### Frontmatter

**Editors**

SIGAPP FY’13 Quarterly Report  
S. Shin  
4

A Message from the Editor  
S. Shin  
5

SAC 2013 Report  
H. Haddad  
6

### Invited Paper

**Buffer Overflow Patching for C and C++ Programs: Rule-Based Approach**  
H. Shahriar, H. Haddad, and I. Vaidya  
8

### Selected Research Articles

**Eliminating Redundancy in CSPs Through Merging and Subsumption of Domain Values**  
C. Likitvivatanavong and R. Yap  
20

**A Framework for Processing Complex Queries in Wireless Sensor Networks**  
J. Maia, A. Brayner, and F. Rodrigues  
30

**An Automatic Blocking Strategy for XML Duplicate Detection**  
L. Leitao and P. Calado  
42

**Data Stream Processing with Concurrency Control**  
M. Oyamada, H. Kawashima, and H. Kitagawa  
54

**RTRS: A Novel Real-Time Reasoning System Based on Active Rules**  
Y. Qiao, Q. Han, H. Wang, X. Li, K. Zhong, K. Zhang, J. Liu, and A. Guo  
66
Editor in Chief  
Sung Y. Shin

Designer/Technical Editor  
John Kim

Associate Editors  
Richard Chbeir
Hisham Haddad
Lorie M. Liebrock
W. Eric Wong

Editorial Board Members

Richard Chbeir  
Junping Sun  
Fernando S. Osorio
Mir Abolfazl Mostafavi  
Ramzi A. Haraty  
Luiz Chaimowicz
Ki-Joune Li  
Apostolos Papadopoulos  
Prasenjit Mitra
Kokou Yetongnon  
Federico Divina  
Lior Rokach
Dan Tulpan  
Raúl Giráldez Rojo  
Yehuda Koren
Paola Lecca  
Artur Caetano  
Antonio Bucchiarone
Mathew Palakal  
Rogério Carvalho  
Raffaela Mirandola
Umesh Bellur  
Maria-Eugenia Iacob  
Patrizia Scandurra
Rajiv Ramnath  
Rafael Accorsi  
W. Eric Wong
S D Madhu Kumar  
Alessio Bechini  
John Kim
Agostinho Rosa  
Cosimo Antonio Prete  
Giampaolo Bella
Yin-Fu Huang  
Gloria Bordogna  
Helge Janicke
Mirko Violi  
Gabriella Pasi  
Alessandro Sorniotti
Gabriella Castelli  
Hong Va Leong  
Somayeh Malakuti
Jose Luis Fernandez  
Alvin Chan  
Wolfgang Lohmann
Rui P. Rocha  
Maria da Graça C. Pimentel  
Mehmet Aksit
Matthew E. Taylor  
Takayuki Itoh  
Gail-Joon Ahn
Rachel Anane  
Rudinei Goulart  
Dongwan Shin
David Parker  
Mario Freire  
Ivan Lanese
Stefano Bistarelli  
Marília Curado  
Manuel Mazzara
Eric Monfroy  
Manuela Pereira  
Fabrizio Montesi
Barry O'Sullivan  
Teresa Vazão  
Jun Pang
Karl M. Goeschka  
Davide Ancona  
Mohammad Reza Mousavi
Rui Oliveira  
Tei-Wei Kuo  
Hyoil Han
Peter Pietzuch  
Seongwon Lee  
Anabela Simões
Giovanni Russello  
Marjan Mermik  
Markus Zanker
Hasan Jamil  
Barrett Bryant  
Jean-Marc Seigneur
Raymond Wong  
Emiliano Tramontana  
Davide Rossi
Pedro Rodrigues  
Corrado Santoro  
Angelo Di Iorio
Albert Bifet  
Yvonne Coady  
Stefano Zacchiroli
Shonali Krishnaswamy  
Maria Lacastre
João Gama  
Denis Wolf
SIGAPP FY’13 Quarterly Report

April 2013 – June 2013
Sung Shin

Mission

To further the interests of the computing professionals engaged in the development of new computing applications and to transfer the capabilities of computing technology to new problem domains.

Officers

<table>
<thead>
<tr>
<th>Role</th>
<th>Name</th>
<th>Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chair</td>
<td>Sung Shin</td>
<td>South Dakota State University, USA</td>
</tr>
<tr>
<td>Vice Chair</td>
<td>Richard Chbeir</td>
<td>Bourgogne University, Dijon, France</td>
</tr>
<tr>
<td>Secretary</td>
<td>W. Eric Wong</td>
<td>University of Texas at Dallas, USA</td>
</tr>
<tr>
<td>Treasurer</td>
<td>Lorie M. Liebrock</td>
<td>New Mexico Institute of Mining and Technology, USA</td>
</tr>
<tr>
<td>Web Master</td>
<td>Hisham Haddad</td>
<td>Kennesaw State University, USA</td>
</tr>
<tr>
<td>ACM Program Coordinator</td>
<td>Irene Frawley</td>
<td>ACM HQ</td>
</tr>
</tbody>
</table>

Notice to Contributing Authors

By submitting your article for distribution in this Special Interest Group publication, you hereby grant to ACM the following non-exclusive, perpetual, worldwide rights:

- to publish in print on condition of acceptance by the editor
- to digitize and post your article in the electronic version of this publication
- to include the article in the ACM Digital Library and in any Digital Library related services
- to allow users to make a personal copy of the article for noncommercial, educational or research purposes

However, as a contributing author, you retain copyright to your article and ACM will refer requests for republication directly to you.
A Message from the Editor

Welcome to the summer issue of Applied Computing Review! It is my pleasure to tell you that the 28th annual *ACM SAC (Symposium on Applied Computing)* was held in Coimbra, Portugal in March, and I would like to take this opportunity to thank everyone who made the conference a great success. This issue of ACR includes one invited paper from a group of prominent researchers, four selected papers presented at the 2013 *ACM SAC*, and one selected paper from the 2012 *ACM RACS (Research in Applied Computation Symposium)*. All the selected papers have been revised and expanded for inclusion in ACR, and I am proud to tell you that all of them maintain high quality. I am grateful to the authors for contributing the state-of-the-art methods in their research area and to the highly qualified reviewers who coordinated an outstanding lineup of technical paper reviews.

ACR is available to everyone who is interested in the modern applied computing research trends. Our goal is to provide you with a platform for sharing innovative thoughts among professionals in various fields of applied computing. We are working with the ACM SIG Governing Board to further expand SIGAPP by increasing membership and developing a new journal on applied computing in the near future.

In closing, I would like to remind you that the 2013 ACM SIGAPP election is being conducted. Eligible members are encouraged to participate in this election, and the deadline to cast your vote is June 14th. Your continuous support and kind cooperation would be greatly appreciated. Have a wonderful summer. Thank you.

Sincerely,

Sung Shin
Editor in Chief & Chair of ACM SIGAPP

Next Issue

The planned release for the next issue of ACR is September 2013.
SAC 2013 Report

The 28th Annual edition of the ACM Symposium on Applied Computing (SAC) was held in Coimbra, Portugal, March 2013. The conference was hosted by the Institute of Engineering of the Polytechnic Institute of Coimbra (ISEC-IPC). This year, the conference has great success with over 95% attendance rate. This is due to the exceptional dedication of the local organizing committee and the generous support the conference received from local sponsors. The Steering Committee extends its thanks and gratitude to the local sponsors for their generous contributions. This year, the conference featured technical sessions, tutorials, keynote speakers, posters, and student research competition.

The Call for Tutorials attracted 8 submissions, of which 6 were accepted for presentation. The tutorials covered verity of topics and attracted over 120 attendees.

The Call for Track Proposals resulted in accepting 36 tracks. The selections were made based on the success of those Tracks in the previous editions of SAC as well as targeting new and emerging areas in applied computing. The Tracks were organized into five different themes: AI & Agents, Distributed Systems, Information Systems, Software Development, and System Software & Security. The Symposium Proceedings and the technical presentations were focused around these themes to form a series of related track sessions.

The Call for Papers attracted 1063 paper submissions from 58 countries. All submitted papers underwent the blind review process and 255 papers were finally accepted as full papers for inclusion in the Conference Proceedings and presentation during the Symposium. The final acceptance rate for SAC 2013 is 24% among all tracks. In addition, 73 papers that received high review scores were invited as short papers for presentation during the Posters Program. The four-day Technical Program consisted of two keynote sessions and research presentations from all 36 tracks covering a wide range of topics on applied computing and emerging technologies. For more details about the technical program, please visit http://www.acm.org/conferences/sac/sac2013/.

The Call for Student Research Abstracts, for participation in the Student Research Competition (SRC) program, attracted 38 submissions, of which 6 submission were invited for participation in the program. The SRC program is sponsored by Microsoft Research. Invited students participated in poster display and oral presentations. A committee of five judges evaluated the posters and selected five winners for the second round (oral presentations). The judges then selected top three winners from the oral presentations round. The winners were recognized during SAC Banquet and presented with award medals and cash prizes (paid directly by ACM after the conference). The winners were:

First Place: Laura Micconi from the Technical University of Denmark, Denmark.
Second Place: Huwaida Tagelsir Elshoush from the University of Khartoum, Sudan.
Third Place: Razvan Ranca from the University of Edinburgh, United Kingdom.

The success of SAC 2013 was made possible through the hard work of many people from the scientific community who had volunteered and committed many hours to make it a successful event, especially, the Track Chairs and their Program Committees. On behalf of the Organizing and Steering Committees, we congratulate all of the authors for having their papers accepted in their respective Tracks. We also wish to thank all of those who made this year's technical program a successful one, including the speakers, track chairs, reviewers, program committee members, session chairs, presenters, and attendees.
The preparation for SAC 2014 is underway. The conference will be held in the historic city of Gyeongju, Korea. It is hosted by Seoul National University, Kyungpook National University, Soongsil University, and Dongguk University. We hope you consider SAC 2014 for your next submission and hope to see you there next year. Please see the organizing committee at http://www.acm.org/conferences/sac/sac2014/.

SAC 2015 and 2016 will be held in Salamanca (Spain) and Pisa (Italy), respectively. Please contact any member of the steering committee for “SAC Hosting Guidelines” if you are interested in hosting SAC in the near future.

Best Regards to all,

Hisham M. Haddad
Member of SAC Organizing and Steering Committees
Buffer Overflow Patching for C and C++ Programs: Rule-Based Approach

Hossain Shahriar  
Department of Computer Science  
Kennesaw State University  
Kennesaw, GA 30144, USA  
hshahria@kennesaw.edu

Hisham M. Haddad  
Department of Computer Science  
Kennesaw State University  
Kennesaw, GA 30144, USA  
hhaddad@kennesaw.edu

Ishan Vaidya  
Department of Computer Science  
Kennesaw State University  
Kennesaw, GA 30144, USA  
ivaidya@students.kennesaw.edu

ABSTRACT
The presence of buffer overflow (BOF) vulnerabilities in programs hampers essential security objectives such as confidentiality, integrity and availability. In particular, exploitations of BOF might lead to many unwanted consequences including denial of service through program crash, control flow hijacking, and corrupted program state. When BOF vulnerabilities are detected, they need to be patched before the software is re-deployed. Source level automatic patching of vulnerabilities has the challenges of finding a set of general rules and consistently applying them without bringing any side effects to intended software. This paper proposes a set of general rules to address the mitigation of BOF vulnerabilities for C/C++ programs. In particular, we developed a set of rules to identify vulnerable code and how to make the code vulnerability free. The proposed rule-based approach addresses both simple (one statement) and complex (multiple statements) forms of code that can be vulnerable to BOF ranging from unsafe library function calls to the pointer usage in control flow structures (loop and conditional statements). We evaluated the proposed approach using two publicly available benchmarks and a number of open source C/C++ applications. The results show that the proposed rules can not only identify previously known BOF vulnerabilities, but also find new vulnerabilities. Moreover, the patching rules impose negligible overhead to the application.

Categories and Subject Descriptors
D.2.5 [Testing and Debugging]: Code inspections and walkthroughs; D.2.7 [Distribution, Maintenance, and Enhancement]: Enhancement.

General Terms
Security, Languages, Verification.

Keywords
Rule-based patching, software vulnerability, buffer overflow, unsafe library function calls, pointer usage.

1. INTRODUCTION
Despite implementing programs with caution and security in mind, vulnerabilities are still discovered widely [4, 17, 27]. Vulnerabilities can be exploited by hackers to perform control flow hijacking and program state corruption. Therefore, programs need to be patched immediately once vulnerabilities are reported. However, in real-world it is common to see that many reported vulnerabilities are not patched in a timely manner [24-26]. A primary reason is that developers need to know the replacement for the responsible source code without changing the intended functionalities during patching. To address this issue, it is important to have a set of generic rules to perform patching of vulnerabilities at the source code level. Automatic patching of vulnerabilities has the challenges of finding a set of general rules and consistently applying them without bringing any side effects to the software. This work addresses the issue of identifying a set of general rules to facilitate patching source code.

In this paper, we present a set of patching rules that can be applied to remove vulnerabilities from programs [23]. We focus on Buffer Overflow (BOF) [1] vulnerabilities that can be exploited through unchecked copy (read) operations to (from) data buffers and format function calls during the generation or displaying formatted outputs, respectively. The presence of vulnerabilities may bring abnormal program behaviors such as control flow hijacking and program crash.

Many detection techniques are available for detecting BOF such as static analysis [5, 6, 7], runtime checking [8, 9, 10], and testing [2, 21]. However, once BOF is detected, patching needs to be done to ensure that the vulnerabilities are removed without introducing any side effects or major code modifications. Ideally, programs are patched before being redeployed. When patching programs, a set of general rules comes as an important step to perform automatic source code patching.

The proposed approach is motivated by a number of observations by examining related works on patching BOF ([12, 14]). First, most approaches perform patching of software at intermediate source code level (e.g., assembly) that is challenging to relate the semantics of high level application code. Second, some approaches perform patching on source code level (program transformation). A few works have addressed BOF patching at source code level (program transformation [13, 15]). However, these approaches do not cover complex forms of BOF vulnerabilities (e.g., BOF caused by pointer operations, null pointer usage that may be present in control flow-based statement such as loop and if-else statements). In practice, some BOF vulnerabilities cannot be fixed by replacing single line of code with another single line (e.g., a buffer overflow present in a loop conditional expression that may have a relationship with how the pointer is incremented and where it is incremented). Thus, the development of complex patching rules is important.

1Copyright is held by the authors. This work is based on an earlier work: ITNG’13, Proceedings of the 2013 IEEE International Conference on Information Technology: New Generations, Copyright 2013 IEEE 978-0-7695-4967-5.
In this paper, we identify BOF vulnerable code structures and corresponding patching code (denoted the process as rule-based patching). We developed a set of code structures that represent BOF due to ANSI C string (e.g., strcpy) and format (e.g., printf, sprintf) library function calls, array indexes, numeric overflows and underflows, lack of null characters at the end of buffers, pointers used in arithmetic operations and loop iterations. To complement, we also addressed vulnerable code patterns for C++ programs and developed corresponding patching rules. We applied the developed patching rules to remove BOF vulnerabilities found in C and C++ programs.

We evaluated the effectiveness of the proposed rules by applying them to C and C++ applications in publicly available benchmarks that contain BOF vulnerabilities. The results indicated that the proposed rules can fix vulnerable code without changing the applications functionalities. Further, they can detect unknown and complex forms of vulnerabilities and require very negligible performance overhead.

The paper is organized as follows: Section 2 provides an overview of related work; Section 3 presents the proposed approach; Section 4 discusses the evaluation and experimental results; Section 5 provides concluding remarks; and Section 6 highlights future work.

2. BACKGROUND AND RELATED WORK

2.1 Overview of BOF Vulnerability

A buffer overflow (BOF) vulnerability occurs while writing data to a program buffer exceeding the allocated size, and overwriting the contents of the neighboring memory locations. The overwriting corrupts sensitive neighboring variables of the buffer such as the return address of a function or the stack frame pointer. BOF can occur due to vulnerable ANSI C library function calls, lack of null characters at the end of buffers, accessing buffer through pointers and aliases, logic errors (off by one), and insufficient checks before accessing buffers in program code.

We provide an example of BOF in a C code snippet in Figure 1. Function foo has buffer buf that is located inside the stack region. It has a capacity of eight bytes of data. The valid location of this buffer is between buf[0] and buf[7]. Line 4 has a strcpy function call which is vulnerable to BOF as a large size string of 14 bytes is being copied into buf.

```c
1. void foo (int a) { 
2.   int var1;
3.   char buf [8];
4.   strcpy(buf, "AAAAAAAAAAAAAA");
5.   return;
6. }
```

Figure 1. C code snippet showing vulnerable line of code.

Figure 2 shows the stack layout of function foo before strcpy function call. Note that variable var1 is located immediately after the ending location of the buffer, followed by the stack frame pointer (sfp) and the return address (ret) of function foo. The return address indicates the memory location where the next instruction is stored and is read immediately after the function is executed.

![Figure 2. Stack layout of foo before strcpy function call.](image)

A BOF might happen during reading or writing operations. Writing past the buf by at least one byte corrupts the value of var1 (assuming no padding performed by a compiler). If overwriting spans more than one byte in stack, it might modify the return address (ret) of function foo. As a result, when the function tries to retrieve the next instruction after its execution, the modified location might not fall within the valid address space of the program. This might result in a segmentation fault and the program crashes.

2.2 Static Analysis and Runtime Technique

Static analysis tools detect BOF and FSB in source code without executing the source code. The ITS4 tool [5] searches potentially unsafe functions (i.e., functions that do not check the buffer length before any operation). Flawfinder [6] generates a list of potential security flaws sorted by risk using text pattern matching from source code. Risk levels are assigned based on the context of function calls (values of the parameters). However, these tools suffer from numerous false positive warnings and do not fix BOF and FSB vulnerabilities.

Ding et al. [22] proposed a pattern based on Symbolic Evaluation for BOF. They developed three general patterns which can cause the programs to be vulnerable to BOF attacks. For checking the effectiveness of the proposed approach, they implemented a system based on CodeSurfer infrastructure. However, the proposed method suffers from false negatives cases. User defined functions that perform the same task as strcpy function calls are not considered in their approach.

Evan et al. [7] proposed a lightweight static analysis approach for ANSI C program to detect BOF. They developed the Splint (Secure Programming Lint) tool. It requires programmers to annotate ANSI C code using special comments (e.g., "/*@notnull*/") that represent desired properties of applications to be maintained during execution. However, the annotation of the properties in code is manual and the analysis of false positive warnings afterwards is a huge burden. In contrast to these...
A number of works detect BOF at runtime. Cowan et al. [8] developed the Stackguard tool which prevents corruption of stack return addresses due to BOF attacks during runtime. They placed the word canary before the return address stored in the stack by modifying the function’s prolog. However, the approach cannot prevent corruption of other sensitive variables due to BOF. StackShield [9] is a compiler patch for GCC that protects function return addresses and function pointer modifications due to BOF. The tool copies the function return addresses into global variables at the beginning function calls (prolog) that cannot be overflowed using buffer. When a function returns (epilog), the tool checks if current return addresses are different from previously saved addresses. If the two values are different, the program is terminated.

Etch et al. [10] reorganized the declared local variables and pointers in stacks in such a way so no overflow can occur for old base pointers and return addresses. A guard variable having a random value is placed between a base pointer and a local buffer during runtime. Necessary code is inserted to check if guard values are modified due to BOF. However, a buffer can still overflow by one byte and modify its neighbor variable leaving the base pointer intact.

Readers are directed to see [3] and [31] for a comprehensive discussion of static analysis and runtime techniques for BOF mitigation.

### 2.3 Patching of BOF

Smirnov et al. [12] developed an automatic patch generation method to prevent further BOF attacks after an initial attack. They record runtime information to reconstruct the data and control dependency during a BOF attack. The tool adds necessary bound checking in object code to prevent BOF. Our approach works directly at the source code level and addresses complex forms of BOF caused by pointer level memory boundary checking.

Dahn et al. [13] work repositions the stack buffers into heap area to prevent stack smashing attacks. They replace local buffers with dynamically allocated buffers. Although the approach reduces the attack scope, it cannot prevent heap-based BOF by one or more byte. In contrast, our approach fixes BOF due to pointers used to access and modified heap memory.

Gao et al. [14] proposed metamorphic software generation approach to avoid BOF attacks that allow executing arbitrary supplied code. Two applications are said to be metamorphic to each other, if their functionality remains the same. However, the internal structures of applications are different. They obtain metamorphic application for a given application using assembly code injection to keep functionalities intact and reduce the possibilities of successful BOF attacks. The injected assembly code includes NOP, JUMP, Logical and arithmetic instructions, PUSH, and POP instructions. However, their approach cannot prevent many BOF attacks such as arbitrary reading. Our proposed approach, on the other hand, can handle arbitrary readings.

Austin et al. [15] work applies program transformation during compile time to prevent memory access errors. They use safe pointer representation of ordinary pointers, which consist of pointer values along with the attributes pointer’s location, size, and lifetime. The approach increases program complexity in terms of storage (to save states of pointers and arrays) and runtime. In contrast, our approach does not impose any storage and runtime overhead.

### 3. RULE-BASED PATCHING

We identify code segments from vulnerability taxonomy [16], reported vulnerabilities [17], and ANSI C library specification [19]. We developed a corresponding set of patching code based on common guidelines for securing applications [11]. The modified programs after patching are re- compilable. Below, Sections 3.1 and 3.2 describe BOF vulnerable code segments and the corresponding patching rules, respectively. Section 3.3 describes the BOF vulnerable code patterns and corresponding rules for C++ programs. Section 3.4 discusses auxiliary modifications of application code to facilitate the patching.

#### 3.1 Vulnerable Code Segments for BOF

Common sources of BOF vulnerabilities include unsafe library function calls, buffer index variables, absence of null characters, arithmetic operations using pointers, and pointer usage in complex code blocks (e.g., loop and if structures). The first four types represent simple form of BOF (single line of code); while the fifth type represents a complex form of BOFs (multiple lines of code).

The first three columns of Table 1 show the elements responsible for BOF, vulnerability descriptions, and examples of code patterns that are vulnerable to BOF. A BOF might occur due to unsafe ANSI library function calls, which do not check destination buffer sizes before performing specific operations. The first example in Table 1 shows a library function call strcpy(dest,src) that copies the content of src into dest without checking the maximum size of buffer dest. Most applications access a buffer using index variables that are used to retrieve or assign memory location of buffers. Unfortunately, the values of index variables are not validated before accessing the buffers. The second example shows that an increment of the index variable i might not always point to locations that belong to buffer dest. Moreover, arithmetic operations on buffer indexes might often result in numeric overflows and underflows (third example in Table 1). Here, i is a short signed integer variables whose value becomes negative, if an increment operation is performed just after reaching the maximum positive value. If such values are used as index to access a buffer, unpredictable results might happen in the application such as segmentation faults.

The addition of a null character at the end of the buffer needs to be done explicitly in code. However, BOF may occur due to the absence of null characters at the end of buffers (fourth example in Table 1). The usage of pointers and aliases opens another avenue for BOF. For example, a pointer can be used as an operand in arithmetic operations followed by saving the end result. The resultant address might not belong to any valid location of a buffer. The fifth example in Table 1 shows that p is a pointer to buffer dest and is used in an arithmetic operation to obtain new address (p = p + 32). The resultant address lies outside buffer dest.
A loop block is vulnerable to BOF, if it has pointers in conditional expressions and is terminated based only on values of the pointed variables. Here, a complex code has multiple control and data flow patterns of BOF. Table 2 shows three types of complex code patterns that can lead to BOF. Table 1 shows examples of simple code patterns causing BOF in C programs.

### Table 1. Examples of simple code patterns causing BOF in C programs.

<table>
<thead>
<tr>
<th>Responsible element</th>
<th>Vulnerability description</th>
<th>Vulnerable code segment</th>
<th>Patching rule description</th>
<th>Patched code segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI C Library function calls</td>
<td>Functions do not check destination buffer length before any operation.</td>
<td>char dest [32];… strcpy (dest, src);</td>
<td>Rule #1: Replace unsafe functions with similar types of safe function calls.</td>
<td>char dest [32];… strcpy (dest, src, sizeof(dest));</td>
</tr>
<tr>
<td>Index variables</td>
<td>Buffer index variable values are not checked before accessing a buffer location.</td>
<td>char dest [32]; for (i=0; i &lt; sizeof (src); i++){ dest[i] = src[i]; }</td>
<td>Rule #2: Perform modulus arithmetic to prevent buffer index being outside valid ranges of buffer.</td>
<td>char dest [32]; for (i=0; i &lt; sizeof (src); i++){ dest[i % 32] = src[i]; }</td>
</tr>
<tr>
<td>Null characters</td>
<td>Numeric overflow (or underflow) of index variables results in high negative (or positive) numbers.</td>
<td>char dest[65600]; signed short i; for (i=0; i &lt; sizeof (src); i++){ dest[i] = src[i]; } //overflow</td>
<td>Rule #3: Replace signed index variables with unsigned type variables.</td>
<td>char dest[65600]; unsigned short i; for (i=0; i &lt; sizeof (src); i++){ dest[i] = src[i]; }</td>
</tr>
<tr>
<td>Pointer arithmetic</td>
<td>Pointer points to invalid location after performing arithmetic operation and obtaining new address from the operation.</td>
<td>char dest [32]; char *p = dest; p = p+ 32; strncpy (dest, p, 32); strcat (p, ”’);</td>
<td>Rule #4: A check if pointers point to invalid locations after performing arithmetic operation. Checks if copying operation is safe on heap memory.</td>
<td>char dest [32]; char *p = dest; p = p+ 32; if (p – dest &lt; 32) strncpy (dest, p, 32); if (strlen(p) + strlen(“’) &lt; 32) strcat (p, “’);</td>
</tr>
</tbody>
</table>

BOF can be more complex when it is accessed through complex code blocks (e.g., loop, if, switch) and controlled by conditional expressions that include pointer variables. These pointer variables are directly or indirectly related to buffers. Table 2 shows three complex code patterns of BOF. Here, a complex code has multiple lines of code that overall contributes to the occurrence of a BOF. A loop block is vulnerable to BOF, if it has pointers in conditional expressions and is terminated based only on values pointed by those pointers.

To avoid BOF, loops should be terminated by comparing the pointer’s current addresses with the last location of pointed buffer. Example 1 in Table 2 illustrates this cause. Here, a while loop has a conditional expression *src++ != ’\0’, where src and dest are pointers to source and destination buffers, respectively. The src pointer is iterated until a null character is found that might result in BOF as there is no check of the src pointer against the last location of the buffer.

### Table 2. Examples of complex code patterns causing BOF in C programs.

<table>
<thead>
<tr>
<th>Responsible element</th>
<th>Vulnerability description</th>
<th>Vulnerable code segment</th>
<th>Patching rule description</th>
<th>Patched code segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointers inside structured code block</td>
<td>Pointers are used to terminate loops based on pointed values only. No address comparison is performed.</td>
<td>while (*src++ != ‘\0’){… *dest = *src; } *++src = ’a’;</td>
<td>Rule #6: Augment condition expressions to check current pointers against end of pointed buffer locations. It can be applied to loop and non-loop cases.</td>
<td>while (src &lt; buflim-1 &amp;&amp; *src++ != ’\0’){… *dest = *src; } if (src &lt; buflim) *++src = ’a’;</td>
</tr>
<tr>
<td>Pointers inside structured code block</td>
<td>Pointers are used to terminate loops by comparing them with dynamic sentinel pointers. However, sentinel pointers are not checked for BOF.</td>
<td>while (src &lt; buflim){… *dest = src++; buflim++; }</td>
<td>Rule #7: Augment condition expressions to check sentinel pointers against pointed buffers end locations.</td>
<td>while ((src &lt; buflim-1) &amp;&amp; (buflim &lt; &amp;source[MAX])) {… *dest = src++; buflim++; }</td>
</tr>
<tr>
<td>Multiple pointer increment and decrement operations in conditional statement blocks (if and switch) do not check end of buffer correctly.</td>
<td>if (*src++ != ’\0’ &amp;&amp; src &lt; buflim){… *dest = *src++; }</td>
<td>Rule #8: Modify condition expressions to check if multiple increments or decrements do not exceed pointed buffers.</td>
<td>if ((src &lt; buflim -2) &amp;&amp; (*src++ != ’\0’)){… *dest = *src++; }</td>
<td></td>
</tr>
</tbody>
</table>
It is common to use pointers as markers for buffer ends. We denote these marker pointers as sentinel pointers or guard pointers. They are widely used in loop iteration to avoid BOF. However, these markers are often modified dynamically. As a result, a check for potential BOF fails if an iteration pointer is compared to a sentinel pointer only. We believe that a sentinel pointer needs to be compared to its original pointed buffer’s end location to avoid BOF.

