary network protocols that are the following: DCCP, SCTP, RDS and TIPC.
- The hardening of the kernel consists of enabling address space layout randomization, disabling SysRq key, configuring packets and so on.
- The hardening of other areas includes uninstalling unnecessary packages, installing additional security packages and so forth.
  - The following packages belong to the additional security packages included: acct, aide, aide-common, apt-show-versions, arpwatch, auditd, clamav, clamav-daemon, debsums, fail2ban, htop, ntp, sysstat, unattended-upgrades and usbguard.
  - The designed configuration also configures some additional security packages, e.g. auditd, unattended-upgrades and usbguard.
  - Among unnecessary packages that are the following: avahi-daemon, binutils, cups, cups-common, gcc, git, make, snapd and telnet.

The designed configuration, however, might be incompatible with some web applications and websites. This is caused by individual requirements for the configurations of those applications and websites. Additional configuration adjustments are needed after the automated configuration is applied.

5. Implementation and Evaluation of Our Solution

Our solution is implemented in the form of a bash script, and it is implemented according to the designed configurations. The implementation also includes a password generator. This generator is useful when SSH key is generated and the password for MariaDB root account is set up. Further, the whole source code is commented for easy understanding and orientation. In the implementation, the port for OpenSSH and MariaDB is obtained from the user. If the user presses the enter key, the default port is set.

Furthermore, the evaluation was done on a virtual private server for the most realistic conditions and also on a virtual machine. As per functional testing, the script is fully functional on Ubuntu Server 16.04.5 distribution, but also in its newer version 18.04.2. Later we have tested our solution on the newest version for the latest non-lts version Ubuntu 19.04. Our solution for this version is also fully functional. We have selected the Oracle VM VirtualBox as a hosted hypervisor for a virtual machine. For a cloud infrastructure, the DigitalOcean provider was selected.

After the compatibility test of the web server configuration, we have found out that the designed configuration requires small additional configuration adjustments depending on a web application or a website. The compatibility test was done on both types of web pages, with static and also dynamic contents.

Security audits were performed using Lynis tool, Nmap tool, SSL Server Test by the Qualys, Mozilla HTTP Observatory tool and Mozilla SSH Observatory tool. Results of security audits are summarized in the table 8 *"Result of all security audits in a designed configuration"* and in the table 9 *"Result of all security audits in a default configuration"*.

TABLE 8: Result of all security audits in a designed configuration

Security audit	Result
Lynis	97 / 100
Nmap	a few
SSL Server Test	A+
Mozilla HTTP	A+
Mozilla SSH	A

TABLE 9: Result of all security audits in a default configuration

Security audit	Result
Lynis	56 / 100
Nmap	a lot
SSL Server Test	B
Mozilla HTTP	F
Mozilla SSH	C

The Nmap tool audit results are rated based on found leaking information without any grading scale. The SSL Server Test audit gives grades A-, B, C, D, E, F *(sorted from the best to the worst)* or T *(if found some trust issues)*. The Mozilla HTTP audit gives grades A+, A, A-, B+, B, B-, C+, C, C-, D+, D, or D- *(sorted from the best to the worst)*. The Mozilla SSH audit gives grades A, B, C, D, or F *(sorted from the best to the worst)*.

Within the security audit using the Lynis we also measured a score of existing open-source solutions. These measurements have been adapted to achieve as highest score as possible. The Lynis security audit was performed with version 2.7.1 of the tool. We have compared our solution with other existing configurations, and the default configuration with the following results:

- The default configuration has achieved a score of **56** / **100**.
- The Ubuntu hardening solution has achieved a score of **72** / **100**.
- The JShielder solution has achieved a score of **89** / **100**.
- Our solution of configuration has achieved a score of **97** / **100**, which is the best rating when compared to existing solutions.

Unfortunately, the designed configuration did not achieve full score because of the absence of GRUB password for booting, presence of an antivirus program and separate mounting of partitions `/tmp`, `/home` and `/var`. Creating a GRUB password is unwanted on remote computers. Configuring separate mounting of partitions was performed during the installation of the operating system. Additional configuration of mounting partitions is complicated. In the designed configuration a ClamAV as antivirus is installed, but the security audit by the Lynis tool did not detect it.

The security audit for the Nmap tool detected much less sensitive information leaked for the designed configuration compared to the default configuration, such as information of server distribution and a version of Apache.

The security audit performed by the SSL Server Test scored on the designed configuration an **A+** rating. The default configuration achieved a **B** rating for using weak cryptography algorithms and not providing forward secrecy to all browsers. With the security audit done by the Mozilla HTTP Observatory tool the designed configuration has achieved an **A+** rating. The default configuration achieved an **F** rating for missing security-related web service response headers. In the security audit by the Mozilla SSH Observatory tool, the designed configuration has achieved an **A** rating. The default configuration achieved a **C** rating for using weak cryptography algorithms.

Results of security audits are summarized in table 10 *"Result of Lynis security audit"*.

TABLE 10: Result of Lynis security audit

Solution	Score
Implemented solution	97
JShielder	89
Ubuntu hardening	72
Default configuration	56

To summarize the implemented solution has the following advantages compared to other existing solutions:

- The design is drawn up according to the use of a large number of resources. For instance, CIS benchmarks guidelines, the proposed Lynis configuration, etc.
- The implemented solution is designed with the required complexity and configuration range for common needs.
- The implemented solution is free and open-source.
- The design uses the latest versions of services.
- The design overwrites existing configurations.
- The design includes configuration backup.

6. Conclusion

In this work, we analyzed the current state of cybersecurity, and we concluded that is always necessary to address cyber security due to the increasing incidence of cyber attacks.

In the analysis of existing solutions, deficiencies have been identified that have been eliminated by the proposed solution. Our solution has following properties:

- it is an open-source solution,
- it creates a backup of existing configurations,
- it uses the latest service versions,
- it overwrites existing configurations,

- it is designed with the required complexity and scope of configuration for common needs included.

Based on the analysis, the design of the solution was made by using recommended configurations from the Lynis audit and guidelines from the non-profit organization CIS. Besides that, we used also recommended instructions from the non-profit organization Mozilla, recommended configurations from the Tiger audit, recommended set of rules for `security2` module from the non-profit organization OWASP and recommended configuration guidelines from the book *"Practical Apache, PHP-FPM & Nginx Reverse Proxy"* [8].

In conclusion, the result of this work is a fully functional configuration script, which in comparison to existing solutions achieved the best rating in Lynis security audits.

The script is easily expandable and editable, covers a large scope of applications and utilities by market share, comprehensively solves the security of the webserver and the operating system.

On another hand, the script requires additional configuration, requires a knowledge of Linux configuration and partially rewriting the configurations of the Apache web service.

The work may be extended by including the compatibility for Debian distribution, including the Nginx web service support and including support for other database services such as MySQL and PostgreSQL.

However, the used security analysis tools used do not cover the full range of potential vulnerabilities. The security of an entity depends on the weakest element. The designed solution has improved web server security against known vulnerabilities, but vulnerability to unknown errors cannot be eliminated. In addressing the security problem, care must be taken to continually improve and mend the security of existing solutions, monitor these solutions and maintain cyber hygiene.

References

[1] CIS. Cis ubuntu linux 16.04 lts benchmark, December 2017.

[2] CIS. Cis apache http server 2.4 benchmark, July 2018.

[3] Cisco. Cisco 2018 annual cybersecurity report, February 2018.

[4] CISOfy. Lynis enterprise, 2019.

[5] CIS. Cis distribution independent linux benchmark, December 2017.

[6] Russel J.T. Dyer. *Learning MySQL and MariaDB*. O'Reilly Media, 1 edition, April 2015.

[7] Douglas Kunda, Sipiwe Chihana, and Muwanei Sinyinda. Web server performance of apache and nginx: A systematic literature review. *IISTE*, 8(2), March 2017.

[8] Adrian Ling. *Practical Apache, PHP-FPM & Nginx Reverse Proxy: How to Build a Secure, Fast and Powerful Webserver from scratch*. Adrain Ling Kong Heng, 1 edition, May 2015.

[9] Alert Logic. Cloud security report, 2017.

[10] Daniel Mihályi and Valerie Novitzká. A coalgebra as an intrusion detection system. *Acta Polytechnica Hungarica*, 7(2):71–79, 2010.

[11] Daniel Mihályi and Valerie Novitzká. Towards to the knowledge in coalgebraic model ids. *Computing and Informatics*, 33(1):61–78, 2014.

[12] Michal Olenčin. Alfavio/Automated-securing-of-Linux-web-server v.1.0.0, June 2019.

[13] Michal Olenčin and Ján Perháč. Automated configuration of a linux web server security. In *2019 IEEE 15th International Scientific Conference on Informatics*, pages 304–308. IEEE, 2019.

[14] Ján Perháč, Daniel Mihalyi, and Lukaš Mat'aš. Resource oriented bdi architecture for ids. In *2017 IEEE 14th International Scientific Conference on Informatics*, pages 293–298. IEEE, 2017.

[15] Ján Perháč and Daniel Mihályi. Coalgebraic modeling of ids behavior. In *2015 IEEE 13th International Scientific Conference on Informatics*, pages 201–205, November 18-20, 2015, Poprad, Slovakia, Danvers: IEEE, 2015.

[16] Joel Radon. Ubuntu hardening, May 2019.

[17] Jason Soto. Jshielder, March 2019.

[18] Symantec. Executive summary - 2018 internet security threat report, March 2018.

Michal Olenčin received his bachelor's degree in Informatics from the Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Košice, Slovak Republic in 2019. His research interests include cybersecurity, computer security, security of operating systems.

Ján Perháč received his PhD. in Informatics from the Technical University of Košice, in 2019. Currently, he is an assistant professor at the Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. His research topics include computer security, category theory, logical systems, type theory, and semantics of programming languages.

Edge Coloring of Set of Graphs with The Use of Data Decomposition and Clustering

Dudáš Adam; Škrinárová Jarmila

Abstract: ***This paper presents solution to a problem of parallel edge coloring of sizable sets of cubic graphs. Since edge coloring of the graph is considered to be non-dividable operation which can be, in some cases, computed several hundreds to thousands of times, use of the parallel computing is necessary. The paper presents design and implementation of methodology based on clustering of graphs and data decomposition model. Methodology aims to satisfy time criterium of effective decomposition of problem to set of subproblems. Proposed methodology was experimentally tested on all available datasets related to group of graphs called snarks.***

Index Terms: ***parallel computing, distributed computing, decomposition, graph coloring, snark***

1. Introduction

THIS paper is motivated by the need for optimization of time of edge coloring of graph sets. Edge coloring is NP-complete problem which invites use of high-performance computing systems and specialized algorithms [1] and can be specified as task of assigning colors to the edges of graph.

Coloring of graphs is frequently used in scheduling, frequency allocation, compiler optimization (specifically register allocation), and pattern matching problems.

Main objective of this paper is to show that, with the use of correct tools and parallel computing, we are able to compute edge coloring of graphs more effectively. We are focused on the effectivity from the point of view of time demands and from the point of view of correct decomposition of problem - division of problem into set of smaller subproblems which can be computed in parallel or can be distributed over computing system.