Example 2 in Table 2 shows that a while loop is iterated using a pointer of source buffer (src). The buflim is a sentinel pointer that marks the end of source buffer. The loop condition checks whether the src exceeds the buflim or not. However, the buflim is dynamically modified inside the loop (buflim++). There is no check whether the buflim pointer is exceeding the last valid location of the source buffer.

Condition statements of the if and switch blocks are often used to perform pointer increment and decrements along with value comparison. Moreover, inside the code block, often unconditional increment or decrement operations are preformed. It is common that the condition expression only checks BOF due to one operation only. Example 3 in Table 2 illustrates this case. Here, src and dest are pointers to source and destination buffers, respectively, and buflim is a sentinel pointer to source buffer. There are two increment operations performed on src pointers. However, BOF related check is preformed for one increment only. The code is vulnerable to BOF by one byte.

3.2 BOF in C Programs

We believe that BOF can be patched by considering the previous previous code patterns that access and modify buffers. To avoid BOF, we propose a set of eight rules to be applied at the unit level of an implementation. Tables 1 and 2 show the relation between responsible elements and rules (in fourth and fifth columns). Examples of rules application are also shown in the tables.

The first rule replaces unsafe ANSI C library function calls with safe library function calls. For example, the strcpy call is replaced by strncpy function call. The second rule performs modulus arithmetic operation on buffer index variables while accessing the buffers. In this example, we replace dest[i] by dest[i % 32] so that the resultant locations always remain within the valid location of buffer dest. To prevent numeric overflow and underflow related to BOF, we replace signed index variables with unsigned variables (third rule) as illustrated in Table 1.

To prevent BOF due to lack of null characters at the end of a buffer, we add a null character statement before accessing the buffer (fourth rule). BOF due to pointer arithmetic operation in a statement can be prevented by adding a check before performing the operation. The fifth example in Table 1 checks whether resultant memory locations of pointers remain within valid locations of the pointed buffers (i.e., if (p – dest < 32).

The complex form of BOF (Table 2) due to structured code block with pointer usage can be prevented primarily by modifying the condition expression of the code block. We propose to augment the condition expression of the loop with an extra condition to avoid BOF. The new condition compares the address of a pointer used for iteration with end of the buffer. In example 1 of Table 2, we add the condition src < buflim-1, where src is an iteration pointer for source buffer and buflim is the end location of the source buffer. If dynamic sentinel or guard pointers are used in the condition expression of loops, we add extra conditions for the sentinel pointers. The new condition checks if a sentinel pointer remains within the limit of its pointed buffer’s last location. In the second example, we add a new condition buflim < &src[MAX], where buflim is a sentinel pointer in the while loop, src is the source buffer, and MAX is the number of bytes allocated by src.

To avoid BOF due to multiple arithmetic operations inside an if condition and its code blocks, we add or modify the existing check on pointers against sentinel pointers. The third example in Table 2 shows that there are two increment operations for the src pointer. We add an extra condition inside the if statement that checks whether two increments can be done safely (i.e., src < buflim -2). Here, buflim is the sentinel pointer of the source buffer and src is a pointer to src buffer.

Note that rules 6 and 7 are applicable to the condition expression of loops only. If there is an inner loop then these rules are applied recursively. Similarly, rule 8 accounts for if and switch conditions along with their corresponding code blocks. If there is another if block then the rule is applied recursively. Operations inside an alternative block (e.g., else if, else, default for switch) are not considered for this rule. We believe that rule 5 can prevent BOF for these cases.

3.3 BOF in C++ Programs

For C++, we developed two categories of patching rules described as follows in this section.

First Category: Buffer accepting lines rules (Rule #9)

C++ does not provide bounds-checking for buffers that accept lines as inputs. As a result, if an input contains more characters than the capacity of the buffer, then it overwrites the adjacent memory locations. The first row of Table 3 shows an example of character buffer (str) that has five bytes of storage capacity. The statement cin >> str causes buffer overflow. Similarly, the getline method is vulnerable to BOF. We propose to patch them in two ways (denoted as 9a and 9b in the example). We either alter the character buffer to string data type supported in C++. The std::string class represent character buffer and is not vulnerable to buffer overflow. It can expand its memory dynamically. We show the application of the patched rule in the last column in Table 3. The other option (9b) is to replace cin >> str with std::cin.get(str, 5) where 5 is the maximum number of bytes allowed to copy in str. Similar replacement can be performed for getline method call (9c).

C++ language supports placement new operator. In particular, the operator allows allocating a specific amount of data buffer from a pool of predefined memory location. However, the size of predefined memory may not be large enough to satisfy the requested amount of memory.
Placement new operator allows allocating memory from pre-specified pools of memory. The pool helps in managing limited memory in a program and avoids wastage of memory resources. However, when the memory is allocated with placement new operator, the target allocation address can be specified with any memory location belong to a program address space. If the target address does not have enough capacity to allocate the request amount of memory, then BOF occurs during copying operation at a later time [29].

**Second Category: Placement new expression.**

This category includes three types of BOF patterns.

**BOF due to dissimilar primitive data types (Rule #10):** Here, an initial data variable is used as a pool which has a lower memory capacity. Later another data variable is allocated on the location of the pool that requires a higher amount of memory. Thus, an overflow occurs. For example, the second row (the third column) of Table 3 shows an example of this situation. Here, the placement new operator is used to allocate an integer variable (declared as \( \text{char} \)). However, the target allocation space has the capacity to allocate a character variable (declared as \( \text{int} \)).

We fix this vulnerability by upgrading the destination memory data type from \( \text{char} \) to \( \text{int} \). We generalize the fixing method as Rule #10 which makes dissimilar data types consistent and converts low-level data type (require less amount of memory) to a high-level data type (require high amount of memory).

**BOF due to copying subtype objects into buffers created for supertype objects (Rule #11):** The second type of BOF can be found while copying one object to the destination memory pool. Here, the initial memory pool is formed with supertype object and the destination object is a subtype object. As a result, during the instantiation process of the subtype object, more space are overwritten since subtype object has more fields than supertype object. For example, let us consider the following two classes in Figure 4. Here, \( \text{Foo1} \) is a superclass whereas, \( \text{Foo2} \) is a subclass. \( \text{Foo1} \) has one member variable with the corresponding constructor, setter (\( \text{setA} \)) and mutator (\( \text{getA} \)) methods. \( \text{Foo2} \) is a subclass of \( \text{Foo1} \) which has an extra variable named \( \text{b} \). The constructor, setter, and mutator, are defined in the subclass. Note that the memory space required to copy the subclass is larger than that of the superclass. Here, we assume that variable \( a \) occupies two bytes, whereas, each of the constructor and the two methods occupies 4 bytes. Thus, it takes 14 bytes for allocating an instance of \( \text{Foo1} \) and 16 bytes for allocating an instance of \( \text{Foo2} \).

**Table 3. Examples of code patterns causing BOF in C++ programs.**

<table>
<thead>
<tr>
<th>Responsible element</th>
<th>Vulnerability description</th>
<th>Vulnerable code segment</th>
<th>Patching rule description</th>
<th>Patched code segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer accepting lines</td>
<td>No bounds-checking for buffer declared locally and taking line as input.</td>
<td>char str[5]; cin &gt;&gt; str; getline(cin, str, 'n');</td>
<td>Rule #9: Replace the character data type with string data type that does not overflow and can expand its size dynamically.</td>
<td>std::string str; //9a cin &gt;&gt; str; std::cin.get (str, 5); //9b std::cin.getline(str, 5, 'n'); //9c</td>
</tr>
<tr>
<td>Placement new expression</td>
<td>Dissimilar primitive data types involved in memory allocation.</td>
<td>char c; int *b = new (&amp;c) int; *b = 50;</td>
<td>Rule #10: Make dissimilar data type consistent and convert low-level data type to a high-level data type.</td>
<td>int ch; int *i = new (&amp;ch) int;</td>
</tr>
<tr>
<td></td>
<td>Copying subtype object into an allocated buffer space that only fits for a supertype object.</td>
<td>char buffer[14]; Foo1* obj = new (buffer) Foo1(3); ... Foo2* obj2 = new (buffer) Foo2(3, 5);</td>
<td>Rule #11: Add a check to ensure that destination buffer size has sufficient space for subtype object.</td>
<td>if (sizeof(buffer) &lt; sizeof(Foo2)) { buffer = (char*) malloc (sizeof(Foo2)); } Foo2* obj2 = new (buffer) Foo2(3, 5);</td>
</tr>
<tr>
<td></td>
<td>Copying subtype object into the space of a supertype object.</td>
<td>Foo1 f1; Foo2* f2 = new (&amp;f1) Foo2(3, 5);</td>
<td>Rule #12: Add a check to define supertype object as subtype to ensure that sufficient space is available to copy a subtype object.</td>
<td>if (sizeof(f1) &lt; sizeof(Foo2)) { f1 = new Foo2(0, 0); } Foo2* f2 = new (&amp;f1) Foo2(3, 5);</td>
</tr>
</tbody>
</table>

**Figure 4.** Superclass and subclass definition in C++.
Let us assume that we allocate a buffer of 14 bytes which is suitable for copying the superclass (Table 3, third row, third column). If we reuse the same buffer to allocate an instance of subtype object Foo2, then a BOF occurs by 2 bytes, overwriting the memory adjacent to the character buffer.

To fix this problem, add an extra check to ensure that the destination buffer size has sufficient space to allocate an instantiated subtype object (Rule #11). The example code added for this checking is shown in the last column of the second row. Similar checking can be added for other similar cases where objects are instantiated and placed in allocated buffers fit for supertype objects.

**BOF due to copying subtype object into supertype object space (Rule #12):** The third case that we consider is copying subtype object at the location of a supertype object. Let us consider an example shown in the third row of Table 3 (fourth row, third column). Here, f1 is an instance of a supertype object Foo1. f2 is an instance of Foo2 where the allocation space is pointed by f1 which does not fit the space needed for Foo2. Thus, an overflow occurs here. We solve this by redefining f1 as an instance of Foo2 instead of Foo1 (Rule #12). An extra checking is done to ensure that subtype objects are copied to a location that has enough space to hold its variable.

### 3.4 Function redefinition

To apply the proposed rules to an existing application, we first augment function definition so that the length information corresponding to buffer arguments are passed inside a program unit or function. Consequently during actual function calls, the argument numbers are increased. For example, Table 4 shows the initial code for function foo defined in an application. The code shows a formal parameter (filename) that is a character buffer. There is no information related to the length of the buffer.

**Table 4. Function redefinition example.**

<table>
<thead>
<tr>
<th>Initial code</th>
<th>Modified code</th>
</tr>
</thead>
<tbody>
<tr>
<td>void foo (char *filename) {</td>
<td>void foo (char *filename, int len) {</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>int main (int argc, char *argv[]) {</td>
<td>int main (int argc, char *argv[]) {</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>foo (argv[1]);</td>
<td>foo (argv[1], strlen(argv[1]));</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

In this situation, BOF related to pointer arithmetic and index variables cannot be fixed. To address this, function foo is modified to include the length of the corresponding buffer (len parameter). However, this mandates another modification at the point of function call (e.g., foo (argv[1]) is replaced with foo (argv[1], strlen(argv[1]))). The first three rules are applied by looking at each statement of a program unit. The fourth rule is applied before using the buffers (e.g., a buffer passed as an argument of a user defined or library functions), so that their last locations contain null characters. The fifth rule is applied when a pointer variable is assigned in a statement whose right hand side is an arithmetic expression. The sixth and seventh rules are applied to the conditional expression of loops, whereas, the eighth rule is applied to condition statements (if and switch).

### 4. EVALUATION

We applied the proposed rules for detecting and patching BOF to two publicly available benchmark suites. The first benchmark was developed by Zister et al. [20] (denote as MIT LL), while the second benchmark is widely known as Bugbench and contains vulnerable applications developed by Lu et al. [28]. Sections 4.1 and 4.2 discuss the feature of the benchmark applications and the evaluation results.

#### 4.1 Evaluation with C Benchmark (MIT LL)

The benchmark developed by Zister et al. [20] contains 14 model programs with and without BOF vulnerabilities. These programs are developed from three real world applications which have been reported in the CVE [17] and CERT [18] databases for having exploitable security bugs. The programs include Wu-ftpd (ftp server), Sendmail (email server), and Bind (DNS server). Wu-ftpd, Sendmail, and Bind have three (f1-f3), seven (s1-s7), and four (b1-b4) model programs, respectively.

The model programs have BOF vulnerabilities due to unsafe library functions calls, numeric overflows, logical errors in loop conditions, and pointer usages. The benchmark contains input test cases for each of the model programs that actually cause BOFs as well as normal test cases to observe functionalities. BOF locations vary among stack, heap, and block starts with symbol (bss) segment. The bss (or data segment) is used to allocate memory for global or static variables of a program.

We apply all the proposed patching rules for each of the model applications manually. Table 5 shows a summary of the number of code modifications based on each of the eight rules. Rule #5 is applied for 129 times followed by Rule #1 for 65 times. This means that it is very common to use pointers in arithmetic operations in C applications. Moreover, unsafe library functions are widely used by application developers. We noticed that null characters are not assigned explicitly in most of the cases whenever it is required (e.g., before passing buffers as arguments of functions). Sentinel pointers are not widely present in applications. However, they are not checked for BOF due to dynamic updating of their addresses. This phenomenon can be explained by the observation of programs s1, b1, b2, and b3 that have sentinel pointers and Rule #7 being applied 9 times to these programs. Furthermore, most of the times loop conditions do not check pointer addresses as a terminating condition. This results in applying Rule #6 widely in all the applications (except for programs s4 and b3, where loop conditions have no iteration pointers).

After applying the rules, we tested each of the applications with BOF test cases. We noticed that each of the applications became free from BOF (second last column of Table 5) and prevented all of the marked “BAD” BOF cases reported in the benchmark. The number of previously reported BOF is shown in the third last column of Table 5. Moreover, we compared the output between a vulnerable program and the corresponding fixed version. In all cases, the output remains the same. However, testing the fixed program with BOF test cases resulted in unusual outputs.
The reason is that all fixing rules do not throw exceptions or exit the entire applications in case of BOF occurrences. Still, from the point of BOF fixing, our approach is effective and shows promising results.

While fixing BOF in the applications, we noticed that some of the proposed rules prevented BOF that were not marked “BAD” by the authors of the benchmark. The last column of Table 5 shows the total number of newly identified vulnerabilities that are reachable with user supplied inputs. We consider a BOF occurrence as new when a rule is applied solely without any other rule (i.e., combined application of rules are not considered for discovering new BOF). Although applying all the rules make an application free from BOF more than applying a single rule, discovering new BOF based on single rule provides more hint on subtle BOF that might be ignored.

Table 6 shows the relationship between new BOF cases and the rules that prevented them. Rule #6 is the most effective rule for preventing BOF due to loop conditions that uses iterative pointers (complex form of BOF). We note that most of the time programmers ignore to compare pointer address with a correct end address of a buffer.

Table 6. New BOF cases in MIT LL benchmark.

<table>
<thead>
<tr>
<th>Application</th>
<th>Model program</th>
<th>Rule #4</th>
<th>Rule #6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wu-ftpd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>f2</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>f3</td>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>s4</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>s5</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>s6</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>s7</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sendmail</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>4</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>b2</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>b3</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>b4</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Passing a buffer argument to a function is often done without assigning a null character at the end of the buffer. If the buffer is used in the called function, it can overflow due to reading and writing through loop iterations or library function calls. Table 7 shows an example of a newly discovered BOF that has been fixed by applying the sixth rule. The curpath is a local buffer that copies another buffer. The where pointer is used to iterate curpath buffer by a while loop, which terminates based on null character value in curpath. By applying Rule #6, a new condition is added in the while loop (where<curpath[MAXPATHLEN]). Without the condition, the curpath buffer can overflow with appropriate test cases consisting of a large number of ‘.’ characters (at least by one byte).

Table 7. Example of fixing unreported BOF in model F3.

<table>
<thead>
<tr>
<th>Before fixing BOF</th>
<th>After fixing BOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *my_realpath(const char *pathname, char *result, char *chroot_path, int len_info){</td>
<td>char *my_realpath(const char *pathname, char *result, char *chroot_path, int len_info){</td>
</tr>
<tr>
<td>char curpath[MAXPATHLEN],</td>
<td>char curpath[MAXPATHLEN],</td>
</tr>
<tr>
<td>*where, *ptr, *last;</td>
<td>*where, *ptr, *last;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>strcpy(curpath,pathname);</td>
<td>strcpy(curpath,pathname);</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>where = curpath;</td>
<td>where = curpath;</td>
</tr>
<tr>
<td>while (*where != '\0'){</td>
<td>while (*where != '\0' &amp;&amp;</td>
</tr>
<tr>
<td>//new BOF</td>
<td>where &lt; &amp;curpath[MAXPATHLEN])</td>
</tr>
<tr>
<td>if (!(strcmp(where, &quot;.&quot;)))</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>where++;</td>
</tr>
<tr>
<td></td>
<td>continue;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

Table 5. Summary of rules applied to MIT LL benchmark.

<table>
<thead>
<tr>
<th>Application name</th>
<th>Model program</th>
<th>Rule #1</th>
<th>Rule #2</th>
<th>Rule #3</th>
<th>Rule #4</th>
<th>Rule #5</th>
<th>Rule #6</th>
<th>Rule #7</th>
<th>Rule #8</th>
<th># of BOF before fixing</th>
<th># of BOF after fixing</th>
<th># of new BOF fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wu-ftpd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>12</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>29</td>
<td>0</td>
<td>0</td>
<td>25</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>8</td>
<td>27</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>27</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>28</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Sendmail</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>15</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>25</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>34</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Bind</td>
<td></td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>25</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>34</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>15</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>25</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>34</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>65</td>
<td>8</td>
<td>10</td>
<td>52</td>
<td>129</td>
<td>42</td>
<td>9</td>
<td>31</td>
<td>82</td>
<td>0</td>
<td>24</td>
</tr>
</tbody>
</table>
Table 8. Characteristics of Bugbench benchmark.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Description</th>
<th># files analyzed</th>
<th>LOC</th>
<th># reported BOF</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Be-1.06</td>
<td>Interactive calculator.</td>
<td>3</td>
<td>3320</td>
<td>3</td>
<td>Heap, Stack</td>
</tr>
<tr>
<td>Gzip-1.2.4</td>
<td>Compression and decompression program (Linux).</td>
<td>1</td>
<td>1747</td>
<td>1</td>
<td>Stack</td>
</tr>
<tr>
<td>Man-1.5h1</td>
<td>On-line manual display utility (Linux).</td>
<td>2</td>
<td>1461</td>
<td>1</td>
<td>Stack</td>
</tr>
<tr>
<td>Ncompress-4.2.4</td>
<td>Compression and decompression program (Linux).</td>
<td>1</td>
<td>1934</td>
<td>1</td>
<td>Stack</td>
</tr>
<tr>
<td>Polymorph-0.4.0</td>
<td>Win32 to Unix filename converter.</td>
<td>1</td>
<td>305</td>
<td>2</td>
<td>Bss (global)</td>
</tr>
<tr>
<td>Squid-2.3</td>
<td>Caching proxy server for the web.</td>
<td>1</td>
<td>2564</td>
<td>1</td>
<td>Heap</td>
</tr>
</tbody>
</table>

Table 9. Summary of rules applied to Bugbench applications.

<table>
<thead>
<tr>
<th>Application name</th>
<th>Rule #1</th>
<th>Rule #2</th>
<th>Rule #3</th>
<th>Rule #4</th>
<th>Rule #5</th>
<th>Rule #6</th>
<th>Rule #7</th>
<th>Rule #8</th>
<th># of BOF before fixing</th>
<th># of BOF after fixing</th>
<th># of new BOF fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Be-1.06</td>
<td>40</td>
<td>62</td>
<td>0</td>
<td>40</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Gzip-1.2.4</td>
<td>12</td>
<td>14</td>
<td>0</td>
<td>12</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Man-1.5h1</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Ncompress-4.2.4</td>
<td>4</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Polymorph-0.4.0</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Squid-2.3</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>76</td>
<td>97</td>
<td>1</td>
<td>59</td>
<td>7</td>
<td>18</td>
<td>2</td>
<td>2</td>
<td>9</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

4.2 Evaluation with Bugbench Benchmark

We evaluated the proposed patching rules in Bugbench benchmark suite that includes C programs containing reported BOF vulnerabilities2. Table 8 shows the characteristics of the programs that include the number of files analyzed and patched, LOC, the number of reported BOF, and the location of BOF. The chosen programs include a calculator (Be-1.0.6), compression and decompression utilities (Gzip-1.2.4 and Ncompress-4.2.4), an online manual display utility (Man-1.5h1), a filename converter (Polymorph-0.4.0), and a caching proxy server for web (Squid-2.3). Among these programs, Be has three BOFs in heap and stack areas, Polymorph has two BOFs in bss segment (due to the presence of a global buffer), and Squid has one BOF in the heap area. The remaining programs have BOF in stack area (i.e., locally declared buffers are overflowed in defined methods).

Table 9 shows a summary of the applications of all rules and the obtained results (columns 2-9). We find that Rule #2 is applied for the highest number of times (97) followed by Rule #1 (76), Rule #4 (59), Rule #6 (18), and Rule #5 (7). In all these applications, a large portion of the buffer access is being performed through numeric index. Moreover, most of these applications rely on library function calls such as strcpy, strcat, and sprint. Pointer arithmetic operations are present in these applications that are often incremented or decremented by one unit. When a pointer variable is used as part of loop conditional statement, then the pointer value is compared against a known value (e.g., p != c). We notice that Rule #3, Rule #7, and Rule #8 are applied for very times. This implies that the applications rarely use signed integer as array index, and compares loop pointers against sentinel pointers.

Columns 10 and 11 of Table 9 show the number of BOF present before and after patching the applications, respectively. We notice that our rules successfully fixed all the reported BOF vulnerabilities. The last column shows the number of new BOF vulnerabilities that our rules are able to patch. These vulnerabilities are not reported by the authors of the benchmark [39]. Our approach successfully fixed four new BOF vulnerabilities (one in Man, one in Polymorphic, two in Squid) in these applications. We confirm the new vulnerabilities by manually inspecting source code and establishing the copying of inputs (coming through function’s arguments) to the stack or heap buffers.

An example of newly discovered BOF vulnerability in Squid program is shown in the first row (the left column) of Table 10. Here, strcpy copies an input (request->login) directly to loginbuf (a locally declared buffer) without checking the size of the destination buffer. Note that the checking before invoking strcpy function ensures that the source buffer length is greater than zero. Thus, this results in BOF vulnerability for loginbuf. The right column of Table 10 shows our applied rule (Rule #1) that substitute strcpy with strncpy where the size information of loginbuf is supplied as the argument (size of loginbuf).

The second row of Table 10 shows a reported BOF vulnerability where strcat function is used to append the loginbuf with @ symbol. Here, it is not checked whether concatenation operation can potentially exceed the destination buffer allocated size. We show the application of our fixing rule (Rule #5) in the right hand side column where we added an extra check to ensure that the current length of loginbuf and the length of the concatenated string does not exceed the allocated size of the destination buffer loginbuf. The size of the loginbuf can be identified easily from the local source file.

2 We excluded applications from Bugbench that do not contain buffer overflow vulnerabilities such as MySQL and CVS.
The proposed rules fix previously unknown BOF cases that are complex in nature. The results indicate that the improper usage of library function calls through conditional statement, absence of null character assignments and incorrect loop conditions often lead to subtle BOF vulnerabilities that are difficult to identify. In addition, the proposed rules can be modified to become more context sensitive so that the rules do not add redundant code to the applications.

Table 10. Fixing BOF in Squid application.

<table>
<thead>
<tr>
<th>Program</th>
<th>Before fixing BOF</th>
<th>After fixing BOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Squid-2.3</td>
<td>if ((int) strlen(request-&gt;login) &gt; 0) { strcpy(loginbuf, request-&gt;login); //new bof case }</td>
<td>if ((int) strlen(request-&gt;login) &gt; 0) { strcpy(loginbuf, request-&gt;login, sizeof(loginbuf)); //Rule #1 }</td>
</tr>
<tr>
<td>Man-1.5h1</td>
<td>strcat(loginbuf, &quot;@&quot;); //reported</td>
<td>strcat(loginbuf, &quot;@&quot;); MAX_LOGIN_SZ + 1) strcat(loginbuf, &quot;@&quot;); //Rule #5</td>
</tr>
</tbody>
</table>

Table 11 shows the rules that when applied fixes the newly discovered BOF from three programs in Bugbench. We notice that Rule #1, Rule #3, and Rule #5 have been applied in Man-1.5h1, Polumorth-0.4.0, and Squid-2.3.

Table 11. New BOF cases in Bugbench benchmark.

<table>
<thead>
<tr>
<th>Program</th>
<th>Rule #1</th>
<th>Rule #3</th>
<th>Rule #5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Man-1.5h1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Polumorth-0.4.0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Squid-2.3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

5. Conclusion

Buffer Overflow (BOF) vulnerability is still widely detected in the wild across many legacy and newly developed applications implemented in C or C++ languages. Once BOF is detected, it needs to be patched. Understanding the complex code pattern resulting in BOF is the key step to developing the necessary patching of code without altering the intended functionality of applications. However, developing a set of generic rules and corresponding patching rules is challenging when BOF involves more than just one line of code such as pointer usage in a loop structure, and copying of subtype object into the allocated space of a supertype object. This paper addresses the patching of program code to remove BOF vulnerabilities without altering functionalities of the applications. The approach first identifies possible code patterns that usually result in BOF in C and C++ applications. Several code patterns are presented in this regard that involves programming elements responsible for BOF such as library function calls, buffer indexes, pointer arithmetic, and iteration pointers using loops. We proposed 12 patching rules to replace vulnerable code and remove BOF at the application unit level. Each rule corresponds to one of the identified elements causing BOF. We believe that these rules are generic enough to be applied to complex code having BOF. To apply the rules effectively, we also propose to augment parameter list of function calls that passes buffer as arguments. Fixing BOF by supplying the buffer’s length information is a novel approach based on this and related work.

Furthermore, we evaluated the proposed rules with two publicly available benchmarks with a number of C applications. The initial testing shows that the proposed rules for handling C code are able to fix the known BOF vulnerabilities. Moreover, some of the proposed rules fix previously unknown BOF cases that are complex in nature. The results indicate that the improper usage of library function calls through conditional statement, absence of null character assignments and incorrect loop conditions often lead to subtle BOF vulnerabilities that are difficult to identify. In addition, the proposed rules can be modified to become more context sensitive so that the rules do not add redundant code to the applications.

6. Future Work

The future goal is to formulate more rules to handle BOF vulnerabilities not covered in this work. These include conditional expressions of loop and if structures with user-defined function calls; pointer re-initialization problem; and code having goto structure. Moreover, future work will include developing automated tool support to apply the rules more efficiently.

The two benchmarks that we evaluated in Sections 4 do not include any C++ programs. As part of our future work and to complement our evaluation, we obtain three open source C++ applications randomly on the web to evaluate Rules #9 to #12. Table 12 shows the characteristics of the C++ programs with their description, LOC, and the corresponding rules applicable for them.

Table 12. C++ programs patching rules.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>LOC</th>
<th>Rule #9</th>
<th>Rule #10</th>
<th>Rule #11</th>
<th>Rule #12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calc</td>
<td>Calculator</td>
<td>259</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Account</td>
<td>An account management program</td>
<td>78</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bus</td>
<td>Bus reservation application</td>
<td>206</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The programs include a calculator, an account management application, and a bus reservation tool. Rule #9 is applied for all the programs. We have not found any instance of placement new expression used in these randomly selected applications, and hence Rules #10 to #12 are not applied. We keep this evaluation as future work subject to the availability of open source applications with the feature. Despite uncommon, the proposed patching rules provide C++ programmers a security concern to be dealt with during implementation should he/she plans to use placement new expressions and buffers accepting inputs as lines.

7. Acknowledgments

This work is supported in part by Kennesaw State University College of Science and Mathematics Faculty Summer Research Award Program and Faculty Startup Research Grant.

8. REFERENCES


3 http://www.cprogramming.com


ABOUT THE AUTHORS:

Dr. Hossain Shahriar is currently an Assistant Professor of Computer Science at Kennesaw State University, Georgia, USA. He completed his Ph.D. in Computing from Queen’s University, Canada. His research interests include software and application level security vulnerabilities and their mitigation techniques. Dr. Shahriar’s research has attracted a number of awards including the Best Paper Award in IEEE DASC, an Outstanding Research Achievement Award from Queen’s University, and an IEEE Kingston Section Research Excellence Award. Dr. Shahriar has published many research articles in Journals and conferences including ACM Computing Surveys, Journal of Systems and Software, and IEEE International Symposium of Software Reliability Engineering (ISSRE). He has been invited as a reviewer of international journals and PC member of international conferences on software and application security. Currently, he is serving as an associate editor of International Journal of Secure Software Engineering. Dr. Shahriar is a member of the ACM, ACM SIGAPP, and IEEE. More information about his research and background can be found at http://cs.kennesaw.edu/hshahria/.