In the paper, we present methodology which uses number of concepts:

- Parallel and distributed computing.
- Graph permutations – various orders of edge coloring of a graph.
- Clustering of graphs.
- Data decomposition – decomposition of a dataset into smaller sets computed on distributed basis.

The rest of this paper consists of three main sections ordered as follows.

Section 2 of presented paper contains works related to the topic of this paper.

In the Section 3, we describe basic terms related to edge coloring of sets of cubic graphs. In the Subsection 3.3 of this section, we present older edge backtracking algorithm based on breadth-first search.

Section 4 of the paper contains information about data that we use in experimental part of paper. Also, key concept of graph permuting is presented in this section.

In the Section 5, we propose methodology for parallel coloring of set of graphs and present experimental results.

2. Related work

Our previous work related to using parallel computing systems to solve problem of edge coloring of graphs was focused on the one-graph-at-a-time approach to coloring [2].

Problem of edge coloring of graphs, which is solved and experimentally tested in this paper is related to well-known Four-color theorem presented in [3]. The work of authors of [3], [4], [5] and [6] proves the four-color theorem, that every loopless planar graph admits a vertex coloring with at most four different colors.

Among the most significant algorithms used for edge coloring of graphs there are Edge color algorithm by L. Kowalik [7], heuristic for edge coloring of graphs by authors of [8] and edge backtracking algorithm based on breadth-first search presented in [9].

Newer works focused on the coloring of graphs are consisting of [10] and [11]. Authors of [10] created population-based memetic algorithm for solving the equitable coloring problem. The equitable coloring problem deals with finding the smallest k for which an equitable legal k-coloring exists. Approach of [11] focuses on computations performed on a set of 35 large benchmark graphs with 450–4000 vertices.

3. Edge coloring of the set of cubic graphs

Problem of proper edge coloring of cubic graphs, described in this section, is classified as

Manuscript received ...

A. Dudáš (contact person) is a PhD. student at the Faculty of Management and Informatics, University of Žilina, Slovakia and works at the Faculty of Natural Sciences, Matej Bel University, Slovakia (e-mail: adam.dudas@fri.uniza.sk).

J. Škrinárová works at the Faculty of Natural Sciences, Matej Bel University, Slovakia (e-mail: jarmila.skrinarova@umb.sk).

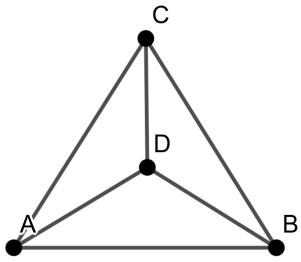


Fig. 1: Example of a simple cubic graph

NP-complete problem [1]. Graph G consists of [12]:

- vertices - elements of set $V(G)$. In the Fig. 1 vertices are labeled with capital letters A, B, C and D.
- edges – elements of set $E(G)$ - edge is a connection between two vertices, therefore, we can label it with the names of these two vertices.

Graph G is pair of sets V and E, where elements of the set E are double element subsets of the set V [14]:

$$G = (V, E), E \subseteq V^2 \tag{1}$$

In the case that vertex V is of degree equal to 3, we use $deg(V) = 3$. This concept represents number of edges, which are incident to a given vertex (since we work strictly with undirected graphs, by incident, we mean connected to the vertex in any way). Highest (maximal) degree of vertex in a graph G is denoted by $\Delta(G)$. In this paper we consider strictly cubic graphs. Graph is cubic when all of its' vertices are of degree three (see Fig. 1).

Since cubic graphs are smallest non-trivial graphs, coloring of these graphs should be less time consuming than coloring of graphs where $\Delta(G) > 3$. Therefore, they are fitting starting point for optimization of computation of graphs coloring.

3.1. Edge Coloring of Graphs

Edge coloring of graph is operation of assignment of colors to individual edges of graph. Coloring is called proper when there is no conflict in the coloring of given graph, this means that no vertex of given graph is incident to two or more edges colored with same color (see Fig. 2). Lowest number of colors usable in proper edge coloring of graph G is called edge chromatic index of graph G, and it is denoted by $\chi'(G)$ [12].

Vizings' theorem [14], which says that minimal number of colors needed for coloring of graph is in the interval $< \Delta(G), \Delta(G) + 1 >$, holds true.

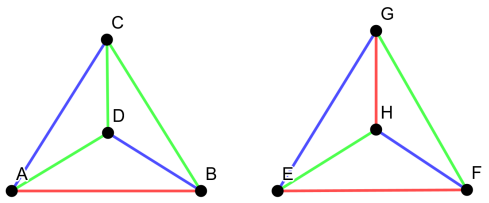


Fig. 2: Improperly (left) and properly (right) colored cubic graph

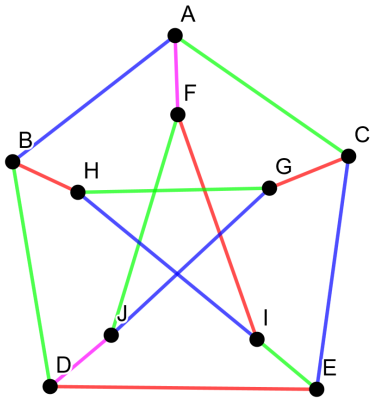


Fig. 3: Petersen graph – smallest known snark

Formal notation of Vizings' theorem focused on minimal number of colors:

$$\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1 \tag{2}$$

where $\Delta(G)$ is maximal degree of vertex in the graph G and $\chi'(G)$ is edge chromatic index of the graph G. Since every vertex of cubic graph is of degree equal to three, we consider three or four colors for proper coloring of cubic graph.

3.2. Edge 3-uncolorable Cubic Graphs

Cubic graph G with chromatic index equal to three is called edge 3-colorable [15]. There is small group of cubic graphs which need four colors for their proper coloring. Graphs from this group of cubic graphs are called edge 3-uncolorable or snarks [15]. Chromatic index of snarks is $\chi'(G) = 4$. In order to find out whether given graph G is snark, we need to edge color it with the use of three colors.

Therefore, coloring algorithm needs to check every possibility of edge 3-coloring of the graph G. Algorithms are able to decide whether the graph is snark or not after checking all possible edge colorings with the use of three colors.

Example of smallest know snark is presented on the Fig. 3.

3.3. Edge Backtracking Algorithm

Edge backtracking algorithm works on the basis of edge coloring of graph with predetermined succession of three colors. In the case that algorithm finds conflict in the coloring of the graph, it backtracks to the previous edge, recolors the edge and continues in coloring. If there are no possible proper colorings of problematic edge, algorithm backtracks even further back - to the edge which precedes both of the recolored edges. Algorithm continues in this way until either whole graph is colored properly, or until algorithm examines all possible ways of edge coloring of the given graph.

Time complexity of edge backtracking algorithm is $O(2^n)$, where n is number of vertices of given graph.

This algorithm is used in the parts of proposed algorithm where number of graphs are colored in parallel.

4. PRECONDITIONS AND DATA FOR PROPOSED METHODOLOGY

This section of the paper contains information about data that we use in the part of the paper focused on experiments. Also, key concept of graph permuting is presented.

Data used in this paper's experiments come from online database of graphs called House of Graphs [13]. We use part of the database dedicated to snarks. Specifically snarks consist of 34 to 40 vertices.

Table 1 presents number of graphs in each set – graphs are divided into sets on the basis of number of their vertices (denoted by v) and then divided into subsets according to mathematical properties of these graphs. Even though we don't work with the following properties yet, we present them here for readability:

- Girth – denoted by g – length of the shortest cycle of given graph G.
- Cyclic edge connectivity – denoted by λc – number of edges which must be removed from graph G to disconnect it into two discrete components, each containing a cycle.

Chosen graphs are all known snarks generated by graph generators minibaum and snarkhunter [13]. Plus sign in the Table 1 denotes the fact that not all of the graphs from the given subset were generated.

v	$g \geq 4$	$g \geq 5$	$g \geq 6$	λc
34	25 286 953	3 833 587	N/A	19 935
36	404 899 916	60 167 732	1	180 612
38	N/A	19 775 768+	39	35 429+
40	N/A	N/A	25+	N/A

TABLE 1: Number of graphs in used datasets

Key concept of presented research is graph permuting. By permuting of graph G, we create new graph G' with same structure but different order of vertices and edges. We say that graphs G and G' are isomorphic. To generate set of isomorphic graphs, we use simple matrix multiplication of the adjacency matrix of graph G (see Relation 3). With this operation we can create great number of permutations of given graph G.

$$A' = P^T * A * P \tag{3}$$

where A is adjacency matrix of given graph G, P is permutation matrix (matrix containing exactly one 1 in every row and column), P^T is transposed permutation matrix P and A' is matrix for new graph G' isomorphic to G.

Main idea behind graph permutations was set in our previous research presented in [2], [16]. We found out, that various permutations of same graph are edge colored in different times. In [16], we were able to find small set of permutations which would help us minimize computational time of coloring of chosen set of graphs.

Similar to [16], we prepared sets of permutations (called P) for experiments presented in this paper. For each one of four main sets of snarks (34, 36, 38 and 40-vertex snarks), we prepared set of 500 permutations – orders of coloring of graph. These sets of permutations were created via random generation of permutations which were, then, tested on graphs from given set. These permutations are those, which after applying them on graphs (as described in relation 3) produce lowest time of coloring of this permuted graph.

5. METHODOLOGY FOR PARALLEL COLORING OF SETS OF GRAPHS

We present methodology designed on the basis of preconditions listed in Section 4. This methodology consists of two main ideas related to decomposition in parallel computing - division of problem into set of smaller subproblem which can be computed in parallel or distributed over computing system.

First concept is one of the data decompositions – partitioning of data into data parts which are then distributed to the nodes of used computing system.

Second key concept is based on the need for effective decomposition of problem of edge coloring set of snarks. Decomposition of any problem is most effective in the case when all subproblems, the original problem is divided to, take same time to compute.

In our case, we use an idea of clustering of graphs to the groups based on the time of their edge coloring while using specific permutation. Therefore, we are able to set condition which secures similar time of edge coloring of all graphs.

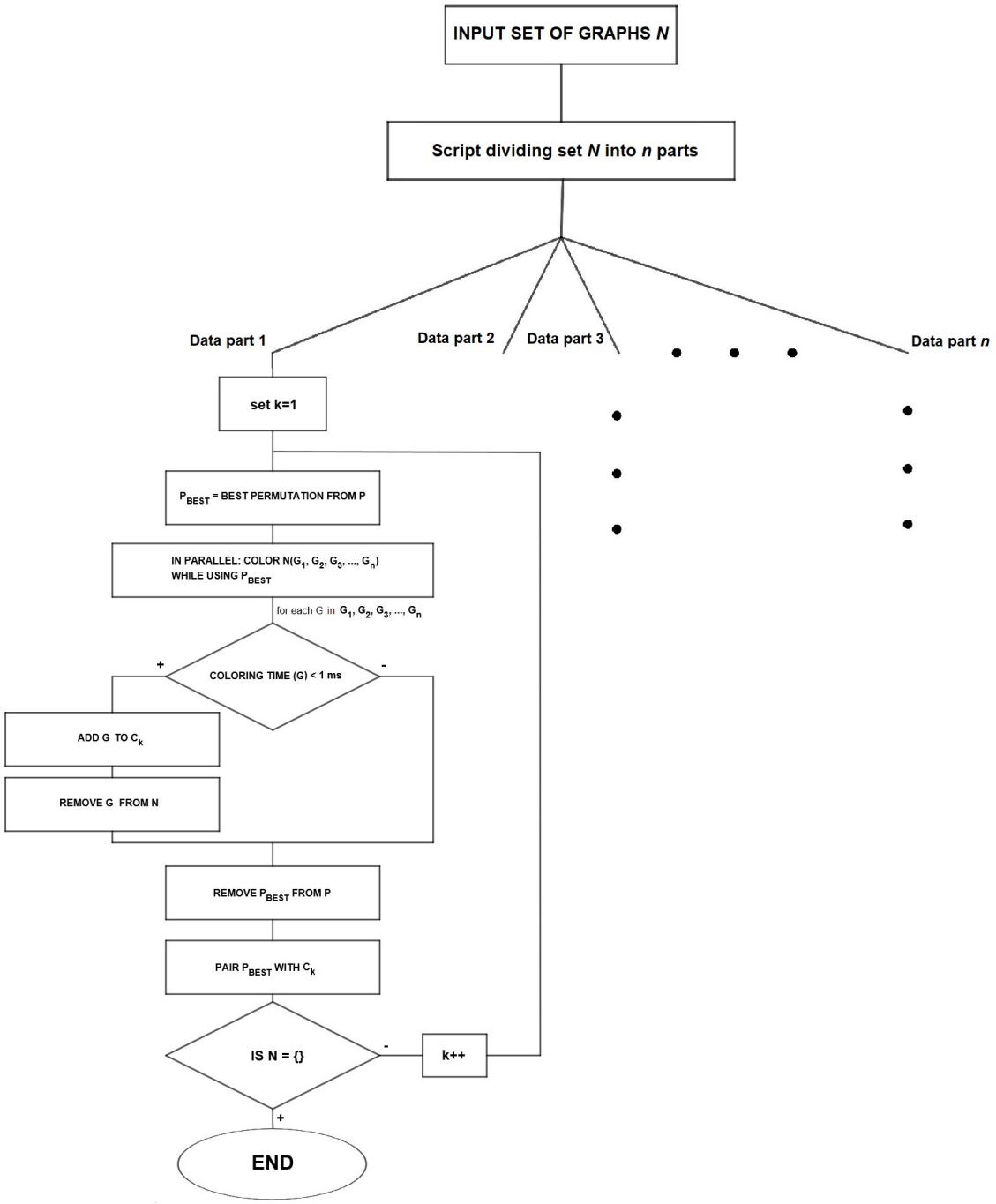


Fig. 4: Algorithm for Parallel Coloring of Sets of Graphs

5.1. Algorithm for Parallel Coloring of Sets of Graphs

We designed algorithm, which secures decrease in time of coloring of graph set and meet the condition of effective decomposition – time constraint was set to one millisecond according to experience gained in previous research [16]. This means that our similar enough time for edge coloring of graphs from the given set is set to one millisecond. As an input for the algorithm we use set of graphs N and set of permutations P – these permutations were chosen from the results of previous research, we can call them heuristically best permutations. Graphs in the input set N consist of the same number of vertices and edges.

Presented algorithm is described as follows:

- Division of graph set N into data parts. Algorithm divides input set of graphs N into n smaller parts with using a division script. Specifically, script divides input data set into

approximately equal subsets of circa 500 000 elements.
- Distribution of data parts. Based on data decomposition model, data parts (subsets) created in previous step, are distributed over available computing system.
- In this step, proposed algorithm executes individual steps based on clustering of graphs for each one of n data parts created from input dataset N:
  - Choose permutation with the lowest time of coloring P^{BEST} from P and apply it on whole data part.
  - Edge color the graphs in data part in parallel using parallel thread model. Each graph is colored with the use of Edge backtracking algorithm (see Subection 3.3).
  - Check constraints for all the graphs in the data part: Was a graph G edge colored in the time under one millisecond?
  - Create cluster of graphs which satisfy the condition above.
  - Allocate P^{BEST} to the created cluster of graphs.
  - Remove P^{BEST} from the set P, remove graphs with coloring time under one millisecond from data part.

  These six steps are repeated until given data part is empty or until algorithm tries all permutations unsuccessfully (this would mean, that there is graph in the set N, which cannot be colored in time lower than one millisecond).

Output of the algorithms consists of set of files containing clusters of graphs with their allocated permutations for each one of n data parts. In order to collect results, it is necessary to use agglomeration function on the set of files. This function connects files (clusters) with the same permutation.

Proposed algorithm is visualized on the Fig. 4.

5.2. Experiments Using Proposed Parallel Coloring of Sets of Graphs

Main objective of presented experiments was to test effectiveness of proposed algorithm on all available (and interesting) datasets.

All presented values were measured on the part of High-Performance Computing Cluster at Matej Bel University, Banská Bystrica, Slovakia. Used part of computing system consisted of 6 nodes, each node contained 12 Intel Xeon processors, connected via 40Gbps InfiniBand network.

In the Tables 2 – 11, we present measured values of:

- P^{BEST} - ID of permutation assigned to given cluster.
- Computing time – time of coloring of all graphs in given cluster.
- Memory needed – memory needed in coloring the whole cluster of graphs.
- Colored $>$ 1 - Number of graphs colored in time higher than one millisecond. These graphs need to be colored using a different permutation.

1) 34-vertex snarks: First set of snarks we used in presented experiments were 34-vertex snarks. This dataset was divided into 3 subsets of 19 935, 3 833 587 and 25 286 953 graphs.

P^{BEST}	Computing time [hh:mm:ss]	Memory needed [GB]	Colored >1 ms
1	00:00:31	0.341	39
2	00:00:02	0.092	0

TABLE 2: Clusters of graph set containing 19 935 graphs

P^{BEST}	Computing time [hh:mm:ss]	Memory needed [GB]	Colored >1 ms
1	01:06:58	994.598	532 364
2	00:07:00	32.059	57 274
3	00:00:49	1.457	5 775
4	00:00:07	0.254	0

TABLE 3: Clusters of graph set containing 3 833 587 graphs

P^{BEST}	Computing time [hh:mm:ss]	Memory needed [GB]	Colored >1 ms
1	04:19:09	3219.341	1 356 175
2	00:11:43	12.242	127 775
3	00:01:26	3.362	15 032
4	00:00:16	0.845	6
5	00:00:01	0.017	0

TABLE 4: Clusters of graph set containing 25 286 953 graphs

Looking at number of clusters created in the experiments shown in the Tables 3 and 4, it appears that there is not big difference between coloring more than 3,8 million graphs and more than 25,2 million graphs.

2) 36-vertex snarks: Set of snarks with 36 vertices contains four subsets. In Table 5 we present measured values for the subset of 180 612 snarks, in Table 6 measurements for second largest set of graphs, which contain 60 167 732 snarks and Table 7 presents measured values for the largest available snark set of 404 899 916 graphs.

Besides subsets presented in the Tables 5 – 7, this group of graphs contains subset of graph with girth ≥ 6, which was colored in the time under one second.

P^{BEST}	Computing time [hh:mm:ss]	Memory needed [GB]	Colored >1 ms
1	00:01:10	1.326	4008
2	00:00:07	0.839	241
3	00:00:07	0.376	12
4	00:00:01	0.017	0

TABLE 5: Clusters of graph set containing 180 612 graphs

P^{BEST}	Computing time [hh:mm:ss]	Memory needed [GB]	Colored >1 ms
1	08:21:18	4372.881	4 229 360
2	00:34:31	35.237	256 902
3	00:03:11	1.245	53 487
4	00:01:07	0.891	16 343
5	00:00:33	0.873	8 901
6	00:00:31	0.754	8 042
7	00:00:04	0.019	0

TABLE 6: Clusters of graph set containing 60 167 732 graphs

P^{BEST}	Computing time [hh:mm:ss]	Memory needed [GB]	Colored >1 ms
1	47:16:14	8472.230	17 642 937
2	03:32:49	56.671	12 876 022
3	00:56:11	30.290	4 038 765
4	00:40:26	12.128	1 844 397
5	00:38:39	9.133	1 000 036
6	00:07:12	2.465	321 009
7	00:07:12	1.026	162 932
8	00:00:22	0.891	21 329
9	00:00:20	0.806	17 890
10	00:00:18	0.650	12 993
11	00:00:17	0.521	9 636
12	00:00:13	0.360	4109
13	00:00:10	0.332	1572
14	00:00:03	0.196	60
15	00:00:02	0.183	38
16	00:00:03	0.067	0

TABLE 7: Clusters of graph set containing 404 899 916 graphs

3) 38-vertex snarks: We used 38-vertex snarks as the third set of graphs for presented experiments. This dataset is divided into 3 subsets of 35 429, 39 and 19 775 768 graphs. Both first and third subset are not fully generated – in the future, there is possibility of growth of these sets of snarks.

P^{BEST}	Computing time [hh:mm:ss]	Memory needed [GB]	Colored >1 ms
1	00:00:42	0.630	1053
2	00:00:11	0.439	0

TABLE 8: Clusters of graph set containing 35 429 graphs

P^{BEST}	Computing time [hh:mm:ss]	Memory needed [GB]	Colored >1 ms
1	00:00:03	0.371	0

TABLE 9: Clusters of graph set containing 39 graphs

P^{BEST}	Computing time [hh:mm:ss]	Memory needed [GB]	Colored >1 ms
1	03:16:02	3018.03	594 945
2	00:01:33	1.139	10 709
3	00:00:03	0.732	4 283
4	00:00:01	0.424	0

TABLE 10: Clusters of graph set containing 19 775 768 graphs

4) 40-vertex snarks: Set of snarks with 40 vertices contains only 25 graphs, which were divided into three clusters. Compared to the other sets of graphs, this number is above average. Measured values are presented in Table 11.

P^{BEST}	Computing time [hh:mm:ss]	Memory needed [GB]	Colored >1 ms
1	00:00:04	0.230	9
2	00:00:01	0.089	2
3	00:00:01	0.019	0

TABLE 11: Clusters of graph set containing 25 graphs

5.3. Evaluation of Experiments Using Parallel Coloring of Sets of Graphs

Overall computation time of coloring of chosen set of graphs o (denoted by To), with the effective decomposition of the problem in mind, is computed as follows:

$$T_o = \sum_{i=1}^{k} Tc_i \tag{4}$$

where k is the number of clusters and Tc_i is the computing time for cluster number i.

We measured following results for 34-vertex snarks. Overall coloring time of the set of 19 935 graphs was 33 seconds. Overall coloring time of

the set of 3 833 587 graphs was 01:14:54 and for the set of 25 286 953 snarks, it was 04:32:35.

36-vertex snarks contain four subsets (one of these subsets consists of over 404 million graphs – a set with the highest number of graphs). First subset contains 180 612 snarks and its overall coloring time was 00:01:25. Subset of graphs with girth ≥ 6 consists of one graph, which was colored in the time under one second. Other subsets of 36-vertex snarks contain more than 60 and more than 404 million snarks. Overall computational time of coloring of set of 60 167 732 snarks was 09:01:15. For the set of 404 899 916 snarks it was 53:18:50.