Hisham Haddad holds a Ph.D. degree in Computer Science from Oklahoma State University, Stillwater, Oklahoma. Currently he is a Professor of Computer Science at Kennesaw State University, Georgia, USA. At Kennesaw State, he served as undergraduate CS Program Coordinator and Assistant Chair, and he teaches both undergraduate and graduate courses. He has been involved in the development of computer-based instructional tools for classroom teaching and he is active participant in ongoing curriculum development efforts. His research interests include Software Engineering, Object-Oriented Technologies, Software Reuse, and CS education. His work is published in professional journals and refereed International and National conferences. Dr. Haddad is active member of the professional community, regular participant in professional activities, and a member of several professional organizations. He is active participant in conference organization and a regular technical reviewer for many conferences and journals.

Ishan Vaidya is currently pursuing Master of Science in Computer Science degree from Kennesaw State University, Georgia, USA. He is a Graduate Research Assistant working with Dr. Hossain Shahriar on research projects including software security vulnerability mitigation and risk assessment techniques. Before joining Kennesaw State University, Ishan completed undergraduate and master degree in Information Technology from Charotar Institute of Technology (in affiliation with Gujarat Technological University), Gujarat, India. He is also a Software Developer Intern at Travelport located in Atlanta, Georgia.
Eliminating Redundancy in CSPs Through Merging and Subsumption of Domain Values

Chavalit Likitvivatanavong  
School of Computing  
National University of Singapore, Singapore  
chavalit@comp.nus.edu.sg

Roland H. C. Yap  
School of Computing  
National University of Singapore, Singapore  
ryap@comp.nus.edu.sg

ABSTRACT
Onto-substitutability has been shown to be intrinsic to how a domain value is considered redundant in Constraint Satisfaction Problems (CSPs). A value is onto-substitutable if any solution involving that value remains a solution when that value is replaced by some other value. We redefine onto-substitutability to accommodate binary relationships and study its implication. Joint interchangeability, an extension of onto-substitutability to its interchangeability counterpart, emerges as one of the results. We propose a new way of removing interchangeable values by constructing a new value as an intermediate step, as well as introduce virtual interchangeability, a local reasoning that leads to joint interchangeability and allows values to be merged together. Algorithms for removing onto-substitutable values are also proposed.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features—Constraints

General Terms
Algorithms, Theory

Keywords
Constraint Satisfaction Problems, Interchangeability, Onto-substitutability

1. INTRODUCTION
An important indicator of the hardness of a constraint satisfaction problem is the size of the search space. Eliminating interchangeable values was introduced in [7] as a way of reducing complexity of a problem by removing portions of the search space that are essentially identical. Recent focus on interchangeability has been on onto-substitutability [2, 6]: a domain value is onto-substitutable if any solution involving that value remains a solution when that value is replaced by some other value. Standard substitutability, by contrast, is a binary relation between two fixed elements.

In this paper, we redefine onto-substitutability as a binary relation and study its consequences. We propose joint interchangeability: two sets are joint-interchangeable iff any solution involving a value in one set remains a solution when that value is replaced by some value in the other set. Joint interchangeability is more practical since any one of the two sets can be eliminated; for onto-substitutability only a single value can be removed. We then propose virtual interchangeability: values are virtually interchangeable if they support the same values in every constraint but one. A set $S$ of virtually interchangeable values can be compactly represented by a value $s$, in effect making $S$ joint-interchangeable with $s$. Hence, virtual interchangeability leads indirectly to joint interchangeability.

To make sure virtually interchangeable set of values can be merged into a single value while retaining all the solutions, we expand the definition of domain value to accommodate the notion of label. A value may have more than one label, and labels are what actually appear as part of solutions. We introduce several new ways of comparing networks based on this concept. As a result, we can solve a CSP by transforming it using virtual interchangeability into a more compact network, solve the derived network, and convert it back — all without losing any solution. Moreover, this method works in the context of the hidden transformation, which is a way of transforming non-binary constraints into binary equivalents, by treating table constraints as a form of hidden variables. Preliminary results show that compressing all virtually interchangeable values is a promising approach to simplify table constraints in structured problems.

2. JOINT INTERCHANGEABILITY
Interchangeability and related ideas were first described in Freuder [7]. Bordeaux et al. [2] provided a formal framework that demonstrates connections among structural properties of CSPs. Removability, to which these properties reduce, is regarded by the authors to be the basis of how a value can be removed without affecting satisfiability of the problem. Freuder [6] later proposed the concept of dispensability: a value is dispensable if removing it will not remove all solutions to the problem. A value can then be dispensable without being removable (or onto-substitutable as called in [6]). Dispensability would therefore appear to be a more fundamental property than onto-substitutability. A survey of interchangeability concepts is reported in [9].
Onto-substitutability underscores the notion that a value’s attributes can be broken down and subsumed by other values. An onto-substitutable value can be removed without affecting satisfiability of the problem precisely because it is semantically redundant. For this reason, onto-substitutability is arguably key to understand many interchangeability concepts, but not as fundamental as dispensability when it comes to determine why a value can be removed.

Given the prospects of onto-substitutability, we will focus on this property and its derivations. First, we give the formal definition of CSP and redefine substitutability so as to make onto-substitutability a binary relation as follows. Any set of values mentioned in this section must be nonempty.

**Definition 1 (CSP).** A finite constraint network \( P \) is a triple \((\mathcal{X}, D, \mathcal{C})\) where \( \mathcal{X} \) is a finite set of \( n \) variables and \( \mathcal{C} \) a finite set of \( e \) constraints. \( D(X) \subseteq D \) represents the set of values allowed for variable \( X \in \mathcal{X} \). Each constraint \( C \in \mathcal{C} \) involves an ordered subset of variables in \( \mathcal{X} \) called scope (denoted by \( \text{scp}(C) \)).

**Definition 2.** Given a value \( v \in D(X) \), the maximally substitutable set of \( v \) is defined as \( \text{maxsub}(v) = \{ b \in D(X) \mid \text{there exists a solution involving } v \text{ which remains a solution when } v \text{ is substituted by } b \} \).

**Definition 3.** A value \( v \in D(X) \) is substitutable by a set of values \( S \subseteq \mathcal{X} \) if \( v \in \text{maxsub}(v) \) and any solution involving \( v \) remains a solution when \( v \) is substituted by some \( b \in S \). \( S \) is called a substitutable set of \( v \).

We similarly say \( v \) is substitutable if there exists \( S \) such that \( v \) is substitutable by \( S \). When \( |S| > 1 \), \( v \) is onto-substitutable and that \( S \) is an onto-substitutable set of \( v \). A substitutable set of \( v \) is minimal if there is no strictly smaller substitutable set of \( v \). Distinct minimally substitutable sets may exist for any given value. We also extend substitutability so that a set of values \( S \) is substitutable by \( T \) if for any value \( v \in S \), \( v \) is substitutable by \( T \).

Interchangeability can be redefined to cover many-to-many relationship in a similar fashion.

**Definition 4.** A set of values \( S \subseteq D(X) \) is said to be joint-interchangeable (JI) with a set of values \( T \subseteq D(X) \) if the set \( (S \setminus T) \cup (T \setminus S) \) is jointly substitutable by \( T \) and is substitutable by \( S \).

When \( |S| = 1 = |T| \), JI reduces to ordinary interchangeability. JI is symmetric and transitive. \( S \) is minimally joint-interchangeable with \( T \) if there exist no \( S' \subseteq S \) and \( T' \subseteq T \) such that \( S' \setminus T' \neq T \setminus S \).

**Example 1.** Consider a network involving two variables with solutions \( \{(1, a), (2, a), (3, b), (4, b), (5, c), (6, c), (1, d), (3, d), (5, d), (2, c), (4, e), (6, e)\} \). \{a, b, c\} is minimally joint-interchangeable with \{d, e\}.

Take note that the definitions of substitutability and JI allow the same value to appear on both sides of the relations. This helps us identify JI sets that would not otherwise be recognized. The following example illustrates.

**Example 2.** Consider a set of solutions \( \{(1, a), (2, a), (1, b), (2, c), (3, b), (3, c)\} \). \( a \) is not interchangeable with \( b \), but \{a, c\} is minimally joint-interchangeable with \{b, c\}.

**Proposition 1.** If \( S \) is joint-interchangeable with \( T \), then either \( S \setminus T \) or \( T \setminus S \) can be eliminated without affecting the network’s satisfiability.

In [2] the authors claim that local reasoning is not sound for onto-substitutability. Freuder [6] shows that this is not the case if the scope of “local reasoning” is broadened to include closure on sub-problems. The same rationale can be applied to joint interchangeability.

JI is a stronger than onto-substitutability but proves to be more useful. The latter lets us eliminate objects only from one side of the relation, whereas JI allows either side to be removed. Since a CSP with more values generally translates to longer search, once it is known \( S \) is JI with \( T \) an easy way to simplify the problem is to eliminate the larger set. Conversely, we want to identify two JI sets such that their size difference is as large as possible. We introduce and study a local reasoning which takes advantage of this fact called virtual interchangeability in §4.

Onto-substitutability can be weakened further by considering substitutability in only some solution.

**Definition 5.** A value \( v \) is nominally substitutable by \( b \) if there exists a solution involving \( v \) and it remains a solution when \( v \) is substituted by \( b \). A value \( v \) is said to be nominally substitutable when there exists such \( b \), that is, when \( \text{maxsub}(v) \neq \emptyset \).

Nominal substitutability and dispensability depend on the existence of a solution. Local reasoning such as closure is thus ineffective, because extending some solution in a bounded area to the whole problem is just as computationally difficult as finding a new solution from scratch. Nominal substitutability is equivalent to “minimal substitutability” in [6] and “context dependent interchangeability” in [16].

Recombination of values in the smallest closure was studied in [3] and it was shown that two values are nominally substitutable if their structure in the closure overlap and a fragment of their intersection involved is in a solution.
3. ENHANCING DOMAIN VALUES

Domain values serve two main purposes: structurally and semantically. These two aspects are intertwined in most CSP models, yet we would have more flexibility in manipulating a network when they are decoupled. To this end, we extend the definition of a value in this section. Combined with virtual interchangeability, this allows us to merge values while preserving all solutions at the same time.

Definition 6 (Labels). A value \( v \) of a constraint network \( \mathcal{P} \) is a tuple \((uid,lab)\) where \(uid\) is an identifier unique to this value in \( \mathcal{P}\) (also denoted by \(uid(v)\)) and \(lab\) is a set, which we also call the value’s labels. (also denoted by \(lab(v)\)). A value must have at least one label. Choosing a label for \( v \) from \(lab(v)\) is called an interpretation of \( v\).

The enhanced definition permits a value to be associated with multiple labels. Different values are allowed to share the same labels.

We consider two possible evaluations of labels when a value \(v\) is part of a network’s solution. In the first approach, every label is legitimate in any solution. Alternatively, only one of the labels is guaranteed to be legitimate. This leads to two new ways for comparing networks involving values with multiple labels: equivalence and conformity.

Definition 7. Given tuple \(t\) where \(scp(t) = \{X_1, \ldots, X_k\}\), \(sol(t) = \prod_{i=1}^{k} lab(t(X_i))\). Given networks \(\mathcal{P}\) and \(\mathcal{Q}\),
- \(\mathcal{P}\) and \(\mathcal{Q}\) are equivalent iff \(sol(\mathcal{P}) = sol(\mathcal{Q})\).
- \(\mathcal{P}\) subsumes \(\mathcal{Q}\) iff \(sol(\mathcal{P}) \supseteq sol(\mathcal{Q})\).
- for any \(t \in rel(\mathcal{P})\), if \(\exists s \in sol(\mathcal{Q})\) such that \(s \in sol(\mathcal{Q})\) then we say the interpretation of \(t\) is sound in \(\mathcal{Q}\), and that \(s\) is an interpretation of \(t\) in \(\mathcal{Q}\).
- \(\mathcal{Q}\) conforms to \(\mathcal{P}\) iff for any \(t \in rel(\mathcal{Q})\) there exists a sound interpretation of \(t\) in \(\mathcal{P}\).

An example of equivalent networks is shown in Figure 1.

Theorem 1. Given CSPs \(\mathcal{P}\) and \(\mathcal{Q}\), the following holds,
- if \(\mathcal{P}\) and \(\mathcal{Q}\) are equivalent then \(\mathcal{P}\) conforms to \(\mathcal{Q}\) and \(\mathcal{Q}\) conforms to \(\mathcal{P}\). The converse does not hold. That is, if \(\mathcal{P}\) conforms to \(\mathcal{Q}\) and \(\mathcal{Q}\) conforms to \(\mathcal{P}\), it is not necessary that \(\mathcal{P}\) and \(\mathcal{Q}\) are equivalent.
- if \(\mathcal{P}\) conforms to \(\mathcal{Q}\) and \(\mathcal{Q}\) conforms to \(\mathcal{P}\), then \(\mathcal{P}\) is satisfiable iff \(\mathcal{Q}\) is satisfiable.

Example 3. Consider network \(\mathcal{P} = (X,D,C)\), where \(X = \{X_1, \ldots, X_n\}\), \(D(X_i) = d\) and \(D(X_i) = \{a_{1i}, \ldots, a_{idi}\}\) for \(1 \leq i \leq n\), \(lab(a_{1i}) = \{l_{ij}\}\), \(C = \{C_1, \ldots, C_{n-1}\}\), \(scp(C_i) = \{X_i, X_{i+1}\}\) and \(rel(C_i) = \{(a_{ij}, a_{(i+1)j})\}\) for \(1 \leq i < n\), and \(\mathcal{P}' = (X', D', C')\), where \(X' = \{X_1', \ldots, X_n'\}\), \(D(X'_i) = \{a'_i\}\), \(lab(a'_i) = \{l_{ij}, \ldots, l_{idi}\}\), \(C' = \{C'_1, \ldots, C'_{n-1}\}\), \(scp(C'_i) = \{X'_i, X'_{i+1}\}\) and \(rel(C'_i) = \{(a_i, a'_{i+1})\}\). The networks are illustrated in Figure 2.

The following holds:
- (1) \(\mathcal{P}'\) subsumes \(\mathcal{P}\).
- (2) \(\mathcal{P}'\) conforms to \(\mathcal{P}\).
- (3) \(\mathcal{P}\) conforms to \(\mathcal{P}'\).
- (4) \(\mathcal{P}\) is satisfiable iff \(\mathcal{P}'\) is satisfiable.
- (5) \(\mathcal{P}\) is not equivalent to \(\mathcal{P}'\).

4. VIRTUAL INTERCHANGEABILITY

Identification followed by elimination has been an established method for dealing with redundant values in CSPs. This practice is simple and straightforward. As a result, much of the attention has been devoted to finding a new kind of interchangeability and developing more efficient algorithms that recognize these properties.

We present a proactive strategy with regard to properties that are binary relations. Given a relation \(R\), rather than the usual passive approach of finding two objects \(x\) and \(y\) such that \(xRy\) and eliminating one of the two, we need only identify \(x\), then create \(y\) from \(x\) so that \(xRy\), and finally eliminate \(x\). Central to this approach, however, is whether a “better” object \(y\) can be created from \(x\). Concretely for \(J\), instead of identifying two sets that are \(J\) with each other, we focus on finding a set of values that possess a certain property, then by exploiting that property create an equivalent value that represents that set more concisely, and only after the new value is added do we remove the original values.

In this section we introduce the concept of virtual interchangeability, a local reasoning that can be used as outlined above. We begin by recalling necessary definitions.

Definition 8. The projection of constraint \(C\) to \(S \subseteq scp(C)\)
is a constraint \( \pi_S(C) \) where \( \text{scp}(\pi_S(C)) = S \) and \( \text{rel}(\pi_S(C)) = \{ \{S\} \mid t \in \text{rel}(C) \} \). The projection of a tuple is defined in the same fashion. The concatenation of \( t_1 \in C_1 \) and \( t_2 \in C_2 \) (\( \text{con}(t_1, t_2) \)) is the tuple \( t \) resulting from the concatenation of \( t_1 \) and \( t_2 \) followed by rearrangement so that \( \text{scp}(t) = \text{scp}(t_1) \cup \text{scp}(t_2) \).

**Definition 9.** Given two values \( a, b \in D(X) \) and constraint \( C \) such that \( X \in \text{scp}(C) \), values \( a \) and \( b \) are neighborhood interchangeable with respect to \( C \) if and only if
\[
\{ t \in D \mid \text{con}(a, t) \in \text{rel}(C) \} = \{ t \in D \mid \text{con}(b, t) \in \text{rel}(C) \}
\]
where \( D = \pi_{\text{scp}(C)} \setminus \{X\}(C) \).

**Definition 10.** Two values \( a, b \in D(X) \) are neighborhood interchangeable (NI) \([7, 13]\) if and only if they are neighborhood interchangeable with respect to \( C \) for every constraint \( C \) such that \( X \in \text{scp}(C) \).

Given NI values, we can combine them into a single value without losing any solution simply by merging their labels into those of the representative value while discarding the remaining values. As a result, the initial network and the network after the NI values are merged are equivalent.

**Definition 11.** Two values \( a, b \in D(X) \) are virtually interchangeable (VI) (with respect to \( C \)) if there is at most one constraint \( C \) such that \( X \in \text{scp}(C) \) and \( a \) and \( b \) are not neighborhood interchangeable with respect to \( C \). A set of values are virtually interchangeable if any two values are virtually interchangeable with each other.

VI and NI are almost the same except for the difference of supports in a single constraint. NI implies VI but VI does not imply NI. Neighborhood Substitutability (NS) \([7]\) is incomparable to VI. A network that contains no VI values is called VI-free.

**Theorem 3.** Given \( a, b \in D(X) \) in \( P \), a new network \( Q \) can be derived from \( P \) by merging \( b \) into a as follows

1. updating \( \text{rel}(C) \) for any \( C \) involving \( X \) by altering any tuple \( t \in \text{rel}(C) \) where \( t[X] = b \) so that \( t[X] = a \)
2. setting \( \text{lab}(a) \cup \text{lab}(b) \) to be the new value of \( \text{lab}(a) \)
3. removing \( b \) from \( D(X) \)

Consequently,
- \( Q \) subsumes \( P \)
- \( P \) conforms to \( Q \) and \( Q \) conforms to \( P \)

And as a result of Theorem 1 and 2, it follows that,
- \( P \) is satisfiable iff \( Q \) is satisfiable.
- For any rendered solution \( s \) in \( P \) there exists a solution \( t \) in \( Q \) such that \( s \) is an interpretation of \( t \) in \( P \)

Theorem 3 tells us there always exists a sound interpretation of \( t \) in \( P \) for any \( t \in \text{rel}(Q) \). That is, the first solution found in \( Q \) will always lead to a rendered solution in \( P \), pending the interpretation. Interpreting a solution must be done only in the original network to maintain soundness and avoid spurious solutions.

**Theorem 4.** Assume the condition in Theorem 3 and in addition suppose that value \( a \) is virtually interchangeable with value \( b \), after merging \( b \) into \( a \) the network \( Q \) is equivalent to \( P \).

Theorem 3 and 4 show that VI values can be combined into a new value so that the original values and the new value are joint-interchangeable. A new network obtained by adding that new value and removing the original set will be equivalent to the original network. Because both networks are equivalent, the interpretation is search-free. The theorem can be extended to a group of VI values.

**Example 4.** Consider \( D(X) \) of the network in Figure 3 (top). Value labeled 0 is VI with value labeled 1, while value labeled 2 is VI with value labeled 3. The network on the bottom is derived according to Theorem 3. Solutions of the top are preserved in the bottom network, which include some spurious rendered solutions. For instance, consider the solution \( t = (2, (0, 1), 0) \) of the bottom network. An interpretation of \( t \) in this network itself would produce \((2, 0, 0)\), a rendered solution that does not belong to the original. However a sound interpretation of \( t \) in the top network exists as a result of Theorem 4. We pick a correct label of \( t[X] \) by inspecting the top network and find out which value in \( D(X) \) supports \( t[Y] \) and \( t[Z] \). In this case, label 1 is the answer.

Given a network, we can derive a more compact one by repeatedly applying Theorem 3. This involves going through each variable one by one and merging all VI values where possible. Results are propagated to the adjacent variables. The process terminates when no more VI values are detected. Propagation is necessary because values that are not initially VI may become so, once their neighbors are modified. Consider Example 3 for instance. In \( P' \) each domain has only one value, which is the best compression possible. Initially however, only the two variables at both ends (\( X_1 \) and \( X_n \)) can be compressed. No VI values are detected in the middle variables. After \( X_1 \) and \( X_n \) are compressed, the adjacent variables \( X_2 \) and \( X_{n-1} \) have to be re-examined in light of the change. Values of \( D(X_2) \) and \( D(X_{n-1}) \) then all become VI and will be merged as a result. The process continues until the propagation converges.

It is important to note that merging all VI values of a variable domain with respect to different constraints produces different results. Because a variable may contain different VI values that are VI with respect to different constraints, to reduce the size of the search space one heuristic is to pick some ordering so that the domain size becomes as small as possible when no VI values can no longer be found. In this paper, we consider only this greedy heuristic. Given a domain, we detect VI values by considering each involved constraint one by one and calculate the possible reduction in domain size. The constraint that gives the best reduction is chosen first for the application of Theorem 3. The reduction for each constraint is re-computed and the process is repeated until all VI values for this variable are merged.

In many cases, as Example 4 has shown, a sound interpretation can be deduced by looking at the instantiation of the neighboring variables in the original network, provided that they have singleton labels. But in general interpretation is not search-free, as shown in Example 3. When a solution
t ∈ rel(C) involves two values each having multiple labels, it is not simple to have a fast and easy way to determine a sound interpretation. However, the cost can be reduced to just a matter of simple look-up by sacrificing some compressibility. This is achieved by ensuring that no values with multiple labels can support each other by imposing the following restriction on Theorem 3 before b is merged into a:

**Proposition 2.** Consider the case for any constraint C such that \( X \in scp(C) \), where there is not any \( t \in rel(C) \) such that \( t[X] \) is either \( a \) or \( b \) and \( |lab(t[X'])| > 1 \) for some \( X' ≠ X \). In such cases, no propagation is necessary.

The broken-triangle property (BTP) [5] has a similar condition to VI in that it also forbids values having different sets of supports in two adjacent variables. Figure 4 shows that the BTP is not comparable to the VI-free property.

5. MERGING VI VALUES

We now present algorithms for finding and merging VI values of a single variable domain. Given a variable, all involving constraints are initially joined together into a single table constraint. The result is then compressed by considering each constraint in turn whether values are VI with regard to this constraint. We consider only variables involving binary constraints. Later, we describe how to compress non-binary constraints.

Given variable \( X \), Algorithm 1 collects all supports of values in \( D(X) \) and tabulates the results. Each row represents supports of a single value in \( D(X) \) from various constraints, and each cell contains the supports’ labels, in effect making the row, a cartesian product representation (CPR). The table can be viewed as partially compressed from the start.

Optionally, after supports are collected the table can be enumerated first so that each cell contains only a single label and each row becomes a simple tuple rather than a CPR. This may yield a better compression later although the side-effect on involving constraints can be more extensive. More importantly, however, Theorem 3 and 4 do not apply since both handle only the merging of domain values, not splitting them. Splitting and recombining domain values are beyond the scope of this paper, but it suffices to say that the resulting network does not necessarily subsume the original and thus some rendered solution may be lost, although conformity and satisfiability are unaffected. A case of missing rendered solutions will be given in Example 5.

Algorithm 2 details the greedy compression process. To compress a table, we hypothesically evaluate whether values are VI with respect to each column. The actual compression is performed with regard to the column that yields the best reduction (stopping at this point is denoted as the single-best compression process). The process is repeated until every column is committed. Constraints are updated using the finished table as the final step.

**Algorithm 1: Collect(X)**

\[\begin{align*}
c &:= \left\{ \{C \in C : \mathbb{X} \in scp(C)\} \right\}, \\
&\text{let } T \text{ be an empty table of } c \text{ columns;}
\end{align*}\]

**foreach** \( C_j \) such that \( X \in scp(C_j) \) do

**foreach** \( t \in rel(C_j) \) s.t. \( scp(C_j) = \{X, Y\} \) for some \( Y \) do

add \( lab(t[Y]) \) to cell \((t[X], j)\) of \( T;\)

**end foreach**

**end foreach**

until every column is committed.

**Proposition 3.** After all VI values are merged with respect to a certain constraint, any further merging of VI values in this domain with respect to the same constraint is not possible so long as neighboring domains are not altered.

**Proof** At time \( t_1 \) after all VI values are merged with respect to constraint \( C \), no two values are NI with respect to all constraints except \( C \). If at some later time \( t_2 \), there exist two values that are NI in all but \( C \), these two values must differ in at least one other constraint \( C' \) beside \( C \). Consequently, the difference in \( C' \) must be eliminated at some point between \( t_1 \) and \( t_2 \). The change in \( C' \) affects the neighboring domains, contradicting the assumption.

**Algorithm 2: Compress(C)**

\[\begin{align*}
\text{remainder} &:= scp(C), \\
\text{repeat} &\quad \text{until } \text{every column is committed . Constraints are updated using the finished table as the final step.}
\end{align*}\]

**Theorem 5.** The cost of \( \text{Compress}(C) \) is \( O(m^3 n \log n) \), where \( C \) is a table constraint of \( m \) columns and \( n \) rows.

**Proof** Line 5 takes \( O(n) \). It takes \( O(mn \log n) \) in line 6 to sort the projection from line 5. After sorting, counting the duplicate in line 7 is just a matter of going through each member of \( A \) while comparing the current and the previous member, costing \( O(mn) \). The whole process then takes \( O(n \sum_{i=1}^{m} imn \log n) = O(m^3 n \log n) \).

**Example 5.** We consider merging VI values of \( X \) in the network in Figure 3 (top). After supports are collected, the table is shown in Table 1(a). Tentative reduction size with respect to \( C_{XY} \) and \( C_{XZ} \) are both 2. The table is then compressed by merging VI values with respect to \( C_{XY} \). The result is shown in Table 1(b). Next, VI values with respect to \( C_{XZ} \) are combined, resulting in Table 1(c). No more VI values exist and this table will be used to update \( C_{XY} \) and \( C_{XZ} \), as depicted in Figure 3 (bottom). There are two values in the new \( D(X) \): one with labels \( \{0, 1\} \), and the other with \( \{2, 3\} \). Alternatively, one can choose to expand Table 1(a) first so that it would contain 9 tuples, each component of each tuple
having a single label. After compressing the table with respect to $C_{XY}$ and $C_{XZ}$ the result is shown in Table 1(d).

The original network has 9 rendered solutions whereas the compressed network in Table 1(c) has 2 solutions and 12 rendered solutions total, and the one in Table 1(d) has 2 solutions and 11 rendered solutions total. The network in Table 1(c) subsumes and conforms to the original. The network in Table 1(d) conforms to but does not subsume the original network. The tuple $t = (2, 2, 0)$, where $scp(t) = (X, Y, Z)$, exists in the original network but not in Table 1(d).

In addition to merging VI values in binary CSPs as discussed above, we can also use Algorithm 2 to compress tables in non-binary networks directly since we can think of a table as a dual variable in the hidden transformation [1], where each tuple in the table stands for a value in the dual variable’s domain. There is no need to invoke Algorithm 1.

**Proposition 4.** Given a non-binary constraint network $P$ where each value has a single label initially. Assume the network is converted into a binary network using the hidden transformation. Assume further that we merge all possible VI values in every variable domain, including dual and ordinary variables, and propagate as necessary, resulting in network $Q$. Given $t \in rel(Q)$ deciding whether interpretation of $t$ in $P$ is sound is search-free.

**Proof** There is no need for the interpretation of a dual variable because it does not exist in the original network. Instantiation of the dual variables determines the interpretation of the ordinary ones. Note that this reasoning is not valid for a mixed binary and non-binary network since there would be no intermediary dual variable between two original variables if there exists a binary constraint involving them. \(\square\)

**Example 6.** Consider the constraint in Table 2(a). Let us consider merging VI values of $H$, the hidden variable for this constraint. Tentative reduction size with respect to each constraint are: $C_{HX} = 3$, $C_{HY} = 2$, $C_{HZ} = 1$, $C_{HW} = 0$. Since the maximum reduction size is 3, we then proceed with merging VI values with respect to $C_{HX}$. Table 2(b) shows the result. At this point, we recompute the reduction size for each remaining constraint: $C_{HY} = 1$, $C_{HZ} = 0$, $C_{HW} = 0$. Table 2(c) shows the result after further merging with respect to $C_{HY}$. No more compression is possible. Since instantiation of hidden variables has no effect on actual solution, there is no need to have multiple labels for values in $H$. We can re-label $aceg$, $bf$, and $d$ to $1$, $2$, and $3$, for instance. Table 2(c) will be used to construct the hidden constraints $C_{HX}$, $C_{HY}$, $C_{HZ}$, and $C_{HW}$.