There are 3 subsets of graphs in the set of 38-vertex snarks. First subset contains 35 429 elements and presented algorithm colored it in 53 seconds. Second subset of 38-vertex snarks consists of 39 graphs, which were colored in 3 seconds. The largest subset of 38-vertex snarks contains 19 775 768 graphs and overall time of its coloring was 03:17:39.

In the case of 40-vertex snarks, there is only one set of 25 graphs, which was divided into three clusters (comparatively high number) in the computational time of 6 seconds.

Table 12 presents number of computed graphs colorings per second for all used subsets of snarks.

Number of vertices	Number of graphs	Number of colored graphs per second
34	19 935	604.04
34	3 833 587	853.05
34	25 286 953	1 546.13
36	180 612	2 124.85
36	60 167 732	1 852.74
36	404 899 916	2 109.62
38	35 429	668.47
38	39	13
38	19 775 768	1 667.57
40	25	4.17

TABLE 12: Number of computed colorings of graphs per second

Therefore, average number of computed colorings of graphs per second is 1 144.36 - considerable improvement over sequential algorithm, which computes on average 189.5 colorings per second.

Parallel speedup of our algorithm for 34-vertex snarks is presented in Table 8. Average speedup for this set of graphs is 32.01.

Number of graphs	Sequential time [hh:mm:ss]	Speedup
19 935	00:03:22	6.12x
3 833 587	34:47:53	31.18x
25 286 953	266:48:49	58.73x

TABLE 13: Parallel speedup computed for the group of 34-vertex snarks

Parallel speedup for bigger sets of graphs (e.g. two sets of 36-vertex graphs with over 60 and over 404 million graphs) are not computed yet.

6. Conclusion

Main objective of this paper was to show that with the use of correct tools and parallel computing, we are able to compute edge coloring of graphs more effectively from the point of view of time demands and from the point of view of correct decomposition of problem.

We presented experimentally tested methodology and algorithm based on data decomposition and clustering of graphs.

Even though presented algorithm needs high amount of memory for its run, it enables us to speed up the computation of coloring of sets of graphs via parallel computation and selection of fitting permutations. The algorithm also creates clusters of graphs in such a way, that each graph from the input set is colored in the comparable (similar) time. This is important concept of effective decomposition of problem. Although it is possible to encounter a graph, which cannot be colored in time lower than the set limit (in our case one millisecond), we were not able to find such graph in the chosen datasets.

From Tables 2 - 11 it's apparent that there is no need for high number of correctly chosen permutations in order to color each graph from the set of graphs in time lower than one millisecond. It is obvious that different set of permutations would produce different results, therefore it is important to approach the problem heuristically and find fitting permutations experimentally.

In the future, it would be beneficial to test conventional clustering methods and divide group of graphs into subgroups which would require same permutation to minimize time of coloring of each graph in the subgroup. This approach, if possible, could save even more time and resources for computation.

Acknowledgments

Computing was performed in the High-Performance Computing Center of the Matej Bel University in Banská Bystrica using the HPC infrastructure acquired in project ITMS 26230120002 and 26210120002 (Slovak infrastructure for high-performance computing) supported by the Research & Development Operational Programme funded by the ERDF.

The research was partially supported by the grant of The Ministry of Education, Science, Research and Sport of the Slovak Republic, VEGA 1/0487/17.

REFERENCES

[1] [1] I. Holyer, “The NP-Completeness of Edge-Colouring,” in SIAM J. COMPUT, vol. 10, Issue 4, 1981, pp. 718-720.
[2] A. Dudáš, P. Voštinár, J. Škrinárová, J. Siláči. (2018) “Improved process of running tasks in the high performance computing system“. Proceedings of 16th IEEE International Conference on Emerging eLearning Technologies and Applications. New Jersey: Institute of Electrical and Electronics Engineers. ISBN 978-1-5386-7912-8. pp. 133-140
[3] N. Robertson, D. Sanders, P. Seymour, R. Thomas. “The Four-Colour Theorem”. Journal of Combinatorial Theory, Series B, Volume 70, Issue 1, 1997, Pages 2-44, ISSN 0095-8956
[4] K. Appel, W. Haken, “Every Planar Map Is Four Colorable”, A.M.S. Contemp. Math. 98 (1989).
[5] K. Appel, W. Haken, “Every planar map is four colorable. Part I”. Discharging, Illinois J. Math. 21 (1977), 429-490.
[6] K. Appel, W. Haken, J. Koch, “Every planar map is four colorable. Part II.” Reducibility, Illinois J. Math. 21 (1977), 491-567.
[7] L. Kowalik, “Improved Edge Coloring with Three Colors”, Theoretical Computer Science, Vol. 410, No. 38-40, 2009, 3733-3742.
[8] M.A. Fiol, J. Vilaltella, “A simple and fast heuristic algorithm for edge-coloring of graphs”, arXiv:1210.5176v1 [math.CO]
[9] R. Nedela, J. Karabáš, M. Škoviera, ‘Nullstellensatz and Recognition of Snarks’. 52th Czech-Slovak Conference Grafy, 2017
[10] W. Sun, J.-K. Hao, W. Wang et al., “Memetic search for the equitable coloring problem”, Knowledge-Based Systems (2019) 105000
[11] Q. Wu, Q. Zhou, Y. Jin, J. Hao. “Minimum sum coloring for large graphs with extraction and backward expansion search”. Applied Soft Computing, Volume 62, 2018, Pages 1056-1065, ISSN 1568-4946
[12] R. Diestel, Graph theory. 5th edition. Springer - Verlag, Heidelberg, 2016
[13] G. Brinkmann, K. Coolsaet, J. Goedgebeur, H. Mélot. “House of Graphs: a database of interesting graphs”. Discrete Applied Mathematics, 161:311-314, 2013 (DOI). Available at http://hog.grinvin.org
[14] Vizing, V. G. (1964), "On an estimate of the chromatic class of a p-graph", Diskret. Analiz., 3: 25–30, MR 0180505
[15] G. Brinkmann, J. Goedgebeur, J. Hagglund and K. Markstrom, “Generation and properties of Snarks”, Journal of Combinatorial Theory, Series B, 103(4):468-488, 20137
[16] A. Dudáš, J. Škrinárová, E. Vesel. “Optimization design for parallel coloring of a set of graphs in the High-Performance Computing”. In: Proceedings of 2019 IEEE 15th International Scientific Conference on Informatics. pp 93-99. ISBN 978-1-7281-3178-8.

Adam Dudáš is PhD. student at Department of Computer science of University of Žilina, Slovakia and works at the Faculty of Natural Sciences, Matej Bel University, Slovakia. His research is focused on parallel and distributed computing and decomposition of computations.

Jarmila Škrinárová is associate professor at Department of Computer science of Matej Bel University in Banská Bystrica, Slovakia. Her research is focused on management systems for high performance computing, parallel and distributed systems and decomposition of computations.

Experiments on Rough Sets Clustering with Various Similarity Measures

Szederjesi-Dragomir, Arnold; Găceanu, Radu D.; Pop, Horia F. and Sârbu, Costel

Abstract: ***Pattern recognition is an operation in which humans usually excel but it is not a trivial task for a computer program due to the uncertainty involved in the whole process. Sources of uncertainty include incomplete data, the presence of outliers, and vagueness in class definitions. The theory of rough sets is a mathematical method for handling uncertainty, providing a formal approximation of a crisp set in terms of a pair of sets representing the lower and upper limits of the original set. The aim of this paper is to investigate the influence of several similarity measures in a rough sets clustering context and, in this sense, datasets with overlapping regions are of particular interest. So, besides the variation of the overall classification result, we are also interested in the way the overlapping detection mechanism is affected by the chosen similarity measure. Since the standard datasets do not provide specific information about overlapping regions, we propose an approach to obtain such data for benchmarking purposes. The importance of properly choosing the similarity measure is outlined by the experiments carried out on standard datasets.***

Index Terms: ***clustering, similarity measure, rough sets, software agent***

1. INTRODUCTION

CLUSTERING is the task of dividing a set of data points into groups (or clusters) such that items from the same group are more similar to each other than items from different groups. For a cluster analysis method, the decision to group two instances together is based on a certain similarity measure, and thus the proper selection of this measure is critical.

One of the major issues in real-life data analysis is uncertainty management which involves dealing with clusters of arbitrary shape, outliers, missing data, and ill-defined clusters. In order to deal with the uncertainty and ambiguity from data, the principles of fuzzy sets and rough sets theory were employed [11], [13], [15], [16], [23]. The proper selection of the similarity measure in a fuzzy or rough sets context could have an even greater impact on the outcome of clustering process if the uncertainty regions are also taken into consideration.

Some studies on the similarity measure influence in clustering algorihms have been performed before [1], [18], [20], but, to our knowledge, there are no such studies for fuzzy or rough clustering scenarios, i.e., for uncertainty driven environments. In [18], the authors perform a comprehensive study considering several distance measures and clustering algorithms aiming to help the research community in identifying suitable distance measures and to facilitate a comparison and evaluation of newly proposed similarity measures with traditional ones. In [20], the authors study the influence of similarity measure selection in conjunction with several clustering algorithms on high dimensional sparse data representing web documents. They conclude that, in this context, the Euclidean distance performs the poorest. In [1], the authors perform a comparative evaluation of a variety of similarity measures for categorical datasets observing that some measures are able to have a consistently higher performance than others on specific datasets.

The aim of this paper is to study the way in which the result of a clustering algorithm based on the rough sets theory is influenced by changing the distance measure. We are particularly interested in the way the reported uncertainty governed regions vary when changing the similarity measure. Since the standard datasets on which we conduct our experiments do not provide too much information regarding the overlapping clusters, we propose a methodology for determining such regions in a dataset, and hence the instances reported by the clustering algorithm as belonging to overlapping areas may be validated against the regions identified by the methodology. In this article we extended our paper from [21] by:

- providing experiments on more datasets with possibly larger overlapping regions
- proposing a different evaluation mechanism of overlapping regions reported by the clustering algorithm
- providing a more in-depth view of the instances from overlapping areas.

Manuscript received Feb, 2020.

A. Szederjesi-Dragomir is a PhD student at the Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania (e-mail: arnold@cs.ubbcluj.ro).

R.D. Găceanu (contact person) works at the Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania (e-mail: rgaceanu@cs.ubbcluj.ro).

H.F. Pop works at the Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania (e-mail: hfpop@cs.ubbcluj.ro).

C. Sârbu works at the Faculty of Chemistry and Chemical Engineering, Babeş-Bolyai University, Cluj-Napoca, Romania (e-mail: csarbu@chem.ubbcluj.ro).

The paper is structured as follows: in Section 2 we provide a more detailed description of the problem of uncertainty management together with the importance of properly choosing the similarity measure, Section 3 offers a brief background on rough sets together with an overview on rough sets clustering. Section 4 presents the methodology proposed by us in this paper. It starts with presenting the similarity measure evaluation methodology from a classification accuracy standpoint and then from an overlapping region detection point of view. The process of finding and evaluating the overlapping regions is also described in this section. Section 5 presents the results obtained in our experiments together with a discussion and, finally, Section 6 presents the conclusions and ideas for future work.