We note that although one can hypothetically view a table as a dual variable, but without actual dual variables present, the structural information of the combined tuples is lost. The CPR thus represents both syntax and semantic of the constraints at the same time. Consistency algorithms that normally work on tuples therefore must be modified to handle the CPR. On the other hand, the network that is explicitly transformed by the hidden transformation and compressed afterwards does not require any specialized algorithm.

**5.1 Experimental Results**

We now present some preliminary results on domain and table compression. We first compare results on random CSPs, which we expect to be unstructured, followed by structured problems. Results for series generated according to model RB are shown in Table 3. The processing time for each instance is negligible — a small fraction of a second at the worst. Enumerating each table before the compression yields almost no improvement. Despite the fact that only a few percentage of all values in a single instance are found to be VI, we do not find even a single pair of NI values in any instance tested. Interestingly, the number of affected variables is significant though the reduction is small, i.e. up to 27% of variables but with only 1.63% values merged. Since detecting NI values dynamically [8, 11] can improve their numbers, we can expect even more improvement for VI.

NS is not considered here because finding all NS values is too expensive and only two values can be checked at the same time. The number of pairs grows exponentially along with domain size. By contrast, algorithms for NI and VI can tackle the whole domain at once in polynomial time.

Table 5 displays the results for crossword puzzle $u\times vG^2$, which involves non-binary constraints of non-random structure: a constraint of arity $k$ contains all the words of length $k$. In contrast to the randomly generated binary problems, we see that compressing all VI values yields remarkable reduction

\[\text{All benchmarks are from http://www.cril.univ-artois.fr/CS009.}\]
rates, as high as 70% for the arity-4 constraint. At low arity, there is not much difference between the single-best compression and the greedy compression, but as the arity increases the reduction rate for the single-best drops sharply while the greedy compression maintains its rate well. The greedy compression can merge twice more values than the single-best compression at high arity.

We have also conducted some experiments to measure the running time of compressed instances from the series rand-3-20-20 and rand-5-30-5 using Abscon as a blackbox solver. Because newer solvers such as Abscon have implemented advanced GAC algorithms, we expect them to perform well on table constraints and relatively poorly on the hidden-transformed binary encoding, which requires more variables, more constraints, and much larger domains. We therefore compare the running time of the original problems against their hidden transformation problems that are simplified by the compression. The reduction rate ranges from 60% to 86%. Overall, the running time of the transformed problems is very competitive with that of the original — winning over half of all the instances from rand-3-20-20 and losing slightly on rand-5-30-5 for most instances — despite the obvious disadvantages of the hidden transformation. These results however are limited by the fact that the solver is not highly efficient.

### 6. IDENTIFYING REDUNDANT VALUES

We turn our attention to onto-substitutability in this section. An onto-substitutable value is redundant in the sense that every solution it participates in is also covered by some other value. We consider only algorithms that compute all onto-substitutable values in the smallest closure of a given variable in a network with extensional constraints.

To remove all onto-substitutable values from the domain of variable X, we do the following:

1. let D be the table created as a result of joining of all table constraints C involving X.

2. sort D in lexicographic order while ignoring column X.

3. merge cells of any two rows that differ only at col. X.

4. while there exists a value $v \in D(X)$ such that every cell containing $v$ also contains another value $v'$, remove $v$ from these cells and from $D(X)$.

An example is demonstrated in Table 4. Variable A is involved in two ternary constraints given as positive tables ((a) and (b)). After the two tables are joined and rows are merged, the result is shown in (c). Let us first consider value 0. In column A, 0 is never contained in a singleton set. Therefore 0 is onto-substitutable and we can remove it from the $D(A)$ and from all the sets in column A. Value 1 does not appear as a singleton as well; hence 1 is onto-substitutable. The set containing value 2 in the first row has become a singleton after the removal of 0, so 2 is not onto-substitutable. So are values 3 and 4 for the same reason. Therefore, values 0, 1, 3, and 4 are onto-substitutable and they can be removed from $D(A)$.

Note that the concept of onto-substitutability is tied to the structure of the network, therefore algorithms may produce different outcomes depending on the sequence of value removal. For instance, suppose value 5 is considered first, rather than value 0 as done in the previous example. Value 5 is onto-substitutable and it is removed. Next, values 2, 3, and 4 are examined and found to be onto-substitutable and removed as well. As a result, values 5, 2, 3, and 4 are found to be onto-substitutable in that order.

While this algorithm is simple and straightforward it has serious inefficiency in the process of joining of the constraints involved, which amounts to joining every constraint in a complete-graph constraint network for instance. We will consider an improved algorithm which places more computation on each table and operates on joined sub-tables only when necessary. Details are given in Algorithm 3.

We assume tables are initially merged as done in Table 4 (d) and (e) (from (a) and (b)). The algorithm decides whether a $\in D(X)$ is onto-substitutable. It first checks if there exist values that do not appear in all tables (line 3). These values cannot be joined and are filtered out. This step is equivalent to enforcing maxRPWC [12]. The next step tests whether

### Table 3. Results for random problems. The results for each series are averaged over 100 instances. #v is the no. of variables in each instance, #d the domain size, #c the no. of constraints, #t the tightness of each constraint, #vi the percentage of values merged via VI, and #va is the percentage of variable affected ($\geq 2$ VI values merged.)

<table>
<thead>
<tr>
<th>Instance</th>
<th>#v</th>
<th>#d</th>
<th>#c</th>
<th>#t</th>
<th>#vi</th>
<th>#va</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-10-5-15-658</td>
<td>10</td>
<td>5</td>
<td>15</td>
<td>0.658</td>
<td>2.58%</td>
<td>10.90%</td>
</tr>
<tr>
<td>2-20-3-30-519</td>
<td>20</td>
<td>3</td>
<td>30</td>
<td>0.519</td>
<td>4.38%</td>
<td>12.35%</td>
</tr>
<tr>
<td>2-20-20-25-909</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>0.909</td>
<td>1.58%</td>
<td>18.60%</td>
</tr>
<tr>
<td>2-50-3-120-367</td>
<td>50</td>
<td>3</td>
<td>120</td>
<td>0.967</td>
<td>2.54%</td>
<td>6.96%</td>
</tr>
<tr>
<td>2-50-5-70-683</td>
<td>50</td>
<td>5</td>
<td>70</td>
<td>0.683</td>
<td>2.59%</td>
<td>11.08%</td>
</tr>
<tr>
<td>2-100-4-200-500</td>
<td>100</td>
<td>4</td>
<td>200</td>
<td>0.500</td>
<td>2.05%</td>
<td>7.36%</td>
</tr>
<tr>
<td>2-100-10-110-877</td>
<td>100</td>
<td>10</td>
<td>110</td>
<td>0.877</td>
<td>1.95%</td>
<td>14.96%</td>
</tr>
<tr>
<td>2-200-100-220-985</td>
<td>200</td>
<td>100</td>
<td>220</td>
<td>0.985</td>
<td>1.63%</td>
<td>27.20%</td>
</tr>
</tbody>
</table>

### Table 4. Tables for constraints involving A.

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(d)</th>
<th>(e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>(0,2,3)</td>
<td>0</td>
</tr>
<tr>
<td>(0,1,4,5)</td>
<td>1</td>
</tr>
<tr>
<td>(0,2,3,5)</td>
<td>3</td>
</tr>
</tbody>
</table>
it is possible for $a$ to be subsumed. Values that appear as
singletons can never be onto-substitutable (line 3). This step
has been employed in the basic algorithm. The algorithm
may take this step first but it has to do it again after the
maxRPWC filtering because values that are not singleton
initially may become so later.

The algorithm then calculates the number of tuples involving
value $a$ (goal). This can be done by simple multiplication
without actually joining the tuples across tables. The upper
bound (ub) is the maximum number of tuples that are shared
by other values. Again, this can be easily computed and if
the number is lower then the value $a$ can never be subsumed
and the algorithm can terminate and give a negative answer.
Otherwise, the algorithm will try to enumerate all the tuples
that share $a$’s structure and see whether $a$ can be completely
covered (line 3) via the set store. Alternatively, this stage
can be implemented more compactly using tuple sequence [14]
marked by lower and upper bounds, instead of storing
the actual tuples themselves.

Example 7. Consider determining whether $0 \in D(A)$ from
Table 4 (a) and (b) is onto-substitutable. The algorithm
will check these four sets $\{0,2\}, \{0,1,4,5\}, \{0,2,4,5\}, \text{and}
\{0,2,3,5\}$ (the tuple involving $1$ has nothing to do with the
onto-substitutability of 0). First, value 1 is removed from
$\{0,1,4,5\}$ because there would be no tuple involving 1 after
joining the two tables. There is no singleton containing 0 so
the algorithm passes through the cutoff test (line 3). Next,
goal is calculated to be 4, while ub is the size of the CPR in-
volving value 2 (2) + value 3’s (1) + value 4’s (1) + value 5’s (2) = 6. The algorithm also passes through the cutoff test
(should the original two tables contain no tuple involving 2
and 5 the values of ub would be 2, for instance, and
false would be returned in this case). The algorithm then
adds the following tuples sequentially to store: $(0,0,2,2),
(0,0,3,3), (1,1,2,2), (1,1,3,3)$. At this point store’s size is
equal to goal so a is proved to be onto-substitutable.

Table 5. Results for crossword puzzles. Column #tuple
gives the original no. of tuples. The third col. shows the
no. of tuples left after the single-best compression, while
the fourth displays the reduction percentage. maxrate
gives the reduction rate for the greedy compression.

<table>
<thead>
<tr>
<th>arity</th>
<th>#tuples</th>
<th>#tuples left</th>
<th>reduction rate</th>
<th>maxrate</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4947</td>
<td>1727</td>
<td>66.69%</td>
<td>70.91%</td>
</tr>
<tr>
<td>5</td>
<td>10935</td>
<td>6205</td>
<td>43.27%</td>
<td>54.60%</td>
</tr>
<tr>
<td>6</td>
<td>18806</td>
<td>13759</td>
<td>26.84%</td>
<td>41.34%</td>
</tr>
<tr>
<td>7</td>
<td>27087</td>
<td>22156</td>
<td>18.20%</td>
<td>32.93%</td>
</tr>
<tr>
<td>8</td>
<td>32387</td>
<td>29558</td>
<td>8.73%</td>
<td>23.84%</td>
</tr>
<tr>
<td>9</td>
<td>32865</td>
<td>30420</td>
<td>7.44%</td>
<td>18.09%</td>
</tr>
<tr>
<td>10</td>
<td>29784</td>
<td>27699</td>
<td>7.00%</td>
<td>14.48%</td>
</tr>
<tr>
<td>11</td>
<td>23333</td>
<td>21897</td>
<td>6.15%</td>
<td>12.76%</td>
</tr>
<tr>
<td>12</td>
<td>16917</td>
<td>15944</td>
<td>5.75%</td>
<td>11.67%</td>
</tr>
<tr>
<td>13</td>
<td>11246</td>
<td>10179</td>
<td>4.42%</td>
<td>10.31%</td>
</tr>
<tr>
<td>14</td>
<td>6998</td>
<td>6745</td>
<td>3.62%</td>
<td>10.02%</td>
</tr>
<tr>
<td>15</td>
<td>3962</td>
<td>3839</td>
<td>3.10%</td>
<td>9.36%</td>
</tr>
<tr>
<td>16</td>
<td>2009</td>
<td>1962</td>
<td>2.34%</td>
<td>7.72%</td>
</tr>
</tbody>
</table>

Onto-substitutability has been shown as a sufficient condi-
tion for a value to be treated as redundant. We extend its
definition from unary to binary relation and introduce JI as
a replacement for interchangeability. As a symmetric binary
relation, JI allows us to remove either one of the interchange-
able set of values, giving us more flexibility as a result. We
then present an original strategy for dealing with redundant
values: detection, creation, followed by elimination. Since
JI is a binary relation, this strategy makes sense: we can
identify a group of values, create a JI-equivalent using fewer
number of values, and then delete the original values.

The definition of values is expanded to include the concept
of labels, which allows us to tease out semantics from the
structure of the network. Different CSPs can be compared
solely on their semantics (via their rendered solution). We
then introduce VI, a new local reasoning that leads to JI.
While it remains to be seen whether future work in this area
will give us a new local property that also leads to JI, we
have empirically shown the promise of VI as a compression
tool. Table constraints can be compressed using other tech-
niques such as decision trees [10], but they require special-
ized consistency algorithms unlike VI. VI may prove useful
in different situations as well, for instance, VI could be used
in addition to NS as a simplification operation before rea-
soning with CSP patterns [4].

In [2], the authors raise "an interesting open issue: do there
exist new (i.e., other than substitutability and inconsistency)
properties for which local reasoning is sound and which im-
ply removability?" We believe VI is one such property, pro-
vided that creation of new values is permitted. Allowing
networks to be augmented in this manner could also lead to
a more powerful general framework.

8. ACKNOWLEDGMENTS

This work has been supported by the Ministry of Education
under Grant No. MOE2012-T2-1-155.

9. REFERENCES


ABOUT THE AUTHORS:

Chavalit Likitvivatanavong is currently a research associate in the School of Computing of the National University of Singapore. He obtained his M.Sc. in Computer Science from Purdue University, U.S. Prior to his current position, he was a research scientist at the Cork Constraint Computation Centre, University College Cork, Ireland, where he continues to pursue his Ph.D.

Roland H. C. Yap is an associate professor in the School of Computing of the National University of Singapore. He obtained his B.Sc. (Honors), M.Sc and Ph.D. degrees in Computer Science from Monash University, Melbourne, Australia. Before joining the National University of Singapore, he was an Australian Research Council research fellow at the University of Melbourne. Prior to that, he had been a visiting scientist at the IBM Thomas J. Watson Research Center in New York.
A Framework for Processing Complex Queries in Wireless Sensor Networks

Jose E. Bessa Maia  
State University of Ceara  
Fortaleza - Ceara - Brazil  
jose.maia@uece.br

Angelo Brayner  
University of Fortaleza  
Fortaleza - Ceara - Brazil  
brayner@unifor.br

Fernando Rodrigues  
University of Fortaleza  
Fortaleza - Ceara - Brazil  
fernandorodrigues@edu.unifor.br

ABSTRACT

In this work, we present a framework, denoted ADAGA – $P^*$, for processing complex queries and for managing sensor-field regression models. The proposed mechanism builds and instantiates sensor-field models. Thus ADAGA – $P^*$ makes query engines able to answer complex queries such as \textit{give the probability of rain for the next two days in the city of Fortaleza}. On the other hand, it is well known that minimizing energy consumption in a Wireless Sensor Network (WSN) is a critical issue for increasing the network lifetime. An efficient strategy for saving power in WSNs is to reduce the data volume injected into the network. For that reason, ADAGA – $P^*$ implements an in-network data prediction mechanism in order to avoid that all sensed data have to be sent to fusion center node (or base station). Thus, sensor nodes only transmit data which are novelties for a regression model applied by ADAGA – $P^*$. Experiments using real data have been executed to validate our approach. The results show that ADAGA – $P^*$ is quite efficient regarding communication cost and the number of executed float-point operations. In fact, the energy consumption rate to run ADAGA – $P^*$ is up to 14 times lower than the energy consumed by kernel distributed regression for an RMSE difference of 0.003.\(^1\)

Categories and Subject Descriptors

H.2 [Database Management]: Miscellaneous

General Terms

Performance, Estimation, Validation

Keywords

Wireless sensor networks, Query processing, Data prediction

1. INTRODUCTION

Sensors are resource-constrained devices used to collect data from the environment aiming at detecting or measuring physical phenomena. Advances in wireless communication have enabled the development of massive-scale wireless sensor networks (WSN). In a WSN, sensors are usually scattered in the network and use low-power communication channels. In a WSN, sensed data are passed from sensor to sensor until a special node, denoted Fusion Center (FC), is reached. A Fusion Center (also called base station) is characterized by having robust disk storage, unlimited energy power and good processing capacity. Potential WSN applications include environmental monitoring (e.g., traffic, habitat), industrial sensing and diagnostics (e.g., factory supply chains), infrastructure protection (e.g., water distribution), battlefield awareness (e.g., multitarget tracking) and context-aware computing (e.g., intelligent home) [3].

The critical point in WSNs is energy consumption [7]. The largest part of energy consumption in a sensor node occurs during data transmission or reception. For that reason, the main goal of most algorithms designed for WSN applications is communication cost reduction in terms of energy consumption [3, 4, 6, 15, 23]. For example, the use of in-network aggregation operators [4, 15, 23] is an efficient strategy to reduce the volume of data transmitted in a given WSN. However, aggregation operators do not give a faithful representation of the original structure of sensed data. To overcome this problem, an efficient strategy for reducing the data flow in a WSN is building models for sensor-field in the FC. With such models, query engines may predict and interpolate data instead of requiring them from the network.

The execution of instructions by the processing unit of sensor nodes consumes energy as well [16]. In this sense, there may exist strategies for reducing energy consumption in WSNs, which requires the execution of a huge amount of instructions (e.g., floating-point operations), in such a way that the overall energy consumption reduction becomes insignificant. This is the case of the approaches proposed by Guestrin et al [9] and Hou et al [11, 10] as we show in Section 5.1.

In this work, we present an engine, denoted ADAGA – $P^*$, for managing sensor-field models. The idea is to provide query engines (running upon ADAGA – $P^*$) the ability of answering complex queries in WSNs. We consider a query $Q$ as complex if, and only if, it is necessary to use sensor-field models to process it. To illustrate the notion of complex query, consider the following query $Q$: \textit{give me the probability of rain for the next two days in the city of Fortaleza.} To process $Q$ ADAGA – $P^*$ applies regression models for predicting temperature, pressure and humidity values for the next 2 days. Afterward, it merges those models to construct a model which is able to give the rain likelihood for the next 2 days. ADAGA – $P^*$ can be used by applications which require a data prediction mechanism for predicting data that will be actually sensed later (in the future), such as weather forecast applications.

Another sort of applications which motivates the use of approach proposed in this paper is top-$k$ query processing [22]. In general, a top-$k$ query returns the $k$ items having largest (or lowest) global

\(^1\)Copyright is held by the authors. This work is based on an earlier work: SAC’13 Proceedings of the 2013 ACM Symposium on Applied Computing. Copyright 2013 ACM 978-1-4503-1656-9/13/03. http://doi.acm.org/10.1145/2480362.2480468.
score in a distributed system. ADAGA – \( P^* \) can be applied to process top-k queries, since largest or lowest values can be found from the instantiation of a sensor-field model. It is worthwhile to mention that ADAGA – \( P^* \) is capable to process conventional queries in WSN as well, such as give me the maximum, minimum and average temperature values in region \( R \) [3].

The gist of ADAGA – \( P^* \) is to construct sensor-field models based on regression (linear, nonlinear, kernel, etc) in regular time intervals \( T_k \). The models are built to attend specific application requirements. Thus, there may be different models for a single physical phenomenon (e.g., temperature or pressure). After building an efficient model \( m \), ADAGA – \( P^* \) can instantiate it by applying \( m \) to a given time series, generating a time sequence of sensor-field models. This way, the proposed mechanism makes query engines able to process complex queries which require forecasting, interpolation or inference.

Differently from the kernel distributed regression approach proposed in [9, 10, 11], ADAGA – \( P^* \) constructs and instantiates sensor-field models in the fusion center. Such a feature saves power in sensor nodes, since they do not have to consume energy for computing the sensor-field model. The kernel distributed regression approach reduces the data volume flowing in the network by performing in-network sensor-field regression based on spatial correlation (of sensed data). Thus, only model coefficients are sent to the fusion center. Notwithstanding, for computing the sensor-field model, the number of floating-point operations (FPO) executed by kernel distributed regression is of one magnitude order higher than the number of FPO executed by ADAGA – \( P^* \) (see Section 5).

Albeit ADAGA – \( P^* \) constructs sensor-field models in the fusion center, it does avoid to send all sensed data to fusion center. Such a feature stems from the fact that ADAGA – \( P^* \) implements an in-network data prediction mechanism. The implemented prediction mechanism relies on a linear regression model for time-series data yielded by sensors belonging to a WSN. Nevertheless, other prediction models can be used, such as polynomial regression or spline. Additionally, ADAGA – \( P^* \) introduces the property of adjustable accuracy for the proposed prediction strategy. The idea is to allow users to specify a “tolerable difference” \( r \) between the sensed value \( s \) and the predicted value \( p \). Thus, a sensor node transmits a sensed value \( s \) if, and only if, \( s \) is not within the “tolerable difference” \( t \), i.e., \( s \not\in [p-t, p+t], t \geq 0 \). Data outside the tolerable difference are treated as novelty for the model.

ADAGA – \( P^* \) implements two different versions of the in-network data prediction mechanism. One version is based on an aggressive approach for predicting data. In the aggressive version of ADAGA – \( P^* \)’s in-network data prediction mechanism, as soon as a sensor node identifies a sensor reading \( r \) for which the absolute difference between \( r \) and the prediction value is greater than the value specified in the threshold clause, the sensor node transmits \( r \) to the sink node, which in turn recalculates regression equation coefficients. That sensed data \( r \) may represent in fact a new tendency for the values which are being collected from the environment. In this case, it is reasonable to update the prediction model promptly. The other version is based on a conservative approach. In this version ADAGA – \( P^* \) postpones the moment for updating regression model. Thus, ADAGA – \( P^* \) is able to identify and differentiate new tendency in sensed-data time series and outliers.

For empirically validating the proposed approach, we have compared results produced by ADAGA – \( P^* \) with those produced by kernel distributed regression [9] w.r.t. the following metrics: communication cost, reconstruction RMS error in the measuring points, and required computation (number of floating-point operations) in sensor nodes. The results show that the energy consumption reduction rate achieved by ADAGA – \( P^* \) is 15 times higher than the rate obtained by kernel distributed regression for an RMSE difference of 0.003.

The remaining sections are organized as follows. Basic concepts on WSN and kernel distributed regression are presented in Section 2. In turn, Section 3 analyzes approaches to build prediction models. In Section 4, we describe and discuss ADAGA – \( P^* \) and two different versions of the in-network data prediction mechanism implemented by it, one version is based on an aggressive approach for predicting data and the other version exploits a conservative approach. Section 5 presents the results obtained from experiments. For that, ADAGA – \( P^* \) and the aggressive approach of its data prediction mechanism have been evaluated using real data. For validating ADAGA – \( P^* \) and the conservative approach, simulations with a prototype implemented in Sinalgo [8] have been executed on real temperature, humidity and luminosity data made available by the Intel Berkeley Research Laboratory [12]. Section 6 brings a discussion on how the proposed approach cope with packet loss, sensor failure, scalability and sensor node synchronization. Finally, Section 7 concludes the paper.

2. BASIC CONCEPTS

Figure 1 illustrates an abstract WSN model topology assumed in this work. In this model, a WSN can be composed of three types of sensor nodes: (i) leaf nodes which are responsible for sensing, transmitting sensed data and receiving and forwarding data from other sensors; (ii) sink nodes which aggregate and predict sensed data, besides sensing data, and; (iii) a Fusion Center (or base station) which is characterized by not having battery or processing capacity restrictions.

![Figure 1. WSN topology.](image-url)

In our approach, users or applications submit complex queries to the query processor which is the interface between users or applications and ADAGA – \( P^* \) (see Figure 1). Tasks such as data fusion, interpolation, estimation and prediction are performed by ADAGA – \( P^* \) in FC. ADAGA – \( P^* \) is not query-engine specific, i.e. it can be deployed on any query engine. Nonetheless, to run the...
Consider a network of $N$ sensor nodes that are distributed over the interest region $\Omega$, where $\Omega \subset R^2$. Each sensor measures a physical phenomenon modeled by a function $f(x, t)$. Let $D$ be the set of $S$ sensor measurements. Thus, the task of data modeling is to determine a function $\hat{f}(x, t)$ to fit sensor measurements in $D$. The region $\Omega$ can be decomposed into $L$ overlapping regions $\{\Omega_j\}_{j=1}^L$, where $\bigcup_{j=1}^L \Omega_j = \Omega$. Each $\Omega_j$ can be modeled by a basis function set $H^j = \{h^j_1, ..., h^j_{d_j}\}$, where $d_j$ is the number of sensors in $\Omega_j$. The local fitting function in $\Omega_j$ is defined $\hat{f}_j(x, t) = \sum_{i=1}^{d_j} w^j_i h^j_i(x, t)$. Now, let $w_j = [w^j_1, w^j_2, ..., w^j_{d_j}]$. With this notation, the global fitting function $\hat{f}(x, t)$ is the weighted sum from all the local models:

$$\hat{f}(x, t) = \sum_{j=1}^L k_j(x) \hat{f}_j(x, t).$$

Here, $k_j(x)$ represents the correlation degree of location $x$ associated with $\Omega_j$, and it is defined continuous and positive on $\Omega_j$ and zero elsewhere. The $k_j(x)$ are the kernel weight functions and sum to one for each $x \in \Omega$. Hence, the goal is to seek the weighted coefficients $w = [w^j_1, w^j_2, ..., w^j_{d_j}]^T$ to minimize the sum of square errors,

$$w^* = \min_{w} \|Hw - f\|^2,$$

where $f \in R^S$ is arranging all sensor measurements into a column, and $H = [H_1, H_2, ..., H_L] \in R^{S \times M}$ is an $S \times M$ matrix, where $\sum_{i=1}^L d_i$, and each submatrix $H_i$ is an $S \times d_i$ matrix. Each row vector $r_j$ in $H_i$ is taken from a sample point say $(x, t)$ in $D$, i.e.,

$$r_j = k_j(x) \times [h^j_1(x, t_1), h^j_2(x, t_1), ..., h^j_{d_j}(x, t_1)].$$

Setting the gradient of this quadratic objective to zero gives the optimal coefficients:

$$w^* = (H^T H)^{-1} H^T f = A^{-1} b,$$

where $A = H^T H = [A_{i,j}] \in R^{M \times M}$ and $b = H^T f = [b_i] \in R^M$. Notice that the local models are coupled and cannot be directly locally solved.

In a sequential process, such as temperature measurement over time, it is necessary to fit a regression model using a sliding window. That is, the coefficients of basis functions have to be fit w.r.t. the measurements performed in the last $T$ minutes. The equations for this update procedure can be found in [9]. Moreover, a distributed implementation of the algorithm requires the updating of partial matrices and vectors in each region using received data from the regions with it has overlapping. The detailed equations for this can be found in [10].

3. RELATED WORK

Ren and Liang present in [18] an approach for automatically selecting the WSN node for processing a given query regarding two complementary criteria: energy consumption and data quality. The idea is to probabilistically guarantees user-defined error bound, given a confidence interval on query accuracy, while reducing the energy consumption. Although the authors claimed that their approach adaptively builds an optimal query plan, they did not present any query engine capable of building and processing query execution plans. Additionally, the paper provides no systematic empirical assessment evidencing that the proposed approach can indeed reduce the levels of energy depletion in real or simulated WSN.

In [14], the authors propose a strategy for saving energy in WSNs by exploiting data spatio-temporal correlation. Thus, the FC node partitions sensor nodes with similar measured values into clusters. Sensor nodes within a cluster are scheduled to work alternatively in order to reduce energy dissipation. Differently from ADAGA – $P^*$, the proposed strategy in [14] is not scalable since it requires all nodes in the WSN to be in direct transmission range of the FC.

Jiang et al propose in [13] the use of a multiple linear regression model for predicting data in WSNs. In that approach, sensor nodes are responsible for checking the approximation in an adaptive way. Similarly, in [5] the authors propose a method based on multivariate spatial and temporal correlation of sensed data to improve data prediction accuracy in WSNs. However, both approaches do not take into account the amount of FPOCs executed at each sensor node, which is not the case of ADAGA – $P^*$.

Guestrin et al [9] propose a distributed Gaussian Elimination algorithm to solve the regression algebraic equation. For that, an asynchronous message passing protocol and a junction tree based routing structure are employed in an in-network fashion. However, as shown in [10] there are several drawbacks in this approach. Firstly, only one hop for asynchronous message exchanging among neighboring nodes will inevitably result in slow convergence. Secondly, the network topology is organized as a large flat structure. For that reason, it will be much costly for reconfiguring the global junction tree if some links fail in unstable networks. Finally, since several nodes do not participate for solving the regression problem, there is redundant communication activity, especially, for those nodes with the same associated clique and factor set.

To overcome the aforementioned drawbacks presented by the approach proposed by Guestrin et al [9], Hou et al [10, 11] introduce the idea of using a skeleton tree for processing the distributed Gaussian elimination algorithm. The distributed Gaussian elimination algorithm is applied to solve regression algebraic equations. The key idea behind the notion of skeleton tree is to combine the clustering based routing mechanism with message passing protocol based on junction tree.

Compared with the strategy of building a junction tree over all nodes [9], the skeleton-tree based routing protocol suppresses the redundant message exchanging among the ramose nodes and cluster heads without deteriorating data modeling precision. In [11, 10], the authors show that the routing scheme proposed by them can save about 33% communication cost w.r.t. the approach proposed by Guestrin et al.

On the other hand, the approaches proposed by Guestrin et al [9] and Hou et al [10, 11] present the same energy consumption rate regardless the sensor field varies rapidly or not. In this sense, they propose as future work an adaptive mechanism for adjusting system parameters such as region scope, time window length and the number of basis functions or kernels. Guestrin et al and Hou et al argue that such an adaptive mechanism make their approaches able to take advantage whenever the sensor field varies more slowly in order to reduce the energy consumption rate by decreasing the
amount of FPOs and message exchange. Notwithstanding, the approaches presented in [9] and [11] still require the execution of an amount of FPOs which is of one magnitude order higher than the number of FPO executed by ADAGA – $P^*$.

Villas et al proposes an algorithm, called EAST[21], for using both spatial and temporal correlations to perform near real-time data collection in WSNs. According to the authors, sensor nodes are grouped into event areas and correlation regions according to spatial correlation. Leader and the representative nodes perform a temporal suppression technique, where the value of a new reading from a representative node is only sent to the sink node when the difference between this value and the last value sent exceeds a certain threshold, denoted temporal coherency tolerance (tct, for short). The proposed strategy considers that all sensor readings, temporarily serial, from a particular correlation region (or representative node) will have the same value, within the error threshold. This is because new readings are considered as new values to be sent to the sink, only if the new value read ($R_{new}$) differs from the previous ($R_{old}$) if the difference is greater than the error threshold (tct).

ADAGA – $P^*$ naturally incorporates the feature of taking advantage whenever the sensor field is varying more slowly. For instance, whenever the sensor field temperatures varies smoothly for a long period, the prediction model becomes more efficient (i.e., yields more precise predictions). Consequently, ADAGA – $P^*$ automatically transmits less data.

To conclude this section, we want to emphasize that the in-network prediction strategy proposed in this paper is more efficient than the aforementioned approaches w.r.t. energy consumption. This is because: (i) message exchange among nodes is not necessary, (ii) the amount of FPOs executed at each sensor node is significantly smaller, and; (iii) the data volume injected into network is automatically reduced case the sensor field varies more smoothly.

4. PROCESSING COMPLEX QUERIES
IN WSN

As already mentioned, we propose an engine, called ADAGA – $P^*$, for managing sensor-field regression models. The idea is to provide a query engine the ability of answering complex queries in WSNs. Thus, the query engine can deliver answers to queries such as provide temperature values for the next 3 days. ADAGA – $P^*$ builds and instantiates sensor-field models in FC. Nonetheless, it implements a prediction mechanism based on a linear regression model. This way, the data volume sent to FC through the network can be diminished. In this section, ADAGA – $P^*$ engine is described. Two different versions of the prediction mechanism implemented by ADAGA – $P^*$ are described, one of them implements an agressive approach for predicting data and the other one exploit a conservative approach.

4.1 Complex-Query Engine

In order to process complex queries in WSNs, we propose a mechanism, called Complex-Query Engine (for short, CQE), which is composed of a query engine and ADAGA – $P^*$ engine. Thus, the proposed CQE may process conventional and complex queries in WSNs. For this work we have used the query engine proposed in [3] to implement CQE. However, ADAGA – $P^*$ can be integrated with any other query engine, which is able to process queries in WSNs. Figure 2 depicts an abstract model for the structure of the proposed CQE.

The ADAGA – $P^*$ engine implements two key functions:

- **Sensor-field model building.** Regression-based sensor-field models are built to attend application-specific requirements. In this sense, it is necessary to have a model database. This way, users may pick a model up, which is more adequate to their applications. For example, there exist classes of applications for which simple linear regression model is enough; On the other hand, for another classes of application, complex regression models are more adequate (e.g. kernel regression or multiple regression);

- **Model instantiation.** This function has the goal of providing to query engines data produced by applying a sensor-field model (from the regression-model database). Thus, the query engine can process queries requiring future data, for instance, temperature, humidity and pressure for the next 24 hours.

Consequently, CQE should manage a regression-based model database, called RM database, and an instance database, denoted IRM database (see Figure 2). The RM database has the functionality of storing regression models. For example, the RM database may store linear regression models and kernel-based regression models. In turn, the IRM database stores instances for the models in the RM database. In other words, the IRM database stores data and parameters produced by applying a regression-based model to execute a complex query for a given application (or user).

To illustrate the usage of the RM and IRM databases, consider that the RM database stores a linear regression model, called LR, and a kernel based distributed regression model, denoted KDR. Thus, for a given data series $t$, the IRM database may store: (i) an instantiation $I_1$, produced by applying the linear regression model $LR$ on the data series $t$, and; (ii) an instantiation $I_2$ yielded by applying the kernel distributed regression model $KDR$ on $t$.
the number of floating-point operations (FPO) executed by kernel
distributed regression is one magnitude order higher than the num-
ber of FPO executed by ADAGA – $P_e$. Moreover, in our approach,
the number of data injected into the network is reduced. This is
because ADAGA – $P_e$ implements an in-network data prediction
mechanism based on temporal correlation of sensed data. As a re-
result, data which can be predicted by the FC within the threshold
error are not transmitted (see Section 4.2).

Therefore, we claim that ADAGA – $P_e$ is more efficient than the
distributed regression approaches proposed in [9, 10, 11] w.r.t. en-
ergy consumption diminution rate. The results presented in Section
5 prove that our claim is true.

Since we are applying a query engine which processes queries ex-
pressed by means of SNQL, we have extended the SNQL’s from
clause to make possible the specification of which model (stored in
the RM database) and the instantiation of that model (stored in the
IRM database) should be used to process a given complex query.

Table 1 presents an example of a complex query expressed in SNQL.
The query depicted in Table 1 returns the predicted temperature val-
ues for each sensor. The model used to predict the data is temperature
(stored in RM database, see Figure 2) whose instantiation is $I_t$
(stored in IRM database). The query should predict temperature
values for the next 24 hours (time window clause) and the values
should be predicted at the rate of one value per sensor per hour
(sense interval clause).

<table>
<thead>
<tr>
<th>Table 1. SNQL complex query.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT t.predicted-values</td>
</tr>
<tr>
<td>FROM MODEL.temperature $&lt; I_t &gt;$ as t</td>
</tr>
<tr>
<td>TIME WINDOW 86400 SENSE INTERVAL 3600</td>
</tr>
</tbody>
</table>

In the next sections, two approaches of the in-network data pre-
diction strategy implemented by ADAGA – $P_e$ are described and
discussed. As already mentioned, their main goal is to reduce the
data volume sent to the FC, since ADAGA – $P_e$ needs sensor data
to build sensor-field models.

4.2 In-Network Data Prediction Mechanism: Aggressive Approach

ADAGA – $P_e$’s in-network prediction mechanism requires that sen-
sor nodes know the limit in the difference between the acquired
value and the estimated value. To dynamically “inform” a sen-
node about that difference, a new clause was added to SNQL
[3], called threshold (see Table 2) [17]. The sensor does not inject
sensed data into the network if the threshold value is not violated.
Thus, it is enough to use the following formula: $P \geq 100(V_c - V_p)/V_c$, where $P$ represents the percentage value specified in the
threshold clause, $V_c$ corresponds to the value collected by the sen-
node node, and $V_p$ is the predicted value.

<table>
<thead>
<tr>
<th>Table 2. Threshold Clause added to SNQL.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clause</td>
</tr>
<tr>
<td>------------------------------------</td>
</tr>
<tr>
<td>THRESHOLD $&lt; snd\text{\ percent} &gt;$</td>
</tr>
<tr>
<td>THRESHOLD $&lt; \text{snd\ percent} &gt;$:</td>
</tr>
</tbody>
</table>

Therefore, in ADAGA – $P_e$ data accuracy can be dynamically spec-
ified by users through the threshold SNQL clause. The idea is to
give to users/applications the decision of reducing even more en-
ergy consumption. The threshold parameter is defined for each
query submitted to ADAGA – $P_e$. We are assuming that the user
is a specialist and for that reason he/she has the expertise to decide
for the best threshold value having in mind that for smaller thresh-
old values more energy is consumed and more precise the results
are. The threshold set to 0 means that the in-network prediction
mechanism is not used.

In the proposed prediction strategy, the linear regression equation
(prediction equation) is sent to all leaf nodes belonging to a WSN.
It is important to observe that, if a leaf node does not receive the
prediction coefficients, it has a chance to receive the coefficients
values again when it sends a new set of novelties to the sink node.

The in-network prediction implemented by ADAGA – $P_e$ is based
on the following equation: $\hat{S}(t) = a + bt$. The time $t$ is an inde-
pendent variable. $\hat{S}(t)$ represents the estimated value of $S(t)$ and is
variable with $t$. Parameter $a$ is the interceptor-t (value of $\hat{S}(t)$ for
$t = 0$) and $b$ is the stretch slope, and are computed as follows:

$$a = \frac{1}{N} \left( \sum S_i - b \sum S_i \right) = \bar{S} - b\bar{t},$$

$$b = \frac{\sum (t_i - \bar{t}) (S_i - \bar{S})}{\sum (t_i - \bar{t})^2}. \quad (6)$$

ADAGA – $P_e$’s prediction mechanism requires that each sink node
(see Figure 1) executes the steps described below and in the follow-
ing chronological order:

1. The sink node receives data from sensor nodes in its region
to compute the coefficients $a$ and $b$;

2. Each sink node sends the coefficients to all sensors belonging
to its region. In turn, sensor nodes start to compare collected
data with predicted data, checking the threshold error. A sen-

sor transmits data only when the absolute difference between the
sensed value and the prediction value is greater than the value
specified in the threshold clause; and

3. Since each sink node is responsible for a sub-network (re-

gion) of sensors, if any value sent by a sensor reaches a sink
node, the latter uses the received value and recalculates reg-
ression equation coefficients. Thus, prediction accuracy is
automatically and continuously adjusted.

Figure 3 illustrates the proposed prediction mechanism behavior
for the query depicted in Table 1. To plot the curves depicted in
Figure 3, a set of 38 real temperature values (collected by a sensor
node) has been used, with a threshold of 5%.

The discontinuities in predicted series curve in Figure 3 represent in
fact points where the predicted values violate the tolerable thresh-
hold. This is the case when $t=511$. One can observe that predicted
data do not violate the threshold of 5% within the interval of $t = 504
and $t = 510$. Notwithstanding, at time $t = 511$, the predicted value
exceeds the tolerable threshold and the sample collected at $t = 511
is sent to the sink node. The prediction model is then updated,
considering the new sensed data. In other words, the new sensed
data is inserted into the sample, which defines the linear regression
model. Thus, from time $t = 511$ to $t = 516$ the prediction values
are within the specified acceptable tolerance (threshold), accord-
ing to the new prediction model. It is important to notice that in
our approach the regression model is adjusted dynamically, when-
ever a sensor senses an “outlier”. For $t = 517$, data collected from

---

**Table 2. Threshold Clause added to SNQL.**

<table>
<thead>
<tr>
<th>Clause</th>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>THRESHOLD $&lt; \text{snd\ percent} &gt;$:</td>
<td>$&lt; \text{snd\ percent} &gt;$: error tolerance (e.g., 5%).</td>
</tr>
<tr>
<td>THRESHOLD $&lt; \text{snd\ percent} &gt;$: Specifies prediction</td>
<td></td>
</tr>
</tbody>
</table>

**Table 1. SNQL complex query.**

```
SELECT t.predicted-values
FROM MODEL.temperature $< I_t >$ as t
TIME WINDOW 86400 SENSE INTERVAL 3600
```
should be updated after a sensor node had sensed 3 values which (see Section 4.2). Thus, one may define that the prediction model promptly.

In this sense, \( ADAGA \) implements a conservative approach w.r.t. the moment for recalculating regression equation coefficients. Moreover, three different storage areas are required: (i) reception area used to store received packets (from other sensor nodes); (ii) processing area, which stores data to be processed (for example, by the prediction mechanism); and (iii) forwarding area, which temporally stores data to be transmitted later [17].

4.3 In-Network Data Prediction Mechanism: Conservative Approach

In the aggressive version of \( ADAGA \) ’s in-network data prediction mechanism, as soon as a sensor node identifies a sensor reading \( r \) for which the absolute difference between \( r \) and the prediction value is greater than the value specified in the threshold clause, the sensor node transmits \( r \) to the sink node, which in turn recomputes the regression equation coefficients. That sensed data \( r \) may represent in fact a new tendency for the values which are being collected from the environment. In this case, it is reasonable to update the prediction model promptly.

Nonetheless, it is also possible that the sensed value \( r \) be an outlier yielded by a temporally sensor-failure or a transient non-filtered noise. By sensor-failure problem we mean a defect in the sensing component of a sensor node, which may hinder sensors of collecting data or may induce sensors to yield wrong data. In such a scenario, immediately recalculating the prediction equation is not an efficient decision.

In this sense, \( ADAGA \) implements a conservative approach w.r.t. the moment for recalculating regression equation coefficients by postponing the instant of updating the prediction model of a sensor node. The idea is to wait for a given number of times, called wait window, in which the sensed value is not in the prediction model, that is the following rule is violated: \( P \geq 100(V_c - V_p)/V_c \) (see Section 4.2). Thus, one may define that the prediction model should be updated after a sensor node had sensed 3 values which were not in the prediction model. In this case, the wait window is 3.

This way, \( ADAGA \) is able to identify and differentiate new tendency in sensed-data time series and outliers. As show in Section 5.2, such a feature brings advantages even regarding a small increase in error of rebuilding sensor field.

5. EVALUATION

In this section, we present results obtained from experiments executed on real data. To run the experiments for evaluating \( ADAGA \) and the aggressive approach of its in-network data prediction mechanism, we have exploited data from the Meteorology and Hydric Resources Foundation of the State of Ceará (FUNCEME) in Brazil. For validating \( ADAGA \) and the conservative approach of the data prediction mechanism, simulations with a prototype implemented in Sinalgo [8] have been executed on real temperature, humidity and luminosity data made available by the Intel Berkeley Research Laboratory [12].

5.1 Evaluating \( ADAGA \) - \( P^* \): Aggressive Approach

To carry out the tests for evaluating \( ADAGA \) – \( P^* \) implementing an aggressive approach of the in-network data prediction, we have used real data on temperature (in degree Celsius) collected by a set of 26 sensors geographically distributed in the State of Ceará (in Brazil). Each of the 26 sensors collected data every one hour (sense interval clause - see Table 1) over an interval of 46 consecutive days, which corresponds to 1,104 sensed data per sensor, totaling 28,704 collected data. Figure 4 shows a Ceará map with the location of 26 sensors. Those sensors are located in three different sub-regions (rectangles) with different climatic profile and consequently with expressive variation of temperature values: coastal, mountain and caatinga (forest composed of stunted trees and thorny bushes, found in areas of small rainfall in northeastern Brazil).

Figure 4. Ceará Map.

We have considered that all sensor nodes initially had the same amount of power and available memory. The query used to generate the evaluation results is the one depicted in Table 1.

Figure 5 shows the temperature profile of the 26 sensors in a day chosen at random. Looking more closely to Figure 5, the temperature has varied in the interval \([20^\circ C, 34^\circ C]\), which represents a variation of \(14^\circ C\) in a single day. Therefore, there is a significant difference in the absolute values measured by each sensor, where
they have their maximum values around noon and minimum values
around midnight.

Figure 5. Temperature profile on a typical day.

Finally, Figure 6 shows the daily maximum and minimum values
in the whole data set. Note that the temperature varies in the interval
[15.2°C, 39.3°C], which represents a wide range (a variation of
24.1°C) and makes the prediction a challenging task.

Figure 6. Temperature range for the whole data set.

The mechanism proposed in [9], which is based on kernel dis-
tributed regression, and ADAGA – P∗ have been compared regard-
ing the following metrics: RMS error (RMSE), communication
cost and the number of floating point operations (FPOs) in sensor
nodes. The methodology of this comparative analysis is described
below.

To evaluate the RMS error we have compared the surface gener-
ated by Distributed Regression method described in Section 3 with
the surface yielded by ADAGA – P∗ in the FC. For that, we have
applied the same parametric linear regression method. In this case,
regression performs interpolation on all sensor data, real or pre-
dicted. That is, the base functions extend for the entire data space
without the kernel restrictions. For evaluating the communication
cost, the average number of messages sent or resent by sensor nodes
has been computed. Finally, to compute the number of instructions
executed by the sensors for generating the regression surface, we
have considered the average number of floating point operations
(FPOs).

In both experiments (distributed regression and ADAGA – P∗) the
basis functions set used was the linear-space in each snapshot, as
in [9]: \( \hat{f}(x,y,t) = c_1x + c_2y. \)

Regarding ADAGA – P∗, we have used 3 different thresholds (1%,
2% and 5%) for computing in-network data prediction (see Section
4.2). The sliding window used by ADAGA – P∗ for data prediction
was of size 7. The results are shown in Table 3. To compute com-
munication cost, we have considered the average number of hops
denoted \( \bar{n}_h \) necessary to carry a message from each sensor to the
FC.

The results in Table 3 are for a small WSN where all nodes reach
the FC in one hop (\( \bar{n}_h = 1 \)). However, variations in \( \bar{n}_h \) does not
change the relative position of the total energy consumption among
the evaluated approaches, since communication cost for all approaches
is computed based only on \( \bar{n}_h \). The model used for simulating en-
ergy consumption considers consumption of 75 nJ per bit transmit-
ted, 50 nJ per bit received, and a consumption of 125 nJ/bit for
sensing and processing [3]. The naïve method represents the strategy
of transmitting all sensed data to the FC to build the regression
model.

Table 3 shows that the energy consumption to run the distributed
regression was 15 times higher than the energy consumed to exe-
cute ADAGA – P∗ (with threshold of 1%), for an RMSE difference
of 0.003.

The number of FPOs in Table 3 was calculated as follows. For
ADAGA – P∗, we have considered 6 FPOs to predict and to com-
pare the predicted data with the sensed data in each sensor. On the
other hand, for the distributed regression approach, we have con-
sidered the following computation: (i) for each sensor in a kernel,
the computation of a linear system by Gaussian elimination; (ii)
5 \( k \times k \) matrices multiplication (product), where \( k \) is the number of
sensors in the kernel region.

Network energy consumption has been evaluated w.r.t. the use of
the in-network data prediction strategy implemented by ADAGA –
P∗. Figure 7 shows energy consumption of the WSN previously
described. It is worthwhile to note that we have considered the
both directions of transmission, i.e. from and into sensor nodes,
for computing communication cost. We have computed the en-
ergy consumption of each of the 26 sensor nodes in three different
scenarios: (i) when the in-network prediction is used (i.e., with in-
network aggregation and prediction), (ii) when only ADAGA [4] is
used (i.e., with only in-network data aggregation) and; (iii) when a
naïve strategy is used (i.e., without data aggregation and predic-
tion). To plot ADAGA – P∗’s energy consumption curves (see Fig-
ure 7), we have considered three different threshold values: 1%,
5% and 10%.

Looking more closely to Figure 7 one can observe that increasing
the threshold to 5% and 10%, the reduction of energy consumption
becomes even more significant. The reason for that is the follow-
ing: with higher threshold values, more predicted data are consid-
ered valid. Consequently, less sensed data have to be injected into
the network, which in turn means a reduction in transmission and
reception activities. As mentioned before, transmitting and receiv-
ing data consumes 50% of the energy consumed in a WSN sensor
node.

The effectiveness of simple linear regression used by ADAGA – P∗
for in the in-network data prediction can be substantially improved
by using a more complex prediction model. For example, the cyclic-
nature of the temperature measurements done by sensor 1 (one of
the sensors used in our experiments) can be seen in Figure 8.
For data with such a behavior is well established that a Seasonal ARIMA model [2] generates excellent prediction results. Although the prediction task in this model requires greater computational costs, it would be held at FC and the sensors would require a few more floating point operations (3 FPOs).

It is also important to highlight that even concerning the RMS error, the results for the ADAGA – $P^r$ can also be improved if we consider that the sensor-field regression is performed at the Fusion Center, which does not suffer from any processing restrictions as sensor nodes do. In other words, more powerful sensor-field models can be used without impacting on the energy consumed by sensor nodes.

### 5.2 Evaluating ADAGA – $P^r$: Conservative Approach

For evaluating the conservative approach of the in-network data prediction mechanism, we have utilized the dataset made available by the Intel Berkeley Research Laboratory [12], comprising timestamped topology information along with real values of temperature, humidity, luminosity and voltage. The readings were acquired from 54 sensor nodes physically deployed in the lab and were transmitted to a base station every 31 s between 02/28/2004 and 04/05/2004.

From the aforementioned dataset, we have used data collected by 53 out of the 54 sensor nodes, since the rate of missing data for the fifth sensor was very high. Indeed, the number of data items available in the dataset for that specific sensor node (only 35) corresponds to circa 0.1% of those available for the sixth sensor, for instance. These 35 data items could not be reallocated to other sensor nodes since the data sensed by different nodes display different time series dynamics reflecting their specific spatial locations. Therefore, the simulated network consisted of 53 nodes in total, with at most $10^3$ readings for each sensor node.

We have validated the conservative approach w.r.t. four metrics: RMS error, volume of messages injected into the network, wait window (0, 2, 4 and 6) and threshold (1%, 2%, 3%, 4% and 5%).

<table>
<thead>
<tr>
<th>Method</th>
<th>RMSE (%)</th>
<th>Comm. cost ($X_1$)</th>
<th>FPOs</th>
<th>Sense cost ($X_3$)</th>
<th>Energy consumption (nJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dist. Reg.</td>
<td>0.011</td>
<td>254 + 28$h_{th}$</td>
<td>7050</td>
<td>2600</td>
<td>73100</td>
</tr>
<tr>
<td>$ADAGA - P^r$ with In-net Pred.(1%)</td>
<td>0.014</td>
<td>14$h_{th}$</td>
<td>156</td>
<td>2600</td>
<td>5130</td>
</tr>
<tr>
<td>$ADAGA - P^r$ with In-net Pred.(2%)</td>
<td>0.022</td>
<td>11$h_{th}$</td>
<td>156</td>
<td>2600</td>
<td>4755</td>
</tr>
<tr>
<td>$ADAGA - P^r$ with In-net Pred.(5%)</td>
<td>0.016</td>
<td>7$h_{th}$</td>
<td>156</td>
<td>2600</td>
<td>4255</td>
</tr>
</tbody>
</table>

For data with such a behavior it is well established that a Seasonal ARIMA model [2] generates excellent prediction results. Although the prediction task in this model requires greater computational costs, it would be held at FC and the sensors would require a few more floating point operations (3 FPOs).

It is also important to highlight that even concerning the RMS error, the results for the ADAGA – $P^r$ can also be improved if we consider that the sensor-field regression is performed at the Fusion Center, which does not suffer from any processing restrictions as sensor nodes do. In other words, more powerful sensor-field models can be used without impacting on the energy consumed by sensor nodes.

**Table 3. Evaluation results of the algorithms - total energy cost for a sensor-field snapshot ($h_{th} = 1$).**

Figure 7. Energy consumption - aggressive approach.

Figure 8. Cyclical behavior of the temperature sensor 1.

Figure 9. RMS error for temperature values.
The results presented in Figure 10 proves our claim that increasing the \textit{wait window} and threshold values reduces the data volume flowing along the network. In Figure 10, for $[\text{wait window, threshold}] = [0, 0]$ the RMS error is 0.136, while for $[\text{wait window, threshold}] = [6, 5]$ is 0.653, which is five times greater than the former case. Nevertheless, for $[\text{wait window, threshold}] = [0, 0]$ the number of transmitted messages is 192,276. On the other hand, for $[\text{wait window, threshold}] = [6, 5]$ the number of messages is reduced to 27,830. This represents a reduction of 85.5\% in the amount of messages sent through the network. Such a property represents a strong economy in energy consumption in the network.

Figures 11 and 12 depict the results for humidity. The curve behaviors are similar to those presented for temperature values.

The results for luminosity are presented in Figures 13 and 14. Regarding RMS error for luminosity, it is important to note that sensed values may vary very quickly. In order to illustrate this fact, we present four sequential reading values for sensor #6: (1.38, 1.38, 861.2, 1.38, 1.38). Such values may be capturing a moment in which a window is closed (or a light is turned off) and suddenly someone opens it (or turns on the light). For that reason, the prediction model is not able to react in same frequency in such scenarios.

Nonetheless, the behavior observed for temperature and humidity is the same for luminosity i.e., the greater the values for \textit{wait window} and \textit{threshold}, the less the result accuracy and the number of sent messages are.

To conclude this section, it is important to highlight that the notions of \textit{wait window} and \textit{threshold} make possible to define different levels of query result accuracy and energy consumption. By this means, it would be possible, for example, to specify that a given query should be serviced with less accuracy w.r.t. its result for the sake of energy conservation of the whole WSN. By customizing the query processing through different values of \textit{wait window} and \textit{threshold}, we claim that significant gains in terms of energy consumption may be achieved, mainly in those cases where the application requirements are more flexible.

The results presented in this section indicates significant gains in energy efficiency that vary for different values of \textit{wait window} and \textit{threshold}, evidencing the suitability of the proposed approach for a plethora of WSN applications.

6. DISCUSSION

The potential packets loss in WSNs has a quite limited impact in the results produced by $ADAGA-P^*$. This happens because $ADAGA-P^*$ builds and instantiates the regression models using predicted values. Such values may be outside the threshold error (used by
$ADAGA - P^*$’s in-network prediction). However, the effect will be diluted by surface regression unless values sent by multiple sensors are lost simultaneously.

By sensor-failure problem we mean a defect in the sensing component of a sensor node, which may hinder sensors of collecting data or may induce sensors to yield wrong data. In this sense, the in-network prediction strategy implemented by $ADAGA - P^*$ may be negatively impacted whenever sensor failures occur. This is because a sensor may stay a long time without transmitting any data even when a long sequence of correct predictions happens. To solve that problem we propose approaches. One is to establish a validity period for the prediction model. Thus, a sensor always transmits data after this interval. Nevertheless, in this case, long intervals might delay failure detection and on the other hand shorter intervals may impact on communication cost. The second approach is to run a process in FC, which uses the spatial correlation of sensor data to detect whenever significant changes in sensor data occur.

In [9], in order to analyze scalability in the kernel distributed regression, the authors use as metric the number of messages sent to the FC. Thus, one can argue that in distributed regression [9, 10, 11], the number of bytes sent to the FC grows linearly with the number of nodes in the network. However, it is important to observe that the number of bytes sent to the FC grows linearly with the number of WSN nodes in the kernel distributed regression if and only if the number of nodes and the number of basis functions in each kernel region is kept constant. In $ADAGA - P^*$, the number of bytes sent to the FC always grows linearly with the number of nodes in the network. This is because $ADAGA - P^*$ requires that each sensor node just sends a subset of sensed data. Furthermore, in the kernel distributed regression approach the order of the linear system to be solved at each sensor node and the number of intra-kernel communications grows with the square number of nodes and basis functions in the kernel. This is not the case of $ADAGA - P^*$.

Sensor node time synchronization may play an important role for energy conservation in MAC layer protocols. Furthermore, it may be a critical feature for several WSN applications, for which sensed values and timestamps (indicating the time at which a sensor reading has been sensed) should be sent along the network [19]. Such a feature requires a common notion of time among the sensors.

$ADAGA - P^*$ is able to work in WSNs which implement a sensor node time synchronization or not. For a WSN without an in-built synchronization mechanism, $ADAGA - P^*$ can infer the timestamp of a predicted value by computing the average of timestamps within a prediction window.

The experiments presented in Section 5.1 have considered a WSN with a synchronization mechanism, while experiments in Section 5.2 have been carried out in a WSN without any synchronization mechanism. For an in-depth read on time synchronization problem the reader is referred to [1, 20].