2. MOTIVATION

Generally speaking, pattern recognition tasks are designed to perform an approximate match of all possible inputs and this requires some kind of similarity measure. Clustering algorithms are no exception to this situation and the primary purpose of this paper is to investigate the effect of various similarity measures on the result of the clustering process.

The problem is that the datasets vary structurally from one another. For example, classes in some datasets can be linearly separated, while there is no clear separation between classes in other datasets. For this reason, it is important to change the considered similarity measure according to the dataset to which it will be applied. Considering the just mentioned problem, we apply several similarity measures on standard datasets in the attempt to find whether any of these measures could be considered suitable enough for any dataset. Similar studies have been performed before in the literature [1], [18], [20], but, as far as we know, there are no such studies for uncertainty driven environments, i.e., overlapping clusters.

The presence of outliers (which could be caused by noisy data) and the overlapping clusters are some of the very important issues to be taken into account when considering a clustering algorithm. In practical scenarios one should probably expect the data to be noisy and the classes not to be clearly separable which is why we conduct our experiments on the algorithm from [8], where a clustering algorithm that can handle such situations is proposed. By using software agents [22] for parallelizing the clustering process, the approach from [8] is scalable, which is an important aspect when dealing with large volumes of data.

We believe that in addition to evaluating the consistency of a clustering solution from an accuracy point of view, it is also very important to examine how the similarity measure choice affects overlapping regions.

Nonetheless, researching the impact of various similarity measures on the overlapping regions raises an important issue: the fact that the standard datasets do not provide accurate information about these overlapping areas, making it difficult to compare the effects of different distance measures on these regions. Therefore, for benchmarking purposes, we introduce a technique for extracting this type of information from a dataset.

3. ROUGH SETS CLUSTERING

Incomplete data, vagueness in class definitions, and outliers are sources of uncertainty and the theories of fuzzy sets [23] and rough sets [13] are used in such cases.

The rough set theory [13] is a major mathematical instrument for managing uncertainty and it consists of offering a formal approximation of a crisp set in terms of a pair of sets representing the lower and upper bound of the original set.

Given a set of objects U called the universe of discourse, an equivalence relation $R \subseteq U \times U$, and a subset of U denoted by X, in order to approximate X with respect to R we consider the following definitions:

Definition 1: The **lower approximation** of a set X with respect to R is the set of all objects which **certainly** belong to X:

$$\underline{R}X = \bigcup_{x \in U} \{R(x) : R(x) \subseteq X\}$$

Definition 2: The **upper approximation** of a set X with respect to R is the set of all objects which **possibly** belong to X:

$$\overline{R}X = \bigcup_{x \in U} \{R(x) : R(x) \cap X \neq \emptyset\}$$

Definition 3: The **boundary region** of a set X with respect to R is the set of all objects which cannot certainly be classified as either belonging to X or not belonging to X:

$$RB = \overline{R}X - \underline{R}X$$

Definition 4: A **rough set** is a tuple $\langle \underline{R}X, \overline{R}X \rangle$, where $\underline{R}X$ is the **lower approximation** (Definition 1) of the target set X and $\overline{R}X$ is the **upper approximation** (Definition 2) of the target set X.

Let $\underline{R}C_K$ and $\overline{R}C_k$ denote the lower and upper approximations of a cluster C_k, and let $RB = \overline{R}C_k - \underline{R}C_k$ be the boundary region of C_k. The tuple $\langle \underline{R}C_k, \overline{R}C_k \rangle$ is called the rough set associated to C_k with respect to some equivalence relation R (see Definition 4). Then the rough sets clustering problem may be defined as in Definition 5.

Definition 5: **Rough sets clustering** [8] is the process of finding a set

$$RC = \{\langle \underline{R}C_k, \overline{R}C_k \rangle | k = \overline{1, p}\}$$

of subsets of a given set of objects $X = \{x^i | i = \overline{1,n}, 1 \leq p \leq n\}$ such that:

- $\forall k, l = \overline{1,p}, k \neq l : \underline{R}C_k \cap \underline{R}C_l = \emptyset$
- $\forall k, l = \overline{1,p}, k \neq l : |\overline{R}C_k \cap \overline{R}C_l| \geq 0$
- $\bigcup_{k=1}^{n} \overline{R}C_k = X$
- $\forall k = \overline{1,p} : \underline{R}C_k \neq \emptyset$
- $\forall k = \overline{1,p} : \underline{R}C_k \subseteq \overline{R}C_k$
- $\forall k = \overline{1,p}, \forall i, j = \overline{1, |\underline{R}C_k|} : x^i R x^j \wedge x^i, x^j \in C_k$
- $\forall k = \overline{1,p}, \forall i, j = 1, |\overline{R}C_k| : x^i R x^j \wedge |\{x^i, x^j\} \cap C_k| \geq 0$.

In a rough sets clustering approach, every cluster C_k is defined in terms of a lower approximation and an upper approximation. The instances from the lower approximation certainly belong to the cluster C_k with respect to some similarity measure. The instances from the upper approximation possibly belong to C_k, but we cannot be certain about this, according to the considered similarity measure. If an instance belongs to a lower approximation then it does not belong to any other rough set. On the other hand, if an instance is in the boundary region of a cluster C_k then it may belong to several other boundary regions. If an instance belongs to the lower approximation of a cluster then it also belongs to the upper approximation of that cluster.

In the following, we present an overview of the main steps performed by the ABARC (Agent BAsed Rough Clustering) algorithm introduced in [8]. The ABARC algorithm addresses the overlapping clusters problem by modeling clusters using notions from the rough set theory. It successfully identifies outliers and, by using software agents, it is scalable. The algorithm uses the following arguments:

- X — the dataset
- i_{max} — the number of iterations
- λ — the maximum number of times an agent may search for a similar one
- σ_1 — the similarity limit
- δ — the similarity measure.

The main steps of the algorithm are as follows:

1) Initialize data and parameters: $X, i_{max}, \lambda, \sigma_1, \delta$.
2) Let $\mathcal{AG}$ be a set of agents, where each agent is associated with one instance from the input dataset X.
3) Each $agent$ from $\mathcal{AG}$ will *asynchronously* start to search for a $similarAgent$:
   a) Nondeterministically select an agent from $\mathcal{AG}$.
   b) If they are not in the same cluster and if their similarity is less than σ_1 then a $similarAgent$ is found. *GOTO Step 4* with the $similarAgent$.
   c) Decrement λ.
   d) If $\lambda = 0$ then *GOTO Step 5* with no $similarAgent$.
   e) If $\lambda > 0$ then *GOTO Step 3a*.
4) If a $similarAgent$ is found then the $agent$ moves to its cluster.
5) Decrement i_{max}.
6) If $i_{max} > 0$ then *GOTO Step 3*.
7) Stop.

The clustering process described above starts by initializing the input data and parameters. The dataset X is normalized using Min-Max normalization [9] and each instance is associated with one agent. There is one cluster for each agent so, in the beginning, the number of clusters is equal to the number of agents which is equal to the number of instances.

Each agent executes in *parallel* (Step 3) and it searches for a similar one based on the provided similarity measure δ and the similarity limit σ_1. If a $similarAgent$ is found then they are grouped together (Step 4) otherwise the search process continues. After λ failed attempts the search for similar process is aborted for the current agent, leaving the task to a different agent or to another iteration (Step 5). Agents are implemented as lightweight processes in the Elixir programming language, yielding a highly scalable solution.

Since agents are grouped together only if they *certainly* belong to the same cluster (based on the similarity limit, σ_1), the first phase of our approach will probably produce a large number of clusters. A second phase (introduced in [8]) unifies similar clusters producing *rough clusters*. In this phase, the unification process is based on a second similarity limit, σ_2, denoting the level up to which two instances are *possibly* similar.

Even after the second phase there might be a significant number of clusters remaining, but most of them are normally composed of a very small number of entities which are not similar to either of the 'normal' clusters. The instances from these small clusters will be marked as possible *outliers*. Nevertheless, in the third phase of our approach (presented in [8]), we assign them to the closest cluster and we get the final clustering structure. Thus, the algorithm produces a set of clusters such that:

- all instances from the dataset belong to at least one cluster
- given any two clusters their intersection might not be the empty set, i.e., the clusters might overlap; we refer to the instances from the overlapping regions as *rough instances*
- even if they are assigned to the closest cluster, the outliers are clearly marked as such

4. METHODOLOGY

This paper aims primarily at examining the effect of different distance measures on a clustering algorithm's performance and this section is dedicated to explaining the methodology used in this study for determining the impact of similarity measures on a clustering process outcome.

The experiments are conducted on several standard datasets: Iris [4], Seeds [10], Wine [5], Ionosphere [19], Ecoli [12], Glass [7]. For each dataset we evaluate the accuracy of the ABARC algorithm (from [8]), as well as the reported rough instances (instances that are close enough to more than one cluster), by selecting in turn each of the distances that we want to evaluate. The clustering algorithm (Section 3) is implemented in Elixir [3], while the analysis part (Sections 4-A, 4-B and 4-C) is implemented in Ruby [17]. Elixir was chosen because it has a great multi-threaded performance making it one of the best choices for highly-concurrent scenarios. Ruby was chosen because of its elegant syntax and its focus of software development productivity.

In order to have a better insight of the cluster overlapping problem, we have generated charts showing the clusters as per the official documentation of all the considered standard datasets (see Figure 1).

For each dataset we have performed a Principal Component Analysis (using scikit-learn [14]) utilizing the first two principal components for the chart generation. The total variation captured by the first two principal components is, for each dataset, as follows: 97.76% (Iris), 99.98% (Wine), 99.67% (Seeds), 44.46% (Ionosphere), 72.52% (Ecoli), 74.04% (Glass).

As it may be seen from Figure 1, the charts corresponding to the first three datasets are very similar to the reality, as given in the official documentation for each dataset. In all cases the vagueness in cluster boundaries is clear: some of the clusters are very close to each other, others overlap. Also, some instances are far away from the denser cluster regions.

4.1. Similarity Measure Evaluation Based on Accuracy

In order to study the influence of various similarity measures on a rough clustering process, our first approach is to compute the accuracy of the algorithm described in Section 3 for each similarity measure.

In this study, we examine the following similarity measures: Manhattan, Chebyshev, Euclidean and Minkowski [2], [6], [9]. Actually, the *Minkowski* distance between two instances x and y is a generalized form of the other three distances and it is defined by

$$d_{MIN} = \Big(\sum_{i=1}^{n} |x_i - y_i|^p \Big)^{\frac{1}{p}}$$

where $p \in \mathrm{R}$, $p \geq 1$, and $x, y \in \mathrm{R}^n$. For $p = 1$ it corresponds to the *Manhattan* distance, while for $p = 2$ it corresponds to the *Euclidean* distance. The *Squared Euclidean* distance (which is just the square of the Euclidean distance) is also considered in our study. When p reaches infinity, the *Chebyshev* distance is obtained

$$\lim_{n \to \infty} \Big(\sum_{i=1}^{n} |x_i - y_i|^p \Big)^{\frac{1}{p}} = \max_{1 \leq i \leq n} |x_i - y_i|$$

We consider several standard datasets and, given a similarity measure, we execute the clustering algorithm several times for each dataset. We collect the accuracy from each execution and we compute the minimum, maximum and average accuracy. This process is repeated for each similarity measure.