7. CONCLUSION

In this work, we described a framework, denoted $ADAGA - P^*$, for managing sensor-field models. The idea is to provide to query engines for WSNs (running upon $ADAGA - P^*$) the ability of processing complex queries. The proposed framework can be applied in scenarios where: (i) applications need data which will be sensed in the future (e.g., weather forecast applications), or (ii) top-K queries are often submitted to access data in WSNs. In order to achieve its goals, the proposed framework is able to construct and instantiate sensor-field models based on any regression models (e.g., polynomial regression or spline). Since sensor-field models are built to match application specific requirements, there may be different models for a single physical phenomenon (e.g., temperature or pressure). Thereafter, $ADAGA - P^*$ can instantiate the models. By instantiating a model we mean that $ADAGA - P^*$ is able to provide to the query engine a regression model of the sensor field from the samples sensed by the WSN.

A key feature of $ADAGA - P^*$ is to build and instantiate sensor-field models in the fusion center in order to save power in sensor nodes, since they do not have to consume energy for computing the sensor-field model. For example, the distributed regression approach proposed in [9, 11, 10] builds the models in the sensor nodes. However, $ADAGA - P^*$ does not require every sensed data to be transmitted to FC to instantiate a model. This is because it implements a mechanism for predicting data which are being sensed by sensor nodes in a WSN. Furthermore, the prediction mechanism implemented by $ADAGA - P^*$ makes viable to define different levels of result accuracy and energy consumption.

By means of experiments on real data, we have validated $ADAGA - P^*$’s efficiency. The results have shown that significant gains in terms of energy savings are achieved.

Finally, we emphasize the applicability and the effectiveness of the proposed framework for a wide variety of applications which consume data produced by WSNs.

8. ACKNOWLEDGEMENT

This work has been partially supported by FUNCAP (Ceara, Brazil) grant PRN-0040-00037.01.00/10 and CAPES (Brazil) grant 18319-12-2.

9. REFERENCES


ABOUT THE AUTHORS:

José Everardo Bessa Maia received the MSc degree in Electrical Engineering from the State University of Campinas (UNICAMP), Brazil, in 1989. In 2011 he received the Doctor degree in Electrical Engineering from the Federal University of Ceará, Brazil, working in Computational Intelligence. He has been with the State University of Ceará, Brazil, since 1992.

Angelo Brayner received the MSc degree in Computer Science from the State University of Campinas (UNICAMP), Brazil, in 1994. In 1999 he received the PhD degree from the University of Kaiserslautern, Germany, working in the field of Transaction Management in Multidatabase systems. He has been with the University of Fortaleza, Brazil, since 2001 as full professor and leader of the CEARA (advanCEd dAtabase Research) research group. His current research interests include high performance transaction systems, query processing in wireless sensor networks and mobile databases.

Fernando Rodrigues received the MSc degree in Computer Science from the Federal University of Ceara, Brazil, in 2001. In 2011, he joined Prof. Angelo Brayner’s research group in order to start his PhD work. His current research interest is query processing in wireless sensor networks.
An Automatic Blocking Strategy for XML Duplicate Detection

Luís Leitão
IST/INESC-ID
Av. Professor Cavaco Silva
2744-016 Porto Salvo, Portugal
luis.leitao@ist.utl.pt

Pável Calado
IST/INESC-ID
Av. Professor Cavaco Silva
2744-016 Porto Salvo, Portugal
pavel.calado@ist.utl.pt

ABSTRACT

Duplicate detection consists in finding objects that, although having different representations in a database, correspond to the same real world entity. This is typically achieved by comparing all objects to each other, which can be unfeasible for large datasets. Blocking strategies have been devised to reduce the number of objects to compare, at the cost of losing some duplicates. However, these strategies typically rely on user knowledge to discover a set of parameters that optimize the comparisons, while minimizing the loss. Also, they do not usually optimize the comparison between each pair of objects. In this paper, we propose a blocking method of combining two optimization strategies: one to select which objects to compare and another to optimize pair-wise object comparisons. In addition, we propose a machine learning approach to determine the required parameters, without the need of user intervention. Experiments performed on several datasets show that not only we are able to effectively determine the optimization parameters, but also to significantly improve efficiency, while maintaining an acceptable loss of recall.\footnote{Copyright is held by the authors. This work is based on an earlier work: SAC'13 Proceedings of the 2013 ACM Symposium on Applied Computing, Copyright 2013 ACM 978-1-4503-1656-9/13/03. http://doi.acm.org/10.1145/2480486.2480523.}

Categories and Subject Descriptors

H.2.5 [Database Management]: Heterogeneous Databases

General Terms

Algorithms, Performance, Experimentation

Keywords

Duplicate Detection, Data Quality, XML, Blocking, Optimization

1. INTRODUCTION

Duplicate detection techniques focus on linking object records that represent the same real world entity, although they may contain different values. This is critical task in several research and business areas, where preserving data quality is essential to the success of organizations [7].

What makes duplicate detection a non-trivial task is the fact that duplicates are not exactly equal, often due to errors in the data. Consequently, we cannot use common comparison algorithms that detect exact duplicates. Instead, we have to compare all objects, using a possibly complex matching strategy, to decide if they refer to the same real-world object or not.

Comparing every pair of objects represents, however, a bottleneck on duplicate detection efficiency. A naive comparison strategy for a set of \(n\) objects would yield a complexity of \(O(n^2)\), making the task hard to scale for large datasets. In addition, even comparing a single pair of objects can be an expensive operation, since it usually involves comparing each pair of object attributes, using a string similarity measure that is itself quadratic on string length.

With the growing availability of digital data, many solutions have been proposed for these problems. One particular type of solution, perhaps the most common, consists on what are usually called blocking strategies. Blocking strategies try to avoid the comparison of all objects, by organizing them into groups, or blocks, of likely duplicate candidates [8, 16]. This is done according to a blocking key (BK) formed by combining attributes, or attribute segments. Only objects in the same group are fully compared to each other, which drastically reduces the total computational complexity. Duplicate candidates are usually determined by comparing all objects, but with a much cheaper comparison measure, using only the BK.

Of course, although this approach significantly improves performance, it also has two major limitations. First, since not all objects are compared, some duplicates may be missed. For this reason, we say blocking techniques are lossy. Second, blocking relies on a set of parameters that need to be defined by a user. Knowing which parameter values to choose, however, is not trivial and may require a deep knowledge of the data.

Furthermore, blocking strategies frequently neglect any possible optimization of the comparisons performed within each block of duplicate candidates. In fact, comparing attributes values using a string similarity measure is the most expensive operation in the duplicate detection process. Thus, by optimizing this step, not only can we improve performance by reducing the number of string comparisons, but also we can have larger candidate blocks, thus reducing the potential loss of duplicate objects, without increasing the overall processing time.
In this work, we are concerned with the problem of finding duplicates in XML databases. XML is an increasingly popular data model, which requires techniques different from those generally used in relational data [18]. Yet, few solutions have been developed specifically for it. Our contribution is, therefore, twofold. First, we propose a method for the optimization of XML duplicate detection that combines a blocking strategy with an object comparison optimization. Second, our proposed method is able to automatically select the required optimization parameters, without the need of user intervention and without any knowledge of the database being processed. Along with the parameter selection we further provide a method to both automatically select the most suited BK and the more relevant attributes to compare.

This procedure uses a distantly supervised machine learning approach, which is capable of discovering the parameters that minimize both the number of object comparisons and the total number of string comparisons, while maintaining the loss of duplicate objects close to a user-defined threshold.

As a choice of blocking strategy we use the Canopy Clustering algorithm [16]. To optimize XML object comparisons, we use the XMLDup algorithm [14]. Experiments performed on a set of real world and artificial XML databases show that our solution is able to effectively discover suitable sets of parameters, in some cases reducing comparisons down to 0.025% of the original value, while loosing less than 8% duplicate objects.

The remaining of this document is organized as follows. In Sec. 2 we put the subject of this work into context by addressing related work. Next, in Sec. 3, we start by introducing the duplicate detection system that serves as basis for our blocking approach. We then explain how the comparison optimization strategy works and how it can be combined with the blocking strategy. Sec. 4 shows how to automatically discover the parameters used in our solution. Using a variant of this method, Sec. 5 describes how it can be generalized in order to automatically select the best attributes, later used for BK selection. In Sec. 6 we validate our proposal through a set of experiments. Finally, in Sec. 7, we finish with some concluding remarks.

2. RELATED WORK

Many solutions have been developed to optimize the duplicate detection task. However, few solutions tackle some of the problems addressed in this work. Namely, the problem of (i) automatically determining the best parameters, including the BK, (ii) optimizing the comparison between pairs of records, and (iii) optimizing duplicate detection in XML data. There have been, however, some attempts at solving these issues, of which we now describe some examples.

Regarding problem (i), some works have been proposed to automatically determine the best BK. An example is that of [2]. In their work, the authors propose to use an optimization algorithm that, using training data, i.e. data where some duplicate objects are known, finds a (sub)optimal indexing function, which is composed by attribute predicates.

Regarding problem (ii), in the work proposed by [20], the authors try to reduce the number of comparisons performed during the record matching phase. For this, they use feature selection techniques to discover the most significant attributes, and only those are compared. With a similar goal, in [5], the authors recur to a machine learning approach to build an object matching rule, composed by parameters such as edit distance restrictions. This solution, however, has the limitation of needing a significant amount of user intervention to label the training data.

Finally, regarding problem (iii), a method called SXNM was proposed in [18]. This approach is an adaptation of the Sorted Neighborhood method [8]. Like in the original solution, the idea is to avoid performing useless comparisons between objects by grouping together those that are more likely to be similar. The difference is that, instead of using one single relation, the sliding window is applied to nested XML elements.

For a more detailed comparison and evaluation of optimization methods, we refer the reader to [3].

3. EFFICIENT DUPLICATE DETECTION

The process of detecting if two objects are duplicates relies on comparing the information they both contain. This information is typically found as values of the objects attributes, i.e., any object field that contains information, such as name or pob in Fig. 1. In this section we describe how two optimization strategies can be combined in order to provide more efficient duplicate detection, by reducing both the number of objects and the number of attributes compared.

From this point on, we will refer to the reduction of the number of objects to compare as list-wise optimization and to the reduction of the number of attributes to compare as pair-wise optimization.

3.1. List-wise Optimization

Our solution to reduce the number of objects that will be subject to a more detailed comparison is to use the well known Canopy Clustering algorithm [16]. Canopy Clustering is a light clustering approach, based on the idea of building overlapping clusters, called canopies. It works by extracting q-grams from the objects BKs and comparing them using a cheap similarity measure, such as the Jaccard [9] coefficient or the cosine similarity [1]. The process is illustrated in Algorithm 1. Strategies to obtain the BKs used in the object clustering can go from selecting a single attribute to the combination of multiple attributes or even parts of them. For instance, one can define a BK to be the concatenation of the first two characters of a given attribute with the last four of another. Despite the many approaches that can be adopted, these are usually performed relying on human effort. In Sec. 5 we propose a method to automatically select the attributes that best compose an optimized BK.

Algorithm 1 starts by randomly selecting an object, $O_1$, cluster $C_1$ (line 3). Each of the remaining objects $O_2$ is then compared to $O_1$, using the BK, (line 5). If $O_2$ falls within a similarity threshold $t_{tight}$, it becomes part of cluster $C_1$ and no other cluster (lines 6–7). If $O_2$ falls within a similarity threshold $t_{loose}$, it becomes part of cluster $C_1$, but can still belong to a different cluster, later on. This process continues.
Algorithm 1: CanopyClustering(O, T\textsubscript{tight}, T\textsubscript{loose})

Require: A set of objects O; thresholds $T_{\text{tight}} > T_{\text{loose}}$

Ensure: A set of clusters $C$

1: $C ← \emptyset$
2: while $O$ is not empty do
3:    $C_i ← O_i$, where $O_i$ is a randomly selected object from $O$
4:    $O ← O - \{O_i\}$
5:    for each object $O_j$ in $O$ do
6:        if $\text{sim}(O_i, O_j) \geq T_{\text{tight}}$ then
7:            $O ← O - \{O_j\}$
8:        end if
9:        if $\text{sim}(O_i, O_j) \geq T_{\text{loose}}$ then
10:           $C_i ← C_i \cup \{O_j\}$
11:       end if
12:    end for
13:    $C ← C \cup \{C_i\}$
14: end while
15: return $C$

until all objects belong to, at least, one cluster (line 2).

Once the clustering has been performed, we still need to
calculate every object within each cluster. In the following,
we present a strategy to optimize this comparison.

3.2. Pair-wise Optimization

The pair-wise optimization approach is based on the XML
duplicate detection system XMLDup [14]. We now explain
how XMLDup works and its strategy for optimizing attribute
comparisons.

3.2.1. The XMLDup System

XMLDup is a duplicate detection system, originally pro-
posed for XML data. Its approach for duplicate detection
centers around one basic assumption: The fact that two
XML nodes are duplicates depends only on the fact that their
values are duplicates and that their children nodes are dupli-
cates. Thus, two XML trees are duplicates if their root nodes
are duplicates. In XMLDup, this process is represented by a
Bayesian network. To illustrate, Fig. 2 shows the Bayesian
network used by XMLDup to determine if the objects in
Fig. 1 are similar.

![Bayesian network](image)

Figure 2. Bayesian network to compute the similarity of the trees in Figure 1.

![XML trees](image)

Figure 1. Two XML objects that represent the same
person. Each person (prs) is composed by the fol-
lowing nodes: name, date of birth (dob), contact
(cnt), place of birth (pob), email (eml) and address
(add).

All nodes in the network of Fig. 2 are assigned binary random
variables, taking the value 1 to represent the fact that the
responding data in trees $U$ and $U'$ are duplicates, and
taking the value 0 to represent the opposite. Thus, to decide
if two objects are duplicates the algorithm has to compute
the probability of the root nodes being duplicates, $P(prs_1 = 1)$,
which can be interpreted as a similarity value between the
two XML elements.

Due to space constraints, details on how the network is built
are not described here, but can be found in [13].

Detailed information on Bayesian Networks and their appli-
cations can be found in [17].

3.2.2. Computing the Similarity

To obtain the probability $P(prs_1 = 1)$, network evalua-
tion is performed by doing a propagation of the prior prob-
abilities, from the leaf nodes, in a bottom up fashion, un-
til reaching the topmost node. The prior probabilities are
obtained by applying a similarity measure to the pair of at-
tribute values represented by the leaf nodes. Computing
such similarities is the most expensive operation in the net-
work evaluation, and in the duplicate detection process in
general. Therefore, the idea behind XMLDup optimization
lies in avoiding the calculation of prior probabilities, unless
they are strictly necessary. The process is presented in detail
in Algorithm 2.

Algorithm 2 takes as input a node $N$ from the BN and a
user defined threshold $T$. It starts by gathering a list of all
the parent nodes of $N$ and assuming that their duplicate
probability score is 1 (lines 1 and 2). It then proceeds to
compute the actual probability value of each of the parents
of $N$ (lines 4–16).

If a given parent node $n$ is a value node (line 6), its proba-
bility score is simply the similarity of the values it represents.
If, on the other hand, $n$ also has parent nodes, its proba-
bility score depends on the scope of its own parents, which
we compute recursively (line 9). However, the algorithm
should now be called with a different threshold value, that
depends on the equation used to combine the probabilities
for node $N$ (line 8). Once the score for node $n$ is computed,
the algorithm checks if the total score for $N$ is still above $T,$
and decides whether to continue computing or to stop the network evaluation (lines 13–15).

In line 2, we state that every node is assumed to have a duplicate probability of 1. We call this assumed probability the **pruning factor**. Having a pruning factor equal to 1 guarantees that the duplicate probability estimated for a given node is always above the true node probability. Therefore, no duplicate pair of objects is ever lost. By lowering the pruning factor, we lose this guarantee. Thus, a pair of objects may be prematurely discarded, even if they are true duplicates. However, with a lower pruning factor, we also know that all probability estimates will be lower. This will cause the defined duplicate threshold to be reached sooner and, consequently, the network evaluation to stop sooner. Thus, fewer similarity calculations will be performed.

Based on this knowledge, the optimization strategy lies on the assumption that, by slightly lowering the pruning factor, we can achieve high gains in performance, while losing only a very small amount of true duplicates. More details on this procedure can be found in [14].

### 3.3. Combining the Optimization Strategies

Combining the list-wise and pair-wise optimization strategies defined above is straightforward. We start by using the blocking strategy of Sec. 3.1 to create the blocks of potential duplicate objects and then apply the optimization procedure of Sec. 3.2 to the pairs of objects within each block. However, assuring that this simple combination is effective is not a trivial task. This is mainly due to the free parameters in each approach, and how they interact when both approaches are combined.

The Canopy Clustering algorithm, for example, raises two important issues. The first is how to define the thresholds $t_{\text{tight}}$ and $t_{\text{loose}}$. If $t_{\text{loose}}$ is too high, we may discard many true duplicate objects. Also, if we allow for a large overlap between clusters (i.e., if $t_{\text{tight}}$ is much higher than $t_{\text{loose}}$), we can unnecessarily deteriorate the performance.

The second issue is that, since the similarity measures used are based on $q$-grams, the size of $q$ directly affects the outcome. For instance, for two BK with values "Madona" and "Madonna", respectively, the score obtained using the Jaccard coefficient will be 0.7 for $q = 3$ and 0.58 for $q = 4$. Given that the similarity score is a conditioner for the thresholds, it is hard to define the $q$-gram size that provides the most efficient separation between clusters.

In the case of XMLDup, the major parameter to account for is the pruning factor. A high pruning factor ensures that few duplicates are lost, while a low pruning factor greatly increases efficiency.

When both strategies are used together, parameter selection becomes even more challenging. For example, we should be able to increase the size of the clusters, thus avoiding losses, and compensate the loss in efficiency by using a lower pruning factor. In fact, the problem we need to solve is how to obtain the best trade-off between all parameter values. In addition, a good choice of parameters depends on the BK used, i.e., different BKs may required a different parameterization.

Since taking all these factors into consideration can be a daunting task, and require a deep knowledge of the data, in the following section we propose a method to automatically determine the optimal parameter values, given a chosen BK.

### 4. LEARNING THE PARAMETERS

As we have seen in Sec. 3, both optimization strategies depend on a set of parameters. A good choice of values for these parameters should maximize efficiency, while minimizing the loss of duplicate objects. These optimal values depend on the chosen BK. Thus, the problem to solve is how to select the best possible parameters according to a selected BK. In other words, we would like to determine a function that, given a BK, yields the appropriate parameter values. We now describe how to build such function.

#### 4.1. Automatic Parameter Selection

To automatically determine which parameters produce the lowest number of comparisons, whilst losing the least number of duplicates, we adopt a solution similar to the one proposed in [14].

Assume that we have an instance of a database, $D$, where we wish to find duplicates. Let $k$ be a BK selected for $D$. This BK can be formed by one or more attributes from the schema of $D$, or from parts of the attributes in $D$. In our solution, we will represent $k$ as a vector of statistical features $\mathbf{k}$, extracted from the data corresponding to its composing attributes (or parts of attributes). Our goal is to find a mapping function $M : k \rightarrow \theta$, which takes the representation of the BK, $\mathbf{k}$ and returns the corresponding parameters $\theta$.

Finding function $M$ can be seen as a regression problem. Our regression model should take the vector of features from one BK and yield the corresponding optimal parameter values. We note that, since we need to estimate more than one parameter, we use a different regression model for each single parameter. We thus assume that the parameters are independent, given the training data. We believe this simplification, as opposed to learning all parameters from a sin-
gle model, can produce effective results while requiring less training data.

Thus, let \( D_1, D_2, \ldots, D_n \) be a set of databases (different from \( D \)), for which we know some of the duplicate objects they contain, let \( B = k_1, k_2, \ldots, k_n \) be a set of BK features, each one extracted from database \( D_i \), and let \( R \) be the minimum recall we allow during the duplicate detection procedure, where recall is defined the percentage of duplicates found, over the total number of duplicates.

Assume, for now, that we know the parameters \( \theta_i \) that produce the ideal results when combined with each \( k_i \), for its respective database. Recall that the ideal parameters are those that minimize the number of comparisons performed, while finding at least \( R \)-percent duplicates. Finding function \( M \) is now simply a matter of using the set of pairs \((k_i, \theta_i)\) as training data for the regression model. In this work, we chose the method of Support Vector Regression [6]. Once the model is trained, we can use \( M \) to map any given BK to the corresponding optimal parameters.

Note that the pruning factor strategy deeply relies on the most significant attributes to either discard or continue with the similarity evaluation. Moreover, we expect the BK to be representative of the object record while composed by distinctive information. Thus, we believe the mapping between the blocking key and the pruning factor should be accurate enough, which is confirmed by experimental evaluation. However, we still have a question to answer. How do we obtain the ideal parameters \( \theta_i \) for training? In the following section, we address this issue.

### 4.2. Discovering the Best Parameters

To find the ideal parameters for our optimization algorithm, a naive solution would be to test every combination for the best possible outcome. This is, of course, unfeasible. Instead, we use a process of Simulated Annealing (SA) [11] to search for the ideal parameter values.

SA is an algorithm to determine the maximum (or minimum) value of a function with several independent variables, under a fraction of the time needed to find an absolute maximum (or minimum) in a large search space. This function is called the *objective function* and its results depend on the possible variable configurations, or *states*. SA consists in shifting among candidate states in order to approximate the objective function to a global optimum, using an acceptance function to decide if the new state should be considered.

In our approach, the objective function \( p \) to minimize corresponds to the number of string comparisons performed during the duplicate detection process and the states \( s \) correspond to the possible parameter set values. Thus, a state will be represented by the threshold values, the q-gram size, and the pruning factor, i.e. \( s = (t_{\text{tight}}, t_{\text{loose}}, q, p_f) \). To determine if the algorithm should change state, we use the following variation of the commonly used acceptance function:

\[
P(p_s, p_n, t) = \begin{cases} 
0 & \text{if recall } \leq R \\
1 & \text{if } p_n > p_s \\
\frac{p_n - p_s}{e^{(p_n - p_s)}} & \text{otherwise}
\end{cases}
\]

where \( p_s \) is the objective function value for the current state, \( p_n \) is the objective function value for the new state, \( t \) is the fraction of iterations the algorithm still needs to execute, and \( R \) is a minimum threshold for the recall achieved. Equation (1) assures that a new set of parameters is accepted only if the loss in recall is not below a given value. In our experiments, we defined \( R \) as 90% of the recall achieved by the duplicate detection without employing any optimization strategy.

Using SA on databases \( D_1, D_2, \ldots, D_n \), with the chosen BKS \( k_1, k_2, \ldots, k_n \) allows to find the optimal, or close to optimal, sets of parameters \( \theta_1, \theta_2, \ldots, \theta_k \) that can be used to train our regression model.

Finally, we only need to determine what features can be used to compose each BK representation \( k_i \). Naturally, to perform SA, we need to be able to measure both recall and to count the number of comparisons. For that we use the XMLODup algorithm described in Sec. 3. To count the number of comparisons we tracked the number of times, through the entire process, that two attribute values are compared against each other using a string similarity measure. We believe this is the most rigorous way of measuring how efficient our strategy performs since almost all of the computational weight of a duplicate detection algorithm relies in this computation. To represent the BKS we use 7 different features that try to capture its unique statistical characteristics. These were selected from the set of features suggested in [14].

### 5. Attribute Selection

The strategy we have proposed so far almost relieves the user from providing any type of input to the duplicate detection algorithm in order to improve its efficiency. For that goal to be completely achieved, one remaining component needs to be automatically discovered. That component is the BK. In this section we present a strategy that has the potential of automatically performing two tasks at once: (i) selecting the best BK and (ii) further reducing the number of attribute comparisons at the pairwise-level.

#### 5.1. Selecting the More Useful Attributes

Using only a subset of attributes that we consider to be the most important has two main advantages. First, it can improve the efficiency of the duplicate detection process by reducing the number of attributes to compare. Second, it can improve the effectiveness of duplicate detection by ignoring attributes that can potentially deteriorate the process. In this work we are mainly interested in the first advantage. Moreover, we will show how this strategy can also be used for inferring the more appropriate BKS to use in our optimization method.

During the process of duplicate detection, it is common for a user to start by defining which attributes should be used for comparison. This task is usually referred to as object descriptor definition [21]. Naturally, this task requires the user to be aware not only of which type of information each attribute contains, but also which attributes hold erroneous content. However, besides the fact that this knowledge is not usually available, it is not clear how to determine how many
of the most useful attributes will produce the best results. To cope with these difficulties we defined a strategy that, given the complete set of attributes in a dataset, returns those that are the most important for the duplicate detection process.

The strategy employed to select the more useful attributes closely follows the idea described in Section 4. The main difference lies in the fact that, instead of discovering a mapping function that, given a BK, returns the value for the corresponding optimization parameter, we will now have a mapping function $M : A \rightarrow S$, where $A$ is the set of attributes and $S = \{-1, +1\}$, which returns whether an attribute should be used for comparison (+1) or not (-1). To discover such function we use a Support Vector Machine Classifier (SVM) [19]. In Fig. 3 we illustrate the complete procedure.

In our method, each attribute is represented as a vector of statistical features. To train the classifier we provide as input a set of examples containing a group of such feature vectors and their corresponding judgments, i.e. whether the corresponding attribute is useful or not. To determine if a given attribute is useful, we try out several different attribute combinations and examine which ones yield the best results. As before, this is done using the SA algorithm on a set of databases for which we know some of the duplicate objects they contain.

This time, however, each state in SA is represented by a binary vector, where each entry indicates if the corresponding attribute will be used or not. To swap between states, we use a single operation—inverting the bit in a random position of the vector. To illustrate, in Fig. 4 we present the several states that can be obtained for an object with three attributes.

Once the classifier is trained, using the examples found recurring to SA, it can be used to select the more relevant attributes for object comparison. In the next section we explain how we use these selected attributes to extract a BK.

### 5.2. Selecting the BK

Finding the most relevant attributes should also satisfy the goal of selecting the best possible BK. Clearly, using such attributes should yield a BK that is highly effective at distinguishing between objects and, thus, result in less comparisons. Nevertheless, we cannot simply include the complete set of the selected attributes in the BK. We need to further filter this set, to obtain a BK that can be efficiently compared. This can be done in two steps.

First, considering that we want to have a BK composed by the best $n$ attributes, we select those $n$ that are closer to the object root, according to the method defined in [12].

Second, given that we can have more than $n$ attributes at the same distance from the root we need to further filter this group. Thus, we apply the distinctiveness measure $D_A = \log\left(\frac{t}{|d_A|}\right)$, where $t$ is total number of objects in the database and $d_A$ is the set of distinct values occurring for attribute $A$. This allows us to select attributes whose content is more rare and that, therefore, should minimize the number of comparisons.

Note that that these heuristics are only applied to attributes that do not present multiple occurrences of the same element per object. In the next section we present the experiments performed to validate our approach.

### 6. EXPERIMENTS

In this section, we present an evaluation of our combined optimization strategy, using real world and artificial datasets. The goal is to assess the efficiency improvements obtained and their impact on the percentage of duplicates retrieved. Moreover, we test our method for automatic BK selection and attribute reduction in order to verify how this optimization can further improve our initial strategies without loss in accuracy.

#### 6.1. Experimental setup

Our tests were performed using eight different datasets, representing six different data domains. The first four datasets, Country, CD 1, IMDB and Person, consist of a set of XML
objects taken from real databases and artificially polluted by inserting duplicate data and errors such as typographical errors, missing data, and duplicate erroneous data. The remaining four datasets, Cora, CD 2, IMDB+FilmDienst, and Restaurant are composed exclusively of real world data, containing naturally occurring duplicates.

The datasets vary in size from 864 objects (Restaurant) to 12000 objects (Persons). The object structure of each dataset was changed, according to the strategy defined in [12], in order to improve the effectiveness of the results. This strategy follows the idea that attributes are placed in the object hierarchy according to their relative relevance, whereas more relevant attributes are placed in upper structure levels. In all experiments, we used a duplicate threshold of 0.5 for all datasets.

The Cora, Country, IMDB, IMDB+FilmDienst and both CD datasets are available at the Hasso Plattner Institute website\(^2\). The Restaurant and Person datasets were obtained from the University of Texas Machine Learning Research Group website\(^3\) and from the FEBRL record linkage tool package [4], respectively.

To assess effectiveness, we applied the commonly used precision, recall and r-precision measures [1]. To assess efficiency, we measured the number of attribute comparisons performed during the duplicate detection process.

Regression, as explained in Sec. 4.1, was performed through Support Vector Regression (SVR) [6], using the SVMLight implementation with a linear function kernel [10]. The same implementation was used for the SVM binary classification. Both models were trained with all BKs except those belonging to the dataset we were testing. We used at most 43 key/parameter pairs for training.