4.2. Identifying Rough Instances for Benchmarking Purposes

The evaluation process from Section 4-A uses the accuracy to compare similarity measures in a rough clustering context. However, a more suitable approach when dealing with hybrid data would be to compare the rough instances produced by the algorithm using a certain similarity measure with the *actual* or *true* rough instances. By rough instances we mean instances that are close enough to more than one cluster (instances from overlapping regions may fall in this category). By hybrid data we understand any instance for which the membership to a cluster is arguable (this includes rough instances and outliers — instances that are far away from any cluster).

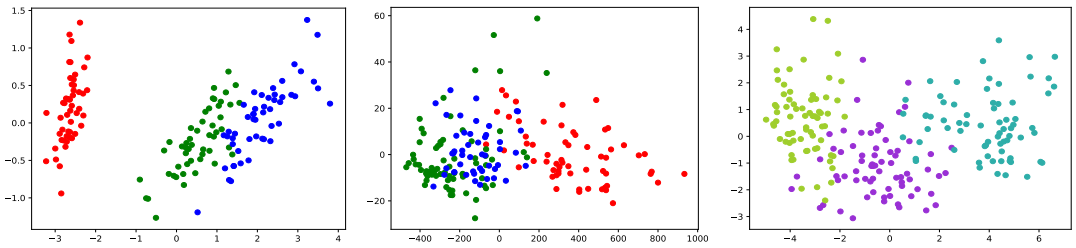
The accuracy of a clustering algorithm is computed based on the *actual* or *true* cluster structure given in the official dataset documentation. In the same fashion, we would like to find some index (like accuracy) that measures the roughness of an approach, i.e., an indicator for the quality of the reported rough instances. Such an index should, of course, not be biased and one possible way in achieving this would be to tie its calculation to the *actual* or *true* rough instances in a similar way the accuracy does.

Unfortunately, the first problem in achieving the aforementioned goal is the fact that the official documentation of any dataset (as far as we know) does not specify the actual rough instances. In practical scenarios, the hybrid instances could be validated by a domain expert, but there should be other possibilities as well, at least for benchmarking purposes. This is why our first intent is to propose an objective approach for finding the *actual* or *true* rough instances from a dataset and, secondly, to introduce a first attempt for a roughness index that takes into account this information.

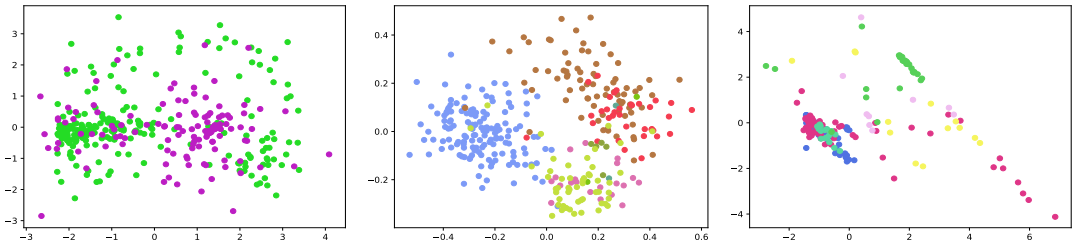
Algorithm 1 shows the proposed process of finding the actual rough instances.

The algorithm receives as a first argument the set of clusters $C = \{C_k | k = \overline{1, p}\}$, as specified in the official dataset documentation, where p represents the number of clusters and C_k are subsets composed of elements from the dataset

Fig. 1: Overlapping clusters in the considered datasets.



(a) Iris. (b) Wine. (c) Seeds.



(d) Ionosphere. (e) Ecoli. (f) Glass.

$X = \{x^i | i = \overline{1,n}, 1 \leq p \leq n\}$. The second argument, τ, denotes the *rough threshold* and it is used for deciding if an instance should be considered a rough one. The last argument, δ, represents the distance function used in the analysis process. The algorithm returns the set RI of instances that will be considered truly rough.

Algorithm 1: Find Rough Instances

Data: C, τ, δ
Result: RI — the set of actual rough instances

```
1  RI = ∅
2  while true do
3  |   ID = computeMeanDistances(C, δ)
4  |   RI_1 = computeRoughInstances(ID, τ)
5  |   if RI_1 = ∅ then
6  |   |   break
7  |   end
8  |   C = C ⊖ RI_1
9  |   RI = RI ∪ RI_1
10 end
```

The algorithm starts by computing a mapping, ID, between each instance and the mean distance corresponding to each cluster

$$ID = X \xrightarrow{m} M$$

where the $\xrightarrow{m}$ symbol denotes a mapping between each element of X and a set

$$M = \{m_k | k = \overline{1,p}\}$$

of mean distances.

Given an instance $x^i \in X$, the mean distance corresponding to cluster C_k is

$$m_k = \frac{\sum_{j=1}^{|C_k|} \delta(x^i, x^j)}{|C_k|}, \; x^j \in C_k, \; j \neq i$$

In line 4 from Algorithm 1, the set RI_1 of rough instances is computed by selecting all instances from ID having the following property

$$\exists \; i, j \ni |m_i - m_j| \leq \tau$$

where $m_i, m_j \in M$ represent, respectively, the mean distances of clusters C_i and C_j for a given instance, and τ is the rough threshold.

The instances from RI_1 are removed from the set of clusters, as indicated by the $\ominus$ operator, and they are added into the result set, RI.

The whole process is repeated until the set RI_1 is empty.

4.3. Similarity Measure Evaluation Based on the Rough Instances

In order to evaluate the influence of a certain similarity measure on the rough instances, we compare the rough instances reported by the algorithm described in Section 3 with the rough instances proposed by the Algorithm 1 from Section 4-B.

Given a similarity measure, we execute the algorithm several times (N) on a certain dataset and we collect the reported rough instances from each execution.

For each rough instance x^i we compute the *occurrence rate*

$$occ_{x^i} = \frac{n}{N}$$

where n denotes the number of occurrences of the instance in the series of N executions.

For a given distance d, we compute the *rough score* as shown in Definition 6.

Definition 6: The **rough score** of a given similarity measure is the ratio between the sum of the *occurrence rates* of the validated rough instances and the sum of all *occurrence rates*:

$$\mathcal{RSC}_d^\kappa = \frac{\sum_{i=1}^{|RI_d|} 1_{RI^\kappa}(x^i) \cdot occ_{x^i}}{|RI_d^\kappa|}$$

where:

- RI_d is the set of rough instances reported by the clustering algorithm for a certain similarity measure (or distance, d)
- RI^κ is the set of actual rough instances produced by the Algorithm 1 from Section 4-B for which $occ_{x^i} \geq \kappa$, where $\kappa \in [0, 1]$ represents the minimum confidence of the rough instance x_i
- 1_{RI^κ} is the indicator function of RI^κ:

$$1_{RI^\kappa}(x^i) = \begin{cases} 1, & \text{if } x^i \in RI^\kappa \wedge occ_{x^i} \geq \kappa \\ 0, & \text{otherwise.} \end{cases}$$

- RI_d^κ is the set of rough instances reported by the clustering algorithm for a distance, d, and minimum confidence, κ.

As seen in Definition 6, the *rough-score* depends on a fixed parameter κ denoting the confidence of each rough instance x^i — so only the instances having an *occurrence rate* higher than κ are taken into account. Given a similarity measure, we compute the rough score for several values of κ.

5. Results and discussion

This section presents the experimental setup of our study and discusses the obtained results. The experiments are conducted on several standard datasets: Iris [4], Seeds [10], Wine [5], Ionosphere [19], Ecoli [12], Glass [7].

The Iris dataset [4], one of the most widely used datasets in pattern recognition, contains 150 instances with four attributes denoting geometric information regarding three species of Iris plants. There are three classes with 50 instances each, one of the classes being linearly separable from the other two. The data is scaled in the $[0, 1]$ range using Min-Max normalization [9] and the algorithm described in Section 3 is applied with the following parameter setting: $i_{max} = 100$, $\lambda = 100$, $\sigma_1 = 0.0115$, and $\sigma_2 = 0.1$. For each of the considered distances, the clustering algorithm is executed 50 times computing the accuracy and rough score as described in Section 4. Table 1 shows the minimal, maximal and average accuracy for each similarity measure.

The Seeds dataset [10] contains 210 instances with seven attributes describing geometric parameters of wheat kernels. There are three classes of 70 instances each. After normalization (as described before), the algorithm from Section 3 is applied with the following parameter setting: $i_{max} = 100$, $\lambda = 100$, $\sigma_1 = 0.024$, and $\sigma_2 = 0.2$. The clustering algorithm is executed 50 times for each of the considered similarity measures and a result summary in terms of accuracy is presented in Table 1.

The Wine dataset [5] contains 178 instances with 13 attributes describing information of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The data is normalized in the same fashion as for the other datasets and the algorithm described in Section 3 is applied 50 times with the following parameter setting: $i_{max} = 100$, $\lambda = 100$, $\sigma_1 = 0.145$, and $\sigma_2 = 0.45$. Tables 1 shows the minimal, maximal and average accuracy for the Wine dataset.

The Ionosphere dataset [19] contains 351 instances with 34 attributes and two classes. After normalizing the data, the algorithm from Section 3 is applied 50 times with the following parameter setting: $i_{max} = 100$, $\lambda = 100$, $\sigma_1 = 0.1583$, and $\sigma_2 = 1.04$ and the results in terms of accuracy are shown in Table 2.

The Ecoli dataset [12] contains 336 instances with 8 attributes and 8 classes out of which three classes contain few instances. After normalizing the data, the algorithm from Section 3 is applied 50 times with the following parameter setting: $i_{max} = 100$, $\lambda = 100$, $\sigma_1 = 0.02$, and $\sigma_2 = 0.16$ and the results in terms of accuracy are shown in Table 2.

The Glass dataset [7] contains 214 instances with 10 attributes and 7 classes. After normalizing the data, the algorithm from Section 3 is applied 50 times with the following parameter setting: $i_{max} = 100$, $\lambda = 100$, $\sigma_1 = 0.1112$, and $\sigma_2 = 0.586$ and the results in terms of accuracy are shown in Table 2.

The rough instances together with their occurrence rates for each similarity measure are available in our previous work from [21] and are excluded from this paper for brevity reasons.

The analysis process described in Sections 4-B and 4-C is then applied for each dataset obtaining the rough scores displayed in Table 3. A NaN value in Table 3 indicates that no rough instances have been reported by the clustering algorithm in that case.

Considering Table 1, the best results on the average are for the Minkowski distance where $p = 2.3$ and, in our previous work from [21], we observed approximately the same situation for the rough scores. Moreover, the following order relations between the scores (accuracy, rough) of different Minkowsky distances (expressed as

TABLE 1: Accuracies for the Iris, Seeds and Wine datasets.