To compare the objects data, we considered all the attribute values as textual strings. To that end we used the formula 
\[
sim(V_1, V_2) = 1 - e^{\text{ed}(V_1, V_2)},
\]
where \(\text{ed}(V_1, V_2)\) is the string edit distance [15] between values \(V_1\) and \(V_2\), and \(|V_i|\) is the length (number of characters) of string \(V_i\).

For the list-wise strategy, described in Sec. 3.1, we used the Jaccard similarity measure to compare BKs.

The experimental evaluation was performed on an Intel 2-core CPU at 2.53GHz, with 4GB of RAM, on a 64-bit Ubuntu 11 operating system. Both tested algorithms were fully implemented in Java, using the DOM API to process the XML objects.

6.2. Results

In this section we show the results obtained by our parameter selection algorithm using our combined pair-wise and list-wise blocking approach. Tests were performed by executing duplicate detection with a set of BKs, once for each BK, and averaging the results. BKs were built by one or two attributes concatenated. We mixed the most distinctive attributes with less distinctive ones in order to have a wide representativeness of possible BK characteristics. Note that the model was always trained without using the BKs from the dataset we were trying to classify.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Comparisons (%)</th>
<th>R-Precision (%)</th>
<th>Time (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LW</td>
<td>LW+PW</td>
<td>No Opt</td>
</tr>
<tr>
<td>Country</td>
<td>40.76 ± 22.10</td>
<td>23.02 ± 14.24</td>
<td>78</td>
</tr>
<tr>
<td>CD 1</td>
<td>50.36 ± 23.21</td>
<td>29.27 ± 15.37</td>
<td>91</td>
</tr>
<tr>
<td>IMDB</td>
<td>19.31 ± 4.46</td>
<td>3.28 ± 1.18</td>
<td>83</td>
</tr>
<tr>
<td>Persons</td>
<td>25.75 ± 6.92</td>
<td>6.87 ± 1.1</td>
<td>96</td>
</tr>
<tr>
<td>Cora</td>
<td>15.55 ± 17.27</td>
<td>10.68 ± 9.20</td>
<td>84</td>
</tr>
<tr>
<td>CD 2</td>
<td>0.51 ± 0.36</td>
<td>0.06 ± 0.06</td>
<td>84</td>
</tr>
<tr>
<td>Restaurants</td>
<td>0.08 ± 0.1</td>
<td>0.04 ± 0.03</td>
<td>84</td>
</tr>
<tr>
<td>IMDB+FD</td>
<td>1.86 ± 1.68</td>
<td>0.98 ± 0.09</td>
<td>98</td>
</tr>
</tbody>
</table>

Table 1. Comparison between list-wise (LW) and the complete optimization (LW+PW) strategy.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>BK</th>
<th>LW+PW</th>
<th>LW</th>
<th>Recall Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMDB+FD</td>
<td>title</td>
<td>0.279</td>
<td>0.668</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>title,year</td>
<td>1.679</td>
<td>3.049</td>
<td>-9</td>
</tr>
<tr>
<td>CD 1</td>
<td>artist</td>
<td>0.032</td>
<td>0.924</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>dtitle</td>
<td>0.137</td>
<td>0.248</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>artist,dtitle</td>
<td>0.025</td>
<td>0.370</td>
<td>2</td>
</tr>
<tr>
<td>Restaurants</td>
<td>addr</td>
<td>0.098</td>
<td>0.286</td>
<td>-12</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td>0.029</td>
<td>0.034</td>
<td>-6</td>
</tr>
<tr>
<td></td>
<td>phone</td>
<td>0.034</td>
<td>0.053</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>addr,name</td>
<td>0.026</td>
<td>0.026</td>
<td>-11</td>
</tr>
<tr>
<td></td>
<td>name,phone</td>
<td>0.029</td>
<td>0.029</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>addr,phone</td>
<td>0.031</td>
<td>0.040</td>
<td>-6</td>
</tr>
<tr>
<td>Cora</td>
<td>title</td>
<td>8.158</td>
<td>9.069</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>author</td>
<td>3.966</td>
<td>4.042</td>
<td>-47</td>
</tr>
<tr>
<td></td>
<td>title,author</td>
<td>6.033</td>
<td>6.104</td>
<td>-9</td>
</tr>
<tr>
<td></td>
<td>title, volume</td>
<td>21.048</td>
<td>34.276</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>title,journal</td>
<td>26.514</td>
<td>46.145</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>author, volume</td>
<td>3.947</td>
<td>4.018</td>
<td>-47</td>
</tr>
<tr>
<td></td>
<td>author,journal</td>
<td>5.096</td>
<td>5.211</td>
<td>-20</td>
</tr>
</tbody>
</table>

Table 2. Efficiency results for real world datasets. Column LW+PW presents the ratio between the number of attribute comparisons using the combined approach, in relation to a non-optimized procedure; column LW shows the same ratio, using only the list-wise optimization; column Recall Dev shows the deviation from the chosen recall level.
6.2.1. Efficiency Evaluation

We now present the efficiency results obtained by our automatic parameter learning strategy. We start by noting that 60% of the classified BKs optimized the procedure while staying above the predefined recall threshold of 90%. Of the remaining 40% BKs, the majority was very close to the defined threshold, although being a few points below. In fact, 89% of the BKs deviated from the predefined recall by less than 10%. This reveals how effective our strategy is in learning the parameters, even when a reduced amount of training data is used. In addition, we highlight that, in every database, there was always at least one BK that managed to achieve the defined recall with high efficiency scores.

Table 3. Efficiency results for the artificial datasets. Columns are the same as in Table 2.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>BK</th>
<th>LW+PW</th>
<th>LW</th>
<th>Recall</th>
<th>Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CD 2</td>
<td>artist</td>
<td>11.429</td>
<td>24.316</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dtitle</td>
<td>11.308</td>
<td>24.106</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>artist,dtitle</td>
<td>5.933</td>
<td>14.070</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dtitle,cdextra</td>
<td>43.230</td>
<td>72.623</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dtitle,genre</td>
<td>34.045</td>
<td>56.662</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dtitle,year</td>
<td>42.486</td>
<td>69.558</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>artist,cdextra</td>
<td>41.002</td>
<td>69.299</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>artist,genre</td>
<td>31.813</td>
<td>53.392</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>artist,year</td>
<td>42.209</td>
<td>69.177</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>IMDB</td>
<td>title</td>
<td>4.049</td>
<td>18.318</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>director</td>
<td>4.499</td>
<td>26.308</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>title,director</td>
<td>3.846</td>
<td>13.895</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>title,year</td>
<td>2.006</td>
<td>18.884</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>director,year</td>
<td>2.020</td>
<td>19.150</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>Country</td>
<td>name</td>
<td>9.269</td>
<td>18.550</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>name,language</td>
<td>37.705</td>
<td>62.738</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>name, city</td>
<td>22.099</td>
<td>40.984</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dob</td>
<td>8.703</td>
<td>39.139</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>phone</td>
<td>8.447</td>
<td>34.030</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>soc sec id</td>
<td>7.199</td>
<td>27.547</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>surname</td>
<td>7.406</td>
<td>30.663</td>
<td>-7</td>
<td></td>
</tr>
<tr>
<td>Persons</td>
<td>dob,soc sec id</td>
<td>6.002</td>
<td>18.359</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dob,phone</td>
<td>6.647</td>
<td>22.136</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dob,surname</td>
<td>5.518</td>
<td>23.232</td>
<td>-8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>phone,soc sec id</td>
<td>6.379</td>
<td>18.378</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>phone,surname</td>
<td>7.037</td>
<td>22.558</td>
<td>-4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>soc sec id,surname</td>
<td>5.261</td>
<td>21.284</td>
<td>-9</td>
<td></td>
</tr>
</tbody>
</table>

Tab. 1 shows the comparison between list-wise (LW) and the complete optimization (LW+PW) strategy. Comparisons are shown as the percentage of the number of comparisons of the unoptimized version. More detailed results, concerning the evaluation of each BK individually, are presented in Tab. 2 and Tab. 3. With the exception of No Opt, all values show averaged results with its corresponding standard deviation.

We can see that there are clear improvements by the addition of the pairwise strategy. For real data, the combined strategies were able to reduce the number of comparisons from about 30% (Cora) to about 80% (CD2), when compared with the list-wise strategy alone. For artificial data, improvements go from about 40% (CD1) to 80% (IMDB).

We note that the higher overall scores in the artificial datasets, both for LW and LW+PW, are caused by the presence of missing data. Missing data causes the appearance of empty BKs, which our approach handles by comparing them to every object in the dataset. Nevertheless, although more candidates are returned, our pairwise optimization strategy is able to avoid a significant number of comparisons for objects that have a low similarity. The same happens in Cora, which is the real world dataset with the highest rate of missing data.

In addition to the number of comparisons, it is also interesting to measure performance in terms of runtime. We note that lower recall values imply a faster execution, thus we do not include BKs that stayed below the defined threshold, since it would introduce a (favorable) bias into our comparison. This rationale holds for the rest of our experiments.

Runtime values confirm the tendency verified by previous tests. Again, the most significant results are obtained on the artificial datasets, which have higher rates of missing data. For most of the real world data, improvements of the pair-wise strategy were smaller, due to the low number of candidates produced by the list-wise optimization. Interestingly, the Cora dataset presented one of the highest decreases in runtime. This reinforces the importance of the pair-wise optimization in cases where a weak BK is selected, or when a damaged dataset causes the list-wise optimization to produce an excessive number of non-duplicate candidates.

On all efficiency tests, standard deviation is quite high. This is expected, since the BKs tested have very different quality degrees, which forces the algorithm to have a loose range of parameter values. However, we note that the standard deviation for the combined approach is proportionally lower in most of the real world cases. Once more, this confirms the advantage of using a combined optimization strategy.

Finally, it is important to note that there was no significant impact on effectiveness. In fact, in datasets like Restaurant and IMDB, some improvements were even achieved. This can be explained by the fact that, during the comparisons, some false duplicates that would otherwise be erroneously considered, were discarded. For the remaining datasets, results show that losses in precision go from 0% to 8%. On average, the difference from the non-optimized algorithm is of about 3%.

6.2.2. Comparison with SXNM

To conclude our experiments regarding our two-level optimization approach we compare it with the SXNM system [18]. We believe SXNM provides a good ground comparison for two main reasons. First, like XMLDup, it was specifically developed to handle the characteristics of XML. Second, given that SXNM is based on a sliding window approach, we are able to compare the impact of using two distinct efficiency improvement techniques. Contrarily to XMLDup, which uses all attributes for comparison, SXNM uses only those that produced the best results. We selected these manually to obtain the best possible results. In Tab. 4 we present the results achieved by both algorithms.

Our comparison shows different behaviors for artificial and real data. For real world datasets SXNM has higher recall, with the exception of Cora. This occurs because our method explicitly attempts to optimize the number of comparisons at
the expense of recall. However, for the artificial datasets, we observe a drastic fall in the number of duplicates detected by SXNM. The phenomenon that causes this decrease is again explained by the appearance of empty BKs. Unlike SXNM, XMLDup deals with this by comparing empty BKs to all objects. Therefore, SXNM is more prone to overlooking true duplicates. Naturally, this effect is evident in the algorithm precision, where results for artificial data also appear quite inferior to those achieved by XMLDup, showing a decrease of 61% in the worst case.

Regarding runtime, as expected, the way XMLDup deals with empty BKs makes it slower than SXNM on the artificial datasets, since it has to compare a much higher number of objects. However, when real world data was tested, XMLDup was able to perform more efficiently in 3 of the 4 datasets. The exception was CD 2 where the poor performance is related to the higher overhead introduced in canopy clustering when long BKs are used.

Finally, we conclude that the appropriate selection of a BK impacts SXNM more than XMLDup. This can be seen on the standard deviation values, which are, in general, lower in XMLDup both for recall and r-precision. We also stress that even when XMLDup recall values were inferior, they did not differ by more than 8%. Moreover, XMLDup proved to be more efficient in real world scenarios, even having to compare a larger number of attributes. This, again, reinforces the importance of our pruning strategy.

### 6.2.3. Selecting the best BKs and Attributes

We conclude our experiments with our method to optimize our two-level strategy by selecting an appropriate BK and eliminating the comparisons of less relevant attributes. The results presented in this section are for the two-level optimization procedure using an automatically selected BK and comparing all attributes and for the two-level optimization procedure using an automatically selected BK and comparing only the best automatically selected attributes. For comparison, we include the results from the previous section. Fig. 5 presents runtime results for each of these configurations.

Experiments reveal that our automatic BK approach produces the best performance in 5 of the 8 datasets. The worst results occurred for the Restaurant dataset because of its poorer attribute selection. This was caused by the inclusion of the address attribute which, although contain-

Table 4. Comparison with SXNM.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Recall (%)</th>
<th>R-Precision (%)</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LW+PW</td>
<td>SXNM</td>
<td>LW+PW</td>
</tr>
<tr>
<td>Country</td>
<td>92.33 ± 2.52</td>
<td>13.33 ± 1.15</td>
<td>98.67 ± 2.31</td>
</tr>
<tr>
<td>CD 1</td>
<td>95.33 ± 2.18</td>
<td>74 ± 7.18</td>
<td>88.11 ± 1.69</td>
</tr>
<tr>
<td>IMDB</td>
<td>98 ± 0</td>
<td>55 ± 16.09</td>
<td>83.33 ± 1.53</td>
</tr>
<tr>
<td>Persons</td>
<td>89.8 ± 0.84</td>
<td>43.4 ± 12.5</td>
<td>89 ± 1.22</td>
</tr>
<tr>
<td>Cora</td>
<td>91.67 ± 1.53</td>
<td>31.67 ± 4.51</td>
<td>92.33 ± 3.21</td>
</tr>
<tr>
<td>CD 2</td>
<td>70 ± 0</td>
<td>78 ± 0</td>
<td>80 ± 0</td>
</tr>
<tr>
<td>Restaurants</td>
<td>92.5 ± 2.12</td>
<td>99 ± 1.41</td>
<td>91.5 ± 3.54</td>
</tr>
<tr>
<td>IMDB+FD</td>
<td>87 ± 0</td>
<td>94 ± 0</td>
<td>98 ± 0</td>
</tr>
</tbody>
</table>
Figure 6. Recall achieved using the combined optimization (LW+PW), the Automatic BK selection (ABK) and ABK with the best attributes selected automatically (ABK_SA).

We can see that, for the majority of the datasets, there is no significant distinction in the number of detected duplicates. Only for the Restaurants and the Cora a reduction can be observed. This happens because of the less effective automatic BK selection, which leads to a drop in recall below the allowed during the training phase. Also, note that the results presented for LW+PW only consider BKs that do not violate this condition. Nevertheless, the obtained recall still stays close to 80% in both cases.

Finally, we look at the quality of the duplicates to see if the gain in performance achieved by the added BK and attribute optimization holds a deterioration in the precision of the duplicate detection algorithm. Fig. 7 shows r-precision results for the tested datasets.

Figure 7. R-precision achieved using the combined optimization (LW+PW), the Automatic BK selection (ABK) and ABK with the best attributes selected automatically (ABK_SA).

As previously observed in Sec. 6.2.1, the choice of the BK has only a small impact in the precision of the duplicate detection. In fact, r-precision does not vary by more than about 5%. The conclusion is that the reduction in the number of attributes to only the most relevant ones, despite improving efficiency, does not cause true duplicates to be erroneously discarded.

With the exception of the Country dataset, this version of the algorithm always manages to stay very close to the results achieved when using all attributes. This happens because the average distinctiveness of the attributes within this dataset is lower than among the remaining. Therefore, results suffer from the absence of a more complete set of attributes that complement the information required to decide whether objects are duplicates.

7. CONCLUSION

In this paper we presented a novel approach for duplicate detection in XML data. Our method combines a traditional blocking strategy, which acts at object comparison level, and a pair-wise optimization strategy, which acts at attribute comparison level. In addition, we show how to automatically learn the parameters required by both strategies, without the need of specific knowledge of the database being processed. These parameters include the automatic discovery of a blocking key, obtained from the selection of the best attributes to use in the duplicate detection. Moreover, we presented an attribute selection procedure that can further optimize our initial method.

The main contributions of our work are two-fold. First, given that the candidates resulting from a traditional blocking strategy frequently contain non-duplicate objects, we present a complementary strategy that is able to detect, at a deeper level, which objects are not similar enough to be duplicates and can eliminate a significant amount of costly operations. Second, we propose a method to avoid the difficult task of manually tuning the optimization parameters, which requires no knowledge of the database being processed.

Experiments performed on both artificial and real world datasets revealed that our proposal is effective, even when only a small amount of training data is available. Moreover, results showed that, despite the efficiency improvements, our optimization strategy does not cause the quality of the results to decrease significantly. We also compared our approach to a state of the art XML duplicate detection algorithm, called SXNM. Our approach was able to outperform SXNM in most cases, either in efficiency or effectiveness.

The promising results achieved leave room for further improvement. We intend to study solutions for the parallel execution of the algorithms here presented. Finally, we plan to develop the possibility of automatically learning the optimization parameters in an fully unsupervised manner.

8. ACKNOWLEDGMENTS

This work was supported by FCT - Fundação para a Ciência e a Tecnologia, under project PEst - OE/EEI/LA0021/2011, project SMARTIS - PTDC/EIA-EIA/115346/2009 and PhD grant SFRH/BD/41450/2007.

9. REFERENCES


### ABOUT THE AUTHORS:

<table>
<thead>
<tr>
<th>Image</th>
<th>Biography</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Luís Leitão" /></td>
<td>Luís Leitão studied Computer Engineering at the Superior Technical Institute of the Technical University of Lisbon. He received his MS degree in 2007, after finishing his master thesis on duplicate detection in XML databases. After working for the industry in between degrees, he started in 2008 his PhD studies at the Superior Technical Institute in Lisbon, Portugal.</td>
</tr>
<tr>
<td><img src="image2" alt="Pável Calado" /></td>
<td>Pável Calado received a degree in Computer Engineering from the Superior Technical Institute of the Technical University of Lisbon. In 2000 he received an MS degree in Computer Science from the Federal University of Minas Gerais (UFMG), where he also obtained his PhD in 2004. He is currently an assistant professor at the Superior Technical Institute and a researcher at INESC-ID, in Lisbon.</td>
</tr>
</tbody>
</table>
ABSTRACT
A recent trend in data stream processing shows the use of advanced continuous queries (CQs) that reference non-streaming resources such as relational data in databases and machine learning models. Since non-streaming resources could be shared among multiple systems, resources may be updated by the systems during the CQ-execution. As a consequence, CQs may reference resources inconsistently, and lead to a wide range of problems from inappropriate results to fatal system failures. In this paper, we address this inconsistency problem by introducing the concept of transaction processing onto data stream processing.

In the first part of this paper, we introduce CQ-derived transaction, a concept that derives read-only transactions from CQs, and illustrate that the inconsistency problem is solved by ensuring serializability of derived transactions and resource updating transactions. To ensure serializability, we propose three CQ-processing strategies based on concurrency control techniques: two-phase lock strategy, snapshot strategy, and optimistic strategy. Experimental study shows our CQ-processing strategies guarantee proper results, and their performances are comparable to the performance of conventional strategy that could produce improper results.

In the second part of this paper, we try to improve the performance of our proposed strategies from the viewpoint of operator scheduling. We notice a characteristic of our proposed strategies: operators could be re-evaluated to prevent non-serializable schedules causing performance degradation. We find the fact that the number of operator re-evaluation depends on operator scheduling, and propose a scheduling constraint that reduces the re-evaluation. Experimental study shows our constraint’s effectiveness: if we add the proposed constraint to operator scheduling, throughput increases up to 5.2 times compared to the naïve scheduling without the constraint.\(^1\)

Categories and Subject Descriptors
H.2.4 [Database Management]: Systems—Query Processing

General Terms
Algorithms, Theory

Keywords
Continuous Query, Transaction, Concurrency Control

\(^1\)Copyright is held by the authors. This work is based on an earlier work: SAC’13 Proceedings of the 2013 ACM Symposium on Applied Computing. Copyright 2013 ACM 978-1-4503-1656-9/13/03. http://doi.acm.org/10.1145/2480362.2480514

1. INTRODUCTION
Data stream processing has been gaining notable attention for several years with the rise of real-time data such as highly-frequent access logs, massive amount of messages from micro-blogs, and reports from wireless sensor nodes. A recent trend shows a new kind of data stream processing pattern with continuous queries (CQs), in which a wide variety of non-streaming resources (e.g., relational data in database systems and model data in machine learning systems) is referenced by data stream management systems (DSMSs) along with streaming data to conduct advanced analysis [7, 8, 13, 14, 16, 21–23]. However, such resources tend to be shared and updated by other systems (Figure 1), and conventional DSMSs could produce improper results in such situation. Surprisingly, in the field of data stream processing, most of the research about the use of external resources do not allow systems to update resources [7,21–23], and so far only a few attempts have been made for such situations [5,9]. In this paper, we consider situations where systems are allowed to share and update resources, and address the problem inherent in these situations.
Figure 2 illustrates a problem inherent in the situation, where resources referenced by DSMSs are updateable by other systems. In the figure, incoming events \( e_1, e_2, e_3, \) and \( e_4 \) are respectively combined with a resource \( x \) by a join operation, and the results of the join operation are finally aggregated every four events. In midstream, the resource \( x \) is updated to \( x' \) by an application. The figure shows three timing of the resource update: (a) after the arrival of \( e_2 \), (b) after the arrival of \( e_1 \), and (c) before the arrival of \( e_1 \). In (a), the aggregation result \( a_1 \) includes both \( x \) and \( x' \), that is, referenced resources are inconsistent. In contrast, referenced resources are consistent in (b) and (c).

Inconsistently referenced resources epitomized by (a), hereafter referred to as referenced-resource inconsistency, could lead to a wide range of problems from inappropriate results to system failures. For instance, if a relational table is referenced as a resource, and the schema of the table is updated in midstream, then aggregation of events that have referenced multiple schema may produce improper results or cause system failures. Therefore, preventing the inconsistency problem is a key aspect for advanced DSMS that heavily reference shared resources and conduct aggregation.

If a CQ is as simple as Figure 2, preventing referenced-resource inconsistency is not a difficult problem. Since we know that join results are aggregated every four events, (1) we prevent the resource from being updated until the resource is referenced four times, (2) release the lock and allow applications to update the resources, and then repeat from the step (1). However, in real systems, CQs are highly-complex, and aggregation operations might be entangled. In such case, we cannot employ the straightforward approach described above.

In this paper, we introduce a solution to referenced-resource inconsistency problems, which works for highly-complex CQs successfully. By introducing the concept of transaction processing onto data stream processing, we describe inconsistency problem in terms of transaction processing. We show a way to derive read-only transactions from CQs, and illustrate that the inconsistency problem is solved by ensuring serializability of transactions. We propose three CQ-processing strategies that ensure serializability of transactions, and measure their performance in a real system.

We also try to improve the performance of our strategies from the viewpoint of operator scheduling. We notice a characteristic of our strategies: some operators are re-evaluated to prevent non-serializable schedules when window-operation target overlaps causing performance degradation. We address this issue in the following steps: (1) figure out the fact that number of operator re-evaluation depends on operator scheduling, (2) propose a scheduling constraint that reduces the number of operator re-evaluation. Experimental study shows our constraint’s effectiveness: if we add the proposed constraint to operator scheduling, the number of operator re-evaluation decreases, and throughput increases up to 5.2 times compared to the naive scheduling without constraint.

The rest of the paper is organized as follows. In Section 2, we describe basic concepts related to data stream processing. In Section 3, we show an idea that derives transactions from CQs, and define the problem clearly. We propose three CQ-processing strategies based on concurrency control techniques in Section 4 and Section 5. In Section 6, we present an experimental evaluation of proposed strategies in a real system. Then, we introduce a new problem, efficient operator scheduling in our CQ-processing strategies, in Section 7. To solve the problem, we propose a scheduling constraint in Section 8. Section 9 shows the effectiveness of our proposed constraint through experiments in a real system. We cover the related work in Section 10. Finally, we conclude in Section 11.

2. PRELIMINARIES

In this section, we describe basic concepts of data stream processing concisely.

2.1 Data Stream

A data stream is a sequence of indefinitely arriving events. To represent data streams formally, a number of models have so far been proposed [2,4,20]. In this paper, we use the following model, which originates from a widely received model based on relational algebra [2].

Definition 2.1 (Data Stream) A data stream \( S \) is a set of an event \( e : \langle s, t \rangle \), where \( s \in S_{\text{schema}} \) is a tuple that belongs to \( S \)'s schema \( S_{\text{schema}} \) and \( t \in T \) is the timestamp of the event.

2.2 Continuous Query and Operator Tree

To process a massive amount of streaming data in on-line fashion, data stream management systems (DSMSs) [1,2,14,17] have been actively studied and developed so far. DSMSs receive continuous queries (CQs) written in human-readable query languages [2,12], and translate them into internal physical plans called operator trees. An operator tree consists of leaf nodes that represent input data streams, and non-leaf nodes that represent relational operators. Each operator has at least one input stream and at least one output stream. For leaf nodes, input streams are raw data streams (input streams for CQs). For non-leaf nodes, input streams are output streams of other operators.

When an event from a data stream arrives at a DSMS, the event is routed to a corresponding leaf node. Then, the event is forwarded to the parent operator of the leaf node and processed by that operator. The operator pushes the result into its output stream, and if the parent operator of the operator exists, the parent operator takes the result from its input stream, and process it in turn. By repeating this process, events climb up the operator tree, and the root operator output events continuously.

For a result of an operator, events used by the operator to produce the result are formally represented by the following function.

Definition 2.2 (Dependent Event Function) Let \( S \) be a set of all data streams appears in a continuous query. The dependent event function \( \text{DEP} : S \rightarrow 2^S \) for an event \( e \in S \in S \) is defined by

\[
\text{DEP}(e) := \bigcup_{n=1}^{\infty} S_n^{\text{dep}}, \text{if } S \text{ is an output stream of an operator whose input streams are } S_1^n, \ldots, S_m^n, \text{ and } e \text{ is produced by the operator using } S_1^{\text{dep}} \subseteq S_1^n, \ldots, S_m^{\text{dep}} \subseteq S_m^n.
\]

\[
\text{DEP}(e) := \emptyset, \text{if } S \text{ is a raw input stream for the CQ (} S \text{ is a leaf node of an operator tree).}
\]
the number of events in a window, and slide $s$ determines the shift between consecutive windows.\(^{2}\) If $s < r$ for windows, they are referred to as overlapping windows. Figure 3 shows an example of non-overlapping windows and Figure 4 shows an example of overlapping windows.

2.2.2 Reference Operator

Operators that reference non-streaming external resources in order to produce their results are called reference operators, which include stream-relation join and data stream classification using external classifiers. For a result of an operator, resource reference operations conducted in order to produce the result are formally represented by the following function.

Definition 2.3 (Dependent Reference Function) Let $S$ be a data stream and $X$ be a set of all external resources possibly referenced in a CQ. The dependent reference function $REF : S \rightarrow \{X \times T\}$ for $e \in S$ is defined by

- $REF(e) := \langle (x_1, t_1), ..., (x_n, t_n) \rangle$, if $S$ is an output stream of a reference operator and each external resource $x_k \in X(1 \leq k \leq n)$ is referenced at the time $t_k \in T$ in the process of $e$ at the reference operator.
- $REF(e) := \emptyset$, otherwise.

3. CQ-DERIVED TRANSACTION

In this section, we introduce CQ-derived transaction, a concept that derives read-only transactions from CQs. We also define the problem we address in terms of CQ-derived transaction.

3.1 Consistency Problem Revisited

In terms of transaction processing, we revisit the three situations illustrated in Figure 2, where events from a data stream are combined with an external resource $x$, and the results are aggregated by a window operation ($r = 4$) in a DSMS. Here, we regard successive resource reference operations whose results are involved in the same aggregate operation as a transaction, and the result output process of the aggregate operation as the commit operation of the transaction; we use $r_1(x)$ to denote a reference operation to the resource $x$ made by the transaction, and $c_1$ to denote its commit operation. Then, we consider that the resource $x$ is updated by a transaction from a system (not the DSMS); we use $w_2(x)$ to denote the update operation, and $c_2$ to denote the commit operation of the transaction. By using these notations, three situations can be represented as schedules of the two transactions: a non-serializable schedule represented in Example 3.1, and serializable schedules represented in Example 3.3 and Example 3.2.

Example 3.1 (Referenced $x$ and $x'$)

$r_1(x), r_1(x), w_2(x), c_2, r_1(x), r_1(x), c_1$

Example 3.2 (Referenced $x$)

$r_1(x), r_1(x), r_1(x), r_1(x), c_1, w_2(x), c_2$

Example 3.3 (Referenced $x'$)

$w_2(x), c_2, r_1(x), r_1(x), r_1(x), r_1(x), c_1$

The goal of this paper is to ensure that a CQ consistently references non-streaming resources when the CQ produces a result that depends on the resources. That is to say, our objective is to prohibit problematic schedules, as shown in Example 3.1, whose execution orders lead to inconsistent results. In transaction processing, such schedules are referred to as non-serializable schedules, and a number of concurrency control techniques for transactions have so far been studied. In order to prevent problematic schedules, we consider a running CQ as consecutive read-only transactions, and evaluate operators in strategies based on concurrency control techniques.

We show how transactions are derived from a CQ in Section 3.2, and we describe when derived transactions are regarded as committed in Section 3.3. Then, by using these concepts, we enunciate the problem in Section 3.4.

3.2 Deriving Transactions

Since a CQ needs to consistently reference a resource to produce a consistent result, we have to evaluate the sequence of reference operations conducted to produce the result transactionally. Thus, we consider a sequence of reference operations in reference operators used to produce a final result of a CQ as a transaction, hereafter referred to as a CQ-derived transaction. To represent a CQ-derived transaction formally, we employ the function $REF$ introduced in Section 2.2.2 and introduce a function that maps a result of an operator to a set of all reference operations conducted for the result, which may include operations in upstream operators, as follows.

Definition 3.1 (Transaction Deriving Function) Let $S$ be a data stream and $X$ be a set of all external resources possibly referenced in a CQ. The transaction deriving function $TXN : S \rightarrow \{X \times T\}$ for $e \in S$ is recursively defined as follows.

$$TXN(e) := \bigcup_{e' \in \text{dep}(e)} TXN(e') \cup REF(e)$$

3.3 Commit Operator

The commit operation for a CQ-derived transaction can be conducted safely after the root operator of the CQ computes the result that corresponds to the CQ-derived transaction, since all reference operations related to the results are conducted transactionally, and there is no possibility the CQ produces an inconsistent result. However, the commit operation for a CQ-derived transaction can be conducted earlier for most CQs. For instance, in the operator
4.1 Two-Phase Lock Strategy

A two-phase lock strategy ensures that until a transaction commits, no other transactions can read-lock when it tries to reference a resource, and keeps the read-lock until the corresponding CQ-derived transaction commits. By enforcing the lock compatibility shown in Table 1 for both CQ-derived transactions and update transactions, two-phase lock strategy ensures that until a transaction commits, no other transactions are able to update resources referenced in the transaction. In doing so, this strategy ensures conflict serializability of transactions.

Running times of CQ-transactions are unpredictable and could be very long, since they highly depend on data stream rates. Thus, transactions in two-phase lock strategy could retain read-locks for a long time, preventing resources from being updated by other systems. Therefore, two-phase lock strategy is not suitable for applications where resource update throughput is important.

4.2 Snapshot Strategy

Second, we introduce snapshot strategy. In contrast to two-phase lock strategy, CQ-derived transactions in this strategy do not retain locks for resources during their running time. Instead, when a CQ-derived transaction begins, the transaction copies all resources it will reference as snapshots. Then, the snapshots are referenced by reference operations in the transaction instead of the current resources. When a CQ-derived transaction commits, all snapshots taken for the transaction are expired and removed. By referencing snapshots at the same point in time in each CQ-derived transaction, this strategy ensures conflict serializability of transactions.

4.3 Supporting Overlapping Window

Even in the above two pessimistic strategies based on concurrency control, some conventional techniques for overlapping windows could lead to non-serializable schedules. One of such techniques is synopsis [1]. A synopsis belongs to an overlapping window-based operator and caches past input events that may be used by future window operations in the operator. By doing so, synopses suppress re-evaluation of upstream operators. While the use of synopses enhances performance of CQ-processing, under our concurrency control strategies, it corresponds to reusing results of transactions among consecutive transactions. In this case, if an update transaction from another system is executed between the consecutive transactions, a generated schedule could be non-serializable (Window 2 in Figure 4).

To prevent non-serializable schedules, we introduce a technique that ensures serializable schedules even with overlapping windows. The main idea of this technique is re-evaluating limited operators in order to prevent reuse of transaction results while leveraging synopses.

4.3.1 Redo Queue

We prepare auxiliary queues called redo queues that back up input events at certain input streams: (1) input streams of a reference operator that has window-based operators in its downstream (ancestor operators in the operator tree) and does not have reference operators in its upstream (descendant operators in the operator tree), and (2) input streams that do not have reference operators in its upstream and belong to a window-based operator that has at least one reference operator in its upstream. If an event arrives at an input stream that has a redo queue, the event is added to the redo queue. The events backed up might be used in re-evaluation of operators.

Figure 5 represents an example of redo queue placement for an operator tree. To conduct such redo queue arrangement efficiently, we use the procedure SETUP-REDO-QUEUES shown in Algorithm 1 in our DSMS implementation. For an operator tree rooted by root, we call the procedure with the two arguments root and false, and then the redo queues for the operator tree are placed efficiently, since the procedure is essentially a DFS algorithm.

If we reuse snapshots among multiple transactions, schedules of the transactions become multi-version whereas schedules in two-phase lock strategy are mono-version [25].
4.3.2 Redo and LWM

When the commit operator in an operator tree produces a result, the corresponding CQ-derived transaction for the result commits and finishes. From the viewpoint of serializability, events in synopses, which may contain results of past read operations, could be improper to be reused in the consecutive transaction. For this reason, when a CQ-derived transaction commits, we clear events in synopses that may contain improper events and re-input events in redo queues into corresponding input streams. Synopses to be reset are limited to those that belong to the subtree whose root node is the commit operator and leaf nodes are operators with redo queues. Such a subtree of an operator tree is called a redo area, and an example of the redo area is represented in Figure 5.

To decide events that will be used in the consecutive transaction, we introduce the following Low Water-Mark (LWM).

Definition 4.1 (Low Water-Mark) Let \( S \) be a data stream. Low Water-Mark for an event \( e \in S \) is

\[ \text{LWM}(e) = \begin{cases} \text{min}(t_e) & \text{if } e \text{ is a raw input} \\ \text{max}(t_e) & \text{if } e \text{ is a result} \end{cases} \]

where \( t_e \) is the timestamp of event \( e \).

By using LWM, the redo process after the commit operation of a CQ-derived transaction is conducted as follows. First, the biggest LWM is selected from events in overlapping windows involved in the transaction, as represented in Figure 6. Second, we reset synopses that belong to operators in the redo area, and re-input events whose timestamps are equal to or higher than the selected LWM. After that, re-inputted events are processed by operators as done previously.

The LWM of an event indicates that events whose timestamps are smaller (older) than the LWM are not needed to compute the event. Thus, when a CQ-derived transaction commits, we can remove the events whose timestamp is smaller than the selected LWM. We use LWM in our DSMS implementation.

5. OPTIMISTIC STRATEGY

In this section, we propose a CQ-processing strategy based on optimistic concurrency control [18]. Optimistic concurrency control consists of three phases: read phase, validate phase, and write phase. Since CQ-derived transactions are read-only, we can omit the write phase. The detailed steps of our optimistic strategy can be represented as follows.

- **Read phase**: Each reference operator references resources.
- **Validate phase**: When a commit operator is ready to output a result, if read operations in the CQ-derived transaction are not interleaved by write operations from update transactions, commit the CQ-derived transaction. Otherwise, abort and redo the CQ-derived transaction.

To implement optimistic strategy, we have to consider two topics: (1) a way to achieve validation and (2) a way to achieve abort and redo of transactions. In this paper, we present simple approaches, and use them in our DSMS implementation.
5.1 Validation

To achieve validation, we assume that a resource holds the timestamps of the latest update operation on the resource, and associate the timestamp with a read operation that references the resource at the time. Such a timestamp is referred to as a read operation’s referenced-resource timestamp. In the validation phase of a CQ-derived transaction, we compare the transaction’s beginning time \( t_0 \) with read operation’s referenced-resource timestamps \( t_1, \ldots, t_n \). If there exists \( k \) such that \( t_k > t_0(1 \leq k \leq n) \), then another transaction must have updated the resource in the middle of the CQ-derived transaction, and the CQ-derived transaction turns out to be invalid. Otherwise, the CQ-derived transaction is valid, since it can be a member of a conflict-serializable schedule.

5.2 Abort and Redo

Since CQ-derived transactions are read-only, they do not make any changes to resources. Therefore, the process for aborting and redoing a transaction is simply canceling the commit operator’s operation and re-inputting the events the transaction depends on. To achieve this process, we apply the redo process mechanism of pessimistic concurrency controls described in Section 4.3.2 without any modification.

6. EXPERIMENTS: STRATEGIES

We perform real-machine experiments to capture characteristics of our proposed CQ-processing strategies.

6.1 Experimental Setup

We have conducted experiments with our DSMS to evaluate proposed continuous query processing strategies. The DSMS is written in C++ and implements the three CQ-processing strategies proposed in this paper. To compile the DSMS, we used GNU g++ 4.6.3 and specified -02 as the optimization level. Experiments have been conducted on a machine equipped with a quad-core Intel Core2 Quad Q9400 CPU (2.66 GHz, 6M L2 Cache, 1333MHz FSB) and 4GB DRAM running Ubuntu Linux with 3.2.0 kernel.

To simulate situations where resources referenced by CQs are updatable by other systems, we register a simple CQ to the DSMS, by no concurrency control support strategy, “S2PL” means two-phase lock strategy described in Section 4.1, “Snapshot” means snapshot strategy described in Section 4.2, and “Optimistic” means optimistic strategy described in Section 5.

6.2 Sensitivity to Update Transaction

First, to see how our CQ-processing strategies are interfered by update transactions, we vary dispatch rate of update transactions from 1 (txn/s) to 100,000 (txn/s), and measure two metrics in the DSMS: (a) proportions of consistent results of the CQ, and (b) throughput of the CQ. Window range \( r \) for the aggregation operator of the CQ is 5.

6.2.1 Consistent Result Ratio

Figure 7(a) represents proportions of consistent results of the CQ for each CQ-processing strategy. While all results of the CQ are consistent in our CQ-processing strategies, the proportion of consistent results in conventional strategy degrades with the increase of update transaction’s arrival rate. Note that even in the case where the arrival rate of update transactions is minimum (1 tps in this experiment), there are several inconsistent results. This means that in applications where no inconsistent results are permitted, conventional strategy cannot be employed.

6.2.2 CQ Throughput

Figure 7(b) shows throughput of the CQ in each CQ-processing strategy. When the arrival rate of update transactions is low, optimistic strategy shows high throughput comparable to the throughput of conventional strategy. However, with the increase of update transaction’s arrival rate, the throughput of optimistic strategy rapidly decreases and becomes the worst one. This is because frequent updates to resources increase the probability that CQ-derived transactions are aborted in the validation phase.

6.3 Sensitivity to Window Range

Second, to see the impact of window range in each CQ-processing strategy, we vary the window range \( r \) of the aggregation operator from 5 to 640, and measured two metrics in the DSMS: (a) throughput of update transactions, and (b) throughput of the CQ. In this experiment, dispatch rate of update transactions is fixed to 100,000 (txn/s), and effective throughput of update transactions are measured.

6.3.1 Update Transaction Throughput

Figure 8(a) shows throughput of update transactions in each CQ-processing strategy. The throughput in two-phase lock strategy is worst, since CQ-derived transactions in the strategy retain readlocks for resources, and block update transactions during their lifetime.
In addition, we can find a trend in Figure 8(a): with the increase of the window range, throughput of update transactions degrade in every CQ-processing strategies. A reason of this trend is inherent in the operator scheduler of the DSMS. In our DSMS, operators in an operator tree are visited in round-robin fashion, and the visited operator is evaluated only if the operator fulfills the condition (e.g., a synopsis has proper number of events). As a consequence, when the window range is quite long, aggregation operator is rarely evaluated and other operators (i.e., join and selection) are evaluated frequently. Since join operator interferes update transactions, frequent evaluation of join operator degrades the throughput of update transactions.

6.3.2 **CQ Throughput**

Figure 8(b) shows throughput of the CQ in each CQ-processing strategy. With the increase of window range, throughput of the CQ in optimistic strategy gradually degrades. When the number of events in a window increases, the number of corresponding reference operations also increases, and CQ-derived transactions become longer. This situation leads to a high probability of CQ-derived transaction’s abortion, and degrades CQ throughput in optimistic strategy.

We also find a trend in Figure 8(b) as in Section 6.3.1: with the increase of the window range, throughput of CQ transactions increase in not optimistic CQ-processing strategies. The cause of this trend is trade-off between CQ throughput and update transaction throughput. For the reason described in 6.3.1, throughput of update transactions degrades with the increase of window range. As a result, resource reference operations in the CQ are less blocked by the update transactions, and CQ throughput increase.

6.4 **Discussion**

Experimental results showed our proposed strategies always produce proper results that reference resources consistently as described in Section 6.2.1, and their throughput are comparable to the throughput of naïve strategy in certain conditions. Since our proposed strategies are not optimal, we should investigate characteristics of an application and carefully choose proper strategies at the time of system deployment. We plan to study automation of such task and adaptive switching of strategies as future work.

7. **OPERATOR SCHEDULING PROBLEM**

As we have seen so far, in CQ-processing with concurrency control, operators could be cleared and re-evaluated when a CQ-derived transaction commits degrading performance. In the rest of this paper, we try to reduce the number of operators to be cleared and improve the performance by carefully scheduling operators. In this section, we show how operator-scheduling affects the performance and define the problem.

7.1 **Operator Scheduler**

In CQ-processing based on relational-algebra, every operators in an operator tree can run autonomously, that is, even if we execute operators in arbitrary orders, we are guaranteed to get the same result stream from the operator tree. In other words, operator-scheduling does not affect the content of result stream. However, it is known that operator-scheduling does affect performance characteristics: memory-usage, throughput, and latency. Extensive studies have been made on this research area [3, 6].

Here, we define several words. When an operator brings out a tuple from its input queue, inserts the tuple to its synopsis, and processes tuples in the synopsis if possible, we say that the operator is evaluated once. When an operator can be evaluated, that is, at least an input queue of the operator has a tuple, we say that the operator is evaluable.

In this paper, we assume that an operator tree is executed by a scheduler repeating the following two phases:

1. **Operator selection phase**: The scheduler selects an evaluable operator.

2. **Operator evaluation phase**: The selected operator is evaluated.

A sequence of selected operators is referred to as a schedule.

7.1.1 **Batch-Scheduler**

We refer to a scheduler that continuously evaluates an operator as much as possible as a batch-scheduler. Most schedulers [3, 6] are classified as batch-schedulers. In a batch-scheduler, an operator is continuously evaluated reducing the number of context-switches and CPU-cache misses, that could improve the performance. Moreover, with a batch-scheduler, we can conduct operator-specific optimizations. For example, a join operator with an external relation can suppress the number of relation-scans by reusing a scan result between consecutive evaluations.

7.2 **Operator-scheduling and Redo**

Figure 9(a) represents internal states of an operator tree with eight input tuples. The operator tree conducts stream-relation join with an external resource and then aggregates the join results stream by every four tuples. In the figure, the sub-tree surrounded by the dotted rectangle corresponds to the operator tree’s redo-area, and the aggregation operator corresponds to the operator tree’s commit operator. We assume that always produces a result tuple when it consumes a tuple.

From the state of Figure 9(a), we consider schedules that make the commit operator output a result tuple. That is, schedules that make the CQ-derived transaction for the result tuple commit. Since conducts window-operation by range \( r = 4 \), \( \odot \) have to be evaluated four times to conduct a commit operation. Moreover, since the input stream of \( \odot \) is the output stream of \( \odot \), \( \odot \) have to be evaluated at least \( n \) times at \( n \)th evaluation. Figure 9(b) and 9(c) show internal states generated by schedules that fulfill these constraints. Considering CQ-processing with concurrency control.
where a redo-area is cleared after a commit operation, four tuples will be cleared in Figure 9(b) whereas eight tuples will be cleared in Figure 9(c). Since both schedules yield the same result stream, the schedule in Figure 9(b) is better than the schedule in Figure 9(c).

7.3 Simple Scheduling Model

To investigate the performance gap caused by operator scheduling in CQ-processing with concurrency control, we consider best / worst scheduling algorithm for the operator tree in Figure 9(a) with a policy that the smaller number of tuples to be cleared is better. In best scheduling algorithms, $\odot \bigoplus$ have to be evaluated at most $n$ times, where $n$ indicates the number of tuples required by a window operation of $\odot$. Algorithm 2 shows an example for the best scheduling algorithms. On the other hand, in worst scheduling algorithms, $\odot \bigoplus$ have to be evaluated as much as possible before $\odot$ consumes the outputs of $\odot \bigoplus$. Algorithm 3 shows an example for the worst scheduling algorithms.

Algorithm 2: A best scheduling algorithm for the operator tree in Figure 9(a)

```
while true do
    Select $\odot \bigoplus$ and evaluate it once
    Select $\odot$ and evaluate it once
```

Now, we investigate the performance of two schedules represented in Algorithm 2 and Algorithm 3 in a quantitative way. Assume the input queue initially has $n$ tuples and $\odot$ conducts window-operation with range $r$ and slide $s$. Number of $\odot \bigoplus$ evaluation $N_{\text{best}}$ for the best scheduler (Algorithm 2) and $N_{\text{worst}}$ for the worst scheduler (Algorithm 3) can be represented as follows.

Algorithm 3: A worst scheduling algorithm for the operator tree in Figure 9(a)

```
while true do
    while $\odot \bigoplus$ is evaluable do
        Select $\odot \bigoplus$ and evaluate it once
        Select $\odot$ and evaluate it once
```

Using models described in equation 1 and equation 2, we can compute the number of $\odot \bigoplus$ evaluation in best- and worst-scheduler. Figure 10 shows the difference between the number of $\odot \bigoplus$ evaluation in best scheduler and worst scheduler where $n$ is varied from 5 to 100 and window parameters are $r = 4$ and $s = 1$. In Figure 10, the number of $\odot \bigoplus$ evaluation in the best scheduler is smaller than in the worst scheduler, while both schedulers yield the same result stream. In real systems, $n$ increases in proportion to the input data stream’s arrival rate, and the performance gap between best and worst scheduler widens.

7.4 Problem Definition

The investigations so far indicate that in CQ-processing with concurrency control, the order of operator evaluation (i.e., operator scheduling) affects the number of total operator evaluation and cause considerable performance gaps between the best schedule and the worst schedule. Considering this aspect, we address the following problem in the next section.

Problem (Efficient scheduling) Given an operator tree and its initial input, execute the operator tree to minimize total number of operator evaluation under conditions that each redo-area is cleared and re-evaluated when the commit operator for the operator tree commits.

8. SCHEDULING CONSTRAINT

We notice that a tuple enters the redo-area for an operator tree only if an operator that has input data streams with redo queues is evaluated. We refer to such an operator as a gatekeeper operator. If all non-gatekeeper operators in the redo-area cannot be evaluated,
operator-tree execution does not proceed until the gatekeeper operator’s evaluation. Conversely, if a non-gatekeeper operator in the redo-area is evaluable, the commit operator might commit and lead to re-evaluation. Considering these characteristics, we add the following constraint to operator scheduling to minimize the number of operator evaluation.

**Constraint** The gatekeeper operator for an operator tree can be evaluated only if every non-gatekeeper operators in the operator tree are not evaluable.

### 9. EXPERIMENTS: SCHEDULING

In this section, we show how our proposed constraint improves the performance of CQ-processing with concurrency control through real machine experiments.

#### 9.1 Experimental Setup

We have conducted experiments with our DSMS to evaluate our proposed constraint for operator scheduling. We have used the same experimental environment (i.e., DSMS and machines) as in the previous experiments described in section 6.1. For concurrency control strategy in DSMS, we have used two-phase locking strategy (2PL) in all experiments.

#### 9.2 Batch-Round-Robin Scheduling

To check effectiveness of our proposed constraint, we have implemented a batch-round-robin scheduler that execute an operator tree by repeating the following two steps:

1. **Operator selection phase**: Select an evaluable operator in round-robin fashion.

2. **Operator evaluation phase**: Evaluate the selected operator up to \( n \) times.

In operator scheduling experiments, we have used an operator tree represented in Figure 11. This operator tree joins data streams \( S_1 \) and \( S_2 \) with external resources \( x \) and \( y \), filters join results using a condition, computes mean values of an attribute for the results, and finally chooses mean values using a condition. In our system, \( S_1 \) and \( S_2 \) are queues filled with 1,000 initial tuples, and \( x \) and \( y \) are relations with 1,000 initial tuples. Join operator conducts join by nested-loop algorithm. Selectivities of operators are as follows: \( \theta_{s_1} = 1/|x|, \theta_{s_2} = 1/|y| \), and \( \theta_{s_3} = \theta_{s_4} = \theta_{s_5} = 0.5 \). Window parameters of window-based operators are \( r = s = 10 \) for \( s_3 \) and \( r = 10 \) and \( s = 1 \) for \( s_4 \).

We have measured the number of operator evaluation and throughput in batch-round-robin scheduler varying batch-evaluation count \( n \) from 1 to 10. Figure 12 and Figure 13 represent the results for different join-selectivities of \( s_3 \), \( \theta_{s_1} = 0.5 \) in Figure 12 and \( \theta_{s_3} = 0.15 \) in Figure 13. In each figure, *Naïve* means the result of scheduling without constraint and *Proposed* means the result of scheduling with our proposed constraint.

Figure 12 and Figure 13 represent our proposed constraint’s effectiveness: with-constraint scheduling (Proposed) resulted in the smaller number of operator evaluation and higher throughput compared to without-constraint scheduling (Naïve). When the batch-evaluation count \( n = 10 \), performance gains of proposed constraint are maximum: our proposed constraint lead to 5.2 times higher throughput when \( \theta_{s_3} = 0.5 \) and 1.7 times higher throughput when \( \theta_{s_3} = 0.15 \).

#### 9.3 Discussion

In our experiments, the proposed constraint have successfully reduced the number of operator evaluation in batch-round-robin scheduling. However, contrary to our expectation, increasing batch-processing count \( n \) have not improved the performance (e.g., throughput). As mentioned in Section 7.1.1, one of the big advantage of batch-scheduling with large batch-processing count \( n \) is the opportunity for reducing CPU cache-misses. However, the query-processing layer of our DSMS is not highly optimized, and CPU cache-miss times are not dominant in query-processing times. As a consequence, increasing batch-processing count \( n \) have not improved the performance of our DSMS.

### 10. RELATED WORK

Our work relates to isolation issues in data streams. Conway studied the concept of transactions in data stream processing and proposed to use a window as the isolation unit [9]. There are sev-
eral systems that ensure isolation in a certain unit: each event \cite{10}, events who share the same timestamp \cite{1}, and events in the same window \cite{11, 17}. However, these systems only ensure isolation in the level of an operator, whereas our work ensures isolation in the level of an operator tree, that is, CQ-level.

Transactional issues in data stream processing are discussed by Gurgen et al \cite{15}. They noticed that a CQ-specific metadata such as selection condition can be modified in the middle of CQ-execution and could lead to false alarms. They considered a CQ as a nested transaction, and proposed a concurrency control technique based on optimistic priority-based concurrency control protocol. We believe that our work can be incorporated into their work, since how a CQ is decomposed into nested sub-transactions is a bit unclear in their paper, whereas our work shows clear way to derive transactions from a CQ using windows.

A general transaction model for data streams is studied by Botan et al \cite{5}. They treated both streaming data and non-streaming data equally as objects to which read and write operations are defined. Their work focused on the general transaction model and the amount of discussions about implementation issues is relatively small. In contrast, our work focuses on real systems and shows several important topics for the implementation.

Wang et al. tackled with transactional issues in Active Complex Event Processing (ACEP), a novel CEP concept that can treat states in CEP queries to conduct advanced event pattern matching \cite{24}. In ACEP, multiple queries reference and update a state concurrently, and as a consequence, a query may read the state inconsistently and detect false patterns. To prevent such a problem, they proposed several pessimistic concurrency control techniques. In contrast to their work where the isolation unit is event patterns, we regard whole CQs as the isolation unit, which potentially include entangled windows.

11. CONCLUSIONS AND FUTURE WORK

In this paper, we address the consistency problem inherent in the situations where resources referenced by CQs are updateable by other systems. We have described the problem in terms of transaction processing, and introduced CQ-derived transaction, a concept that derives read-only transactions from CQs. Based on the concept, we have shown that the consistency problem is solved by ensuring serializable schedules of CQ-derived transactions and update transactions. To ensure serializable schedules, we have proposed three CQ-processing strategies: two-phase lock strategy, snapshot strategy, and optimistic strategy. Experimental study showed our proposed strategies ensured consistent results, and the performance of the proposed strategies was comparable to naïve strategy that generated inconsistent results.

We also have improved the performance of CQ-processing with concurrency control by adding a constraint to operator schedulers. Our proposed constraint successfully reduced the number of operator re-evaluation, and achieved 5.2 times higher throughput compared to Naïve methods.

Since each concurrency control strategy has a performance characteristic, we plan to investigate a way to select proper concurrency-control strategies for a target application as described in Section 6.4. We also plan to study a way to switch concurrency control strategies to adapt to dynamically changing characteristics of data streams.

12. ACKNOWLEDGEMENTS

This work was done while the first author was a student at University of Tsukuba, and the work was partially supported by NICT (New generation network R&D program for innovative network virtualization platform and its applications) and KAKENHI (#24500106). We thank Mateus Cruz and Wang Yan for their comments on the paper.

13. REFERENCES


\cite{5} Irina Botan, Peter Fischer Bijay, and Donald Kossmann Nesime Tatbul. Transactional Stream Processing. In EDBT. ACM, 2012.


\cite{9} Neil Conway. CISC 499*: Transactions and Data Stream Processing, 2008.


ABOUT THE AUTHORS:

Masafumi Oyamada is a researcher at Cloud System Research Laboratories, NEC Corp. His research interests include data stream processing, query optimization, transaction processing, distributed systems, and programming language. He received his M.S. and B.S. degrees in Computer Science from University of Tsukuba in 2013 and 2011 respectively.

Hideyuki Kawashima received Ph.D. from Science for Open and Environmental Systems Graduate School of Keio University, Japan. He was a research associate at Department of Science and Engineering, Keio University from 2005 to 2007. From 2007 to 2011, he was an assistant professor at both Graduate School of Systems and Information Engineering and Center for Computational Sciences, University of Tsukuba, Japan. From 2011, he is an assistant professor at Faculty of Information, Systems and Engineering, University of Tsukuba.

Hiroyuki Kitagawa received the B.Sc. degree in physics and the M.Sc. and Dr.Sc. degrees in computer science, all from the University of Tokyo, in 1978, 1980, and 1987, respectively. He is currently a full professor at Faculty of Engineering, Information and Systems, University of Tsukuba. His research interests include data integration, data mining, information retrieval, stream processing, data-intensive computing, XML, and scientific databases. He is an editorial board member of IEEE TKDE and WWWJ, a Fellow of IPSJ and IEICE, Vice Chairperson of the Database Society of Japan, and a member of ACM, IEEE Computer Society, and JSSST.
RTRS: A Novel Real-Time Reasoning System Based on Active Rules

Ying Qiao† QiNan Han†‡ Beijing Key Lab of Human-computer Interaction, Institute of Software, Chinese Academy of Sciences {qiaoying,hanqinan}@iscas.ac.cn
Hongan Wang† Xiang Li‡ Beijing Key Lab of Human-computer Interaction, Institute of Software, Chinese Academy of Sciences hongan@iscas.ac.cn Tsinghua University, China cqlixiang@gmail.com
Kang Zhong† Keming Zhang‡ Beijing Key Lab of Human-computer Interaction, Institute of Software, Chinese Academy of Sciences {kangzhong,keming}@iscas.ac.cn
Jian Liu‡ Anxiang Guo†‡ Shanxi Electric Power Research Institute, China edliu@bylink.com.cn 13060406568@163.com

ABSTRACT
Event streaming processing (ESP) has been well investigated in recent years. Many approaches have been proposed on this aspect. However, none of them considers the timing constraints held by high-level reactive applications. In this paper, we propose a real-time reasoning system based on active rules (i.e., event-condition-action rules), called RTRS, to make automatic decisions about how to react to continuously arriving events (i.e., event streams) so that the deadlines of inference delay for rules can be met as much as possible. A series of simulations are conducted to evaluate the performance of RTRS. Simulation results show that the heuristic searching policy used by the inference algorithm – RTEIA, which is the core of RTRS, effectively improves the number of times that rules are fired within the deadlines of their inference delay (NAFS).1

Categories and Subject Descriptors
H.2.m [Miscellaneous].

General Terms
Algorithms.

Keywords

1. INTRODUCTION
Many reactive applications need to monitor continuously arriving events that are known as event streams, discern the interested situations implied by the sequence of events, and take time-critical actions in response to the identified situations[1]. For example, in high-speed train on-board diagnosis