Similarity measure	Acc Iris (%)			Acc Seeds (%)			Acc Wine (%)		
	Min	***Avg***	***Max***	***Min***	***Avg***	***Max***	***Min***	***Avg***	***Max***
Manhattan	86.0	95.69	96.67	61.43	86.17	87.62	78.65	88.33	91.01
Chebyshev	83.33	84.07	85.33	83.33	90.36	90.95	**89.89**	91.52	92.7
Euclidean	91.33	96.49	96.67	79.52	91.1	91.43	72.47	96.81	**98.31**
Squared Euclidean	88.0	96.36	96.67	**90.95**	90.98	91.43	72.47	96.57	97.75
Minkowski $p = \sqrt{2}$	85.33	96.39	**97.33**	89.05	91.32	91.43	82.02	92.07	92.7
Minkowski $p = 2.3$	**96.0**	**96.65**	96.67	**90.95**	**91.39**	91.43	84.83	**97.48**	97.75
Minkowski $p = 2\sqrt{2}$	**96.0**	**96.65**	96.67	89.52	90.97	**91.9**	84.83	95.34	96.63
Minkowski $p = 3$	94.0	96.6	96.67	73.81	90.29	**91.9**	78.65	92.62	95.51

TABLE 2: Accuracies for the Ionosphere, Ecoli and Glass datasets.

Similarity measure	Acc Ionosphere (%)			Acc Ecoli (%)			Acc Glass (%)		
	Min	***Avg***	***Max***	***Min***	***Avg***	***Max***	***Min***	***Avg***	***Max***
Manhattan	60.29	65.54	66.57	63.99	65.51	70.24	86.92	88.69	90.19
Chebyshev	58.29	58.29	62.29	45.54	47.77	50.89	0.0	75.42	89.72
Euclidean	69.71	**69.77**	70.29	59.82	64.11	71.13	88.32	**89.72**	90.65
Squared Euclidean	62.0	68.46	70.57	59.23	63.63	67.86	88.32	89.58	90.65
Minkowski $p = \sqrt{2}$	46.0	63.94	68.86	61.01	**66.16**	72.92	87.38	88.41	89.25
Minkowski $p = 2.3$	62.57	69.6	70.57	61.01	64.64	71.43	88.32	88.88	90.19
Minkowski $p = 2\sqrt{2}$	51.43	65.86	70.0	63.39	65.09	67.26	0.0	71.59	90.19
Minkowski $p = 3$	51.14	64.74	70.86	59.82	59.82	72.62	86.92	87.99	89.72

values of p) appeared to emerge

$$(p = 1) \leq (p = \sqrt{2}) \leq (p = 2) \leq (p = 2.3)$$
$$(p = 2.3) \geq (p = 2\sqrt{2}) \geq (p = 3) \geq (p = \infty)$$

where $(p = n)$ is the score (accuracy or rough) of the Minkowsky distance for $p = n$.

One of the goals of the current work was to check to what degree do these relations hold for other datasets and in doing so we have tried to choose datasets with many hybrid instances. As we can see in Table 2, the aforementioned relations do not hold, but still, for $p = 2.3$ we obtain results very close to the best one in all cases. In what concerns the rough scores (Table 3), it seems that the best results (high rough scores for high confidence values) are obtained in most cases for values of p close to 2.3. For example, for Wine and Seeds the best results are obtained for $p = \sqrt{2}$, while for the other datasets $p = 2\sqrt{2}$ gives the best results, but these values of p are both close to 2.3.

Even though the results from our previous work [21] suggested that the Minkowsky distance with $p = 2.3$ is the best one, our current study shows, as expected, that this is not the case, but still, the best result is in general obtained for values of p close to 2.3. This is probably caused by the fact that the clusters from most datasets have a similar spherical shape. The scores tend to decrease for $p > 2.3$ probably because the value of the Minkowsky distance decreases with increasing values of p and hence more instances are accepted as similar including possible outliers. Moreover, with increasing values of p the sphere like shape of clusters expands, becoming a square for $p = \infty$. Similarly, for decreasing values of p the spherical shapes shrink accepting fewer and fewer instances, which could explain why the scores tend to degrade for smaller values of p.

6. Conclusions and future work

An important challenge in classification problems is the proper selection of the similarity measure because the outcome is greatly influenced by this decision. A widely used similarity measure is the Euclidean distance and we wanted to know whether this is the best choice when dealing with hybrid data, with situations where the cluster boundaries are not clearly defined. In this regard, we have tried to select some standard datasets having clusters with large overlapping regions and we have studied the influence of each considered similarity measure from two perspectives: the influence on the classification accuracy and the influence on the hybrid data detection (particularly, rough instances). Our experiments indicate that the best results are obtained in general for the Minkowsky distance with values of p close to 2.3, but, as discussed in the paper, this might be caused by the fact that most clusters have similar spherical shapes.

A major issue in pattern recognition is outlier management, which could have a major impact on the result if handled improperly. For example, in a regression analysis context, even a single outlier could greatly influence the outcome. Our algorithm is able to detect outliers and outputs the result with and without outliers. We would like to direct our future efforts to outlier management first, by validating our results and second, by investigating the way outliers are influenced by the similarity measure selection.

TABLE 3: Rough scores with minimum confidences (κ) ranging between 0 and 1 with a step of 0.1.

	Similarity measure	***0***	***0.1***	***0.2***	***0.3***	***0.4***	***0.5***	***0.6***	***0.7***	***0.8***	***0.9***	***1***
Iris	Manhattan	2.61	11.0	25.5	25.5	25.5	20.0	20.0	NaN	NaN	NaN	NaN
	Chebyshev	6.36	9.5	13.13	14.31	19.5	11.0	11.0	0.0	0.0	0.0	0.0
	Euclidean	5.9	38.5	38.5	38.5	38.5	51.33	51.33	45.0	45.0	45.0	0.0
	Squared Euclidean	14.88	15.25	18.39	18.39	59.33	89.0	89.0	89.0	89.0	98.0	NaN
	Minkowski $p = \sqrt{2}$	5.56	36.0	36.0	36.0	36.0	48.0	46.0	46.0	46.0	46.0	0.0
	Minkowski $p = 2.3$	15.71	18.17	27.25	43.6	43.6	54.5	80.0	80.0	86.0	NaN	NaN
	Minkowski $p = 2\sqrt{2}$	7.61	18.88	25.17	31.71	62.0	62.0	62.0	62.0	93.0	98.0	NaN
	Minkowski $p = 3$	6.12	16.0	17.33	29.71	59.33	59.33	59.33	89.0	89.0	96.0	NaN
Wine	Manhattan	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	Chebyshev	7.73	11.94	20.24	20.0	20.29	33.33	33.33	33.33	50.0	100.0	100.0
	Euclidean	5.36	8.0	10.47	15.33	23.0	0.0	0.0	0.0	0.0	NaN	NaN
	Squared Euclidean	3.65	6.83	10.91	6.4	0.0	0.0	0.0	NaN	NaN	NaN	NaN
	Minkowski $p = \sqrt{2}$	5.72	8.66	10.63	11.47	14.0	16.55	10.57	12.33	0.0	NaN	NaN
	Minkowski $p = 2.3$	4.99	8.87	9.39	10.67	0.0	0.0	0.0	0.0	0.0	NaN	NaN
	Minkowski $p = 2\sqrt{2}$	4.03	6.33	8.29	8.5	0.0	0.0	0.0	0.0	NaN	NaN	NaN
	Minkowski $p = 3$	5.15	7.04	10.74	23.33	16.0	0.0	0.0	0.0	0.0	NaN	NaN
Seeds	Manhattan	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	Chebyshev	1.74	2.51	7.4	4.75	0.0	0.0	0.0	0.0	NaN	NaN	NaN
	Euclidean	3.62	4.87	10.34	17.5	18.62	19.0	15.0	NaN	NaN	NaN	NaN
	Squared Euclidean	4.49	6.94	8.4	8.64	27.56	41.6	52.0	86.0	86.0	NaN	NaN
	Minkowski $p = \sqrt{2}$	4.97	7.8	9.35	13.52	17.45	14.86	22.29	52.0	40.0	0.0	0.0
	Minkowski $p = 2.3$	3.58	6.07	11.64	21.62	25.08	37.6	33.5	35.0	NaN	NaN	NaN
	Minkowski $p = 2\sqrt{2}$	4.09	7.01	11.74	20.46	24.11	29.56	41.0	41.0	31.33	31.33	0.0
	Minkowski $p = 3$	3.41	4.73	7.04	8.38	18.92	7.78	14.0	14.0	0.0	0.0	NaN
Ionosphere	Manhattan	2.0	1.27	0.0	0.0	0.0	0.0	0.0	0.0	0.0	NaN	NaN
	Chebyshev	14.5	19.33	19.33	19.33	29.0	29.0	NaN	NaN	NaN	NaN	NaN
	Euclidean	2.65	6.45	0.0	0.0	0.0	0.0	0.0	0.0	0.0	NaN	NaN
	Squared Euclidean	1.85	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	NaN	NaN
	Minkowski $p = \sqrt{2}$	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	NaN	NaN	NaN
	Minkowski $p = 2.3$	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	NaN	NaN
	Minkowski $p = 2\sqrt{2}$	2.29	37.0	37.0	37.0	74.0	74.0	74.0	74.0	NaN	NaN	NaN
	Minkowski $p = 3$	5.0	8.24	0.0	0.0	0.0	0.0	0.0	0.0	NaN	NaN	NaN
Ecoli	Manhattan	3.05	7.04	6.62	0.0	0.0	0.0	0.0	0.0	0.0	0.0	NaN
	Chebyshev	4.64	22.93	47.0	52.0	56.0	56.0	56.0	48.0	48.0	48.0	NaN
	Euclidean	2.88	8.98	11.46	13.45	11.14	15.6	26.0	78.0	NaN	NaN	NaN
	Squared Euclidean	3.96	10.48	14.16	18.22	27.2	56.0	NaN	NaN	NaN	NaN	NaN
	Minkowski $p = \sqrt{2}$	3.23	8.57	13.83	28.4	47.33	47.33	47.33	37.0	NaN	NaN	NaN
	Minkowski $p = 2.3$	4.61	12.11	16.35	17.87	19.0	22.4	0.0	NaN	NaN	NaN	NaN
	Minkowski $p = 2\sqrt{2}$	3.31	9.31	13.43	19.5	26.0	39.0	78.0	78.0	NaN	NaN	NaN
	Minkowski $p = 3$	2.23	9.5	10.36	10.29	24.0	24.0	24.0	36.0	NaN	NaN	NaN
Glass	Manhattan	10.0	34.0	34.0	34.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	Chebyshev	8.0	16.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	Euclidean	6.44	34.0	34.0	34.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	Squared Euclidean	9.2	34.0	34.0	34.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	Minkowski $p = \sqrt{2}$	8.53	25.5	25.5	36.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	Minkowski $p = 2.3$	11.67	17.0	34.0	34.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	Minkowski $p = 2\sqrt{2}$	50.0	50.0	50.0	80.0	80.0	80.0	80.0	80.0	80.0	NaN	NaN
	Minkowski $p = 3$	35.0	35.0	52.0	52.0	72.0	72.0	72.0	72.0	NaN	NaN	NaN

REFERENCES

[1] Shyam Boriah, Varun Chandola, and Vipin Kumar. Similarity measures for categorical data: A comparative evaluation. In *SDM*, 2008.

[2] Sung-Hyuk Cha. Comprehensive survey on distance/similarity measures between probability density functions, 2007.

[3] Programming Language Elixir. Elixir, 2019.

[4] Ronald A. Fisher. *UCI Machine Learning Repository: Iris Data Set*. http://archive.ics.uci.edu/ml/datasets/Iris, 1936.

[5] M Forina. *UCI Machine Learning Repository: Wine Data Set*. https://archive.ics.uci.edu/ml/datasets/wine, 1991.

[6] Guojun Gan, Chaoqun Ma, and Jianhong Wu. *Data Clustering: Theory, Algorithms, and Applications (ASA-SIAM Series on Statistics and Applied Probability)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2007.

[7] B. German. *UCI Machine Learning Repository: Glass Data Set*. https://archive.ics.uci.edu/ml/datasets/glass + identification, 1987.

[8] Radu D. Găceanu, Arnold Szederjesi-Dragomir, Horia F. Pop, and Costel Sârbu. ABARC: An agent-based rough sets clustering algorithm. Submitted, 2019.

[9] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[10] Piotr Kulczycki. *UCI Machine Learning Repository: Seeds Data Set*. https://archive.ics.uci.edu/ml/datasets/seeds, 2012.

[11] J. Li and H. W. Lewis. Fuzzy clustering algorithms — review of the applications. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 282–288, Nov 2016.

[12] Kenta Nakai. *UCI Machine Learning Repository: Ecoli Data Set*. https://archive.ics.uci.edu/ml/datasets/ecoli, 1996.

[13] Zdzislaw Pawlak. *Rough Sets: Theoretical Aspects of Reasoning About Data*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-

learn: Machine Learning in Python . *Journal of Machine Learning Research*, 12:2825–2830, 2011.
[15] D. Perdukova, P. Fedor, and V. Fedák. A fuzzy approach to optimal dc motor controller design. In *2019 International Conference on Electrical Drives Power Electronics (EDPE)*, pages 48–53, Sep. 2019.
[16] Nina Radojičić. An approach to solving the min-max diversity problem using genetic algorithm with fuzzy decisions. *IPSI BgD Transactions on Internet Research (TIR)*, 13(1), 2017.
[17] Programming Language Ruby. Ruby, 2019.
[18] Ali Seyed Shirkhorshidi, Saeed Reza Aghabozorgi, Teh Ying Wah, and Andrew R. Dalby. A comparison study on similarity and dissimilarity measures in clustering continuous data. In *PloS one*, 2015.
[19] Vince Sigillito. *UCI Machine Learning Repository: Ionosphere Data Set*. https://archive.ics.uci.edu/ml/datasets/ionosphere, 1989.
[20] Alexander Strehl, Er Strehl, Joydeep Ghosh, and Raymond Mooney. Impact of similarity measures on web-page clustering. In *In Workshop on Artificial Intelligence for Web Search (AAAI 2000)*, pages 58–64. AAAI, 2000.
[21] Arnold Szederjesi-Dragomir, Radu D. Găceanu, Horia F. Pop, and Costel Sârbu. A comparison study of similarity measures in rough sets clustering. In *Proceedings of the 2019 IEEE 15th International Scientific Conference on Informatics (INFORMATICS 2019). Editors: William Steingartner, Štefan Korečko, Anikó Szakál*, pages 399 – 405. IEEE Press, Poprad, Slovakia, Nov 20 – 22, 2019.
[22] Michael Wooldridge. *An Introduction to MultiAgent Systems*. Wiley Publishing, 2nd edition, 2009.
[23] L.A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338 – 353, 1965.

Arnold Szederjesi-Dragomir is a PhD student since 2017 at the Department of Computer Science, Faculty of Mathematics and Computer Science from the Babes-Bolyai University of Cluj-Napoca, Romania. His main research interests are pattern recognition and agent-based computing.

Radu Dan Găceanu is a lecturer at the Department of Computer Science, Faculty of Mathematics and Computer Science from the Babes-Bolyai University of Cluj-Napoca, Romania. He has received the PhD degree in Computer Science in 2012, with the “magna cum laude” distinction. His main research interests are pattern recognition and soft computing.

Horia Florin Pop is a professor at the Department of Computer Science, Faculty of Mathematics and Computer Science from the Babes-Bolyai University of Cluj-Napoca, Romania. He published more than 125 papers in prestigious journals and conference proceedings. His research interests include soft computing and pattern recognition.

Costel Sârbu is a professor at the Department of Analytical Chemistry, Faculty of Chemistry and Chemical Engineering from the Babes-Bolyai University of Cluj-Napoca, Romania. As per Google Scholar, he has over 2600 citations and an h-index of 26. His research interests include chemometrics and soft computing.

Reviewers:

Australia
Abramov, Vyacheslav; Monash University
Begg, Rezaul; Victoria University
Bem, Derek; University of Western Sydney
Betts, Christopher; Pegacat Computing Pty. Ltd.
Buyya, Rajkumar; The University of Melbourne
Chapman, Judith; Australian University Limited
Chen, Yi-Ping Phoebe; Deakin University
Hammond, Mark; Flinders University
Henman, Paul; University of Queensland
Palmisano, Stephen; University of Wollongong
Ristic, Branko; Science and Technology Organisation
Sajjanhar, Atul; Deakin University
Sidhu, Amandeep; University of Technology, Sydney
Sudweeks, Fay; Murdoch University

Austria
Derntl, Michael; University of Vienna
Hug, Theo; University of Innsbruck
Loidl, Susanne; Johannes Kepler University
Linz Stockinger, Heinz; University of Vienna
Sutter, Matthias; University of Innsbruck

Brazil
Parracho, Annibal; Universidade Federal
Fluminense Traina, Agma; University of Sao Paulo
Traina, Caetano; University of Sao Paulo
Vicari, Rosa; Federal University of Rio Grande

Belgium
Huang, Ping; European Commission

Canada
Fung, Benjamin; Simon Fraser University
Grayson, Paul; York University Gray,
Bette; Alberta Education
Memmi, Daniel; UQAM
Neti, Sangeeta; University of Victoria
Nickull, Duane; Adobe Systems, Inc. Ollivier-Gooch,
Carl; The University of British Columbia Paulin,
Michele; Concordia University
Plaisent, Michel; University of Quebec Reid, Keith;
Ontario Ministry og Agriculture Shewchenko,
Nicholas; Biokinetics and Associates Steffan,
Gregory; University of Toronto Vandenberghe,
Christian; HEC Montreal

Czech Republic
Kala, Zdenek; Brno University of Technology
Korab, Vojtech; Brno University of technology
Lhotska, Lenka; Czech Technical University

Finland
Lahdelma, Risto; University of Turku
Salminen, Pekka; University of Jyvaskyla

France
Cardey, Sylviane; University of Franche-Comte
Klinger, Evelyne; LTCI – ENST, Paris
Roche, Christophe; University of Savoie
Valette, Robert; LAAS - CNRS

Germany
Accorsi, Rafael; University of Freiburg
Glatzer, Wolfgang; Goethe-University
Gradmann, Stefan; Universitat Hamburg
Groll, Andre; University of Siegen
Klamma, Ralf; RWTH Aachen University
Wurtz, Rolf P.; Ruhr-Universitat Bochum

Hungary
Mester, Gyula; Óbuda University, Budapest

India
Pareek, Deepak; Technology4Development
Scaria, Vinod; Institute of Integrative Biology
Shah, Mugdha; Mansukhlal Svayam

Ireland
Eisenberg, Jacob; University College Dublin

Israel
Feintuch, Uri; Hadassah-Hebrew University

Italy
Badia, Leonardo; IMT Institute for Advanced
Studies Berrittella, Maria; University of Palermo
Carpaneto, Enrico; Politecnico di Torino

Japan
Hattori, Yasunao; Shimane University
Livingston, Paisley; Linghan University
Srinivas, Hari; Global Development Research Center
Obayashi, Shigeru; Institute of Fluid Science,
Tohoku University

Netherlands
Mills, Melinda C.; University of Groningen
Pires, Luís Ferreira; University of Twente

New Zealand
Anderson, Tim; Van Der Veer Institute

Portugal
Cardoso, Jorge; University of Madeira
Natividade, Eduardo; Polytechnic Institute of
Coimbra Oliveira, Eugenio; University of Porto

Singapore
Tan, Fock-Lai; Nanyang Technological University

South Korea
Kwon, Wook Hyun; Seoul National University

Spain
Barrera, Juan Pablo Soto; University of Castilla
Gonzalez, Evelio J.; University of La Laguna
Perez, Juan Mendez; Universidad de La
Laguna Royuela, Vicente; Universidad de
Barcelona Vizcaino, Aurora; University of
Castilla-La Mancha Vilarrasa, Clelia Colombo;
Open University of Catalonia

Sweden
Johansson, Mats; Royal Institute of Technology

Switzerland
Niinimaki, Marko; Helsinki Institute of Physics
Pletka, Roman; AdNovum Informatik AG
Rizzotti, Sven; University of Basel
Specht, Matthias; University of Zurich

Taiwan
Lin, Hsiung Cheng; Chienkuo Technology University
Shyu, Yuh-Huei; Tamkang University
Sue, Chuan-Ching; National Cheng Kung
University

United Kingdom
Ariwa, Ezendu; London Metropolitan University
Biggam, John; Glasgow Caledonian University
Coleman, Shirley; University of Newcastle
Conole, Grainne; University of Southampton
Dorfler, Viktor; Strathclyde University
Engelmann, Dirk; University of London
Eze, Emmanuel; University of Hull
Forrester, John; Stockholm Environment Institute
Jensen, Jens; STFC Rutherford Appleton
Laboratory Kolovos, Dimitrios S.; The University
of York McBurney, Peter; University of Liverpool
Vetta, Atam; Oxford Brookes University
WHYTE, William Stewart; University of Leeds
Xie, Changwen; Wicks and Wilson Limited

USA
Bach, Eric; University of Wisconsin Bolzendahl,
Catherine; University of California Bussler,
Christoph; Cisco Systems, Inc. Charpentier,
Michel; University of New Hampshire Chong,
Stephen; Cornell University
Collison, George; The Concord Consortium
DeWeaver, Eric; University of Wisconsin -
Madison Gans, Eric; University of California
Gill, Sam; San Francisco State University
Hunter, Lynette; University of California Davis
Iceland, John; University of Maryland
Kaplan, Samantha W.; University of Wisconsin
Langou, Julien; The University of Tennessee
Liu, Yuliang; Southern Illinois University
Edwardsville Lok, Benjamin; University of Florida
Minh, Chi Cao; Stanford University
Morrissey, Robert; The University of Chicago
Mui, Lik; Google, Inc
Rizzo, Albert ; University of Southern
California Rosenberg, Jonathan M. ; University
of Maryland Shaffer, Cliff ; Virginia Tech
Sherman, Elaine; Hofstra University
Snyder, David F.; Texas State University
Song, Zhe; University of Iowa
Wei, Chen; Intelligent Automation, Inc.
Yu, Zhiyi; University of California

Welcome to IPSI Conferences and Journals!

http://tir.ipsitransactions.org

http://www.ipsitransactions.org

CIP – Katalogizacija u publikaciji
Narodna biblioteka Srbije, Beograd
ISSN 1820 – 4503
The IPSI Transactions
on Internet Research
COBISS.SR - ID 119127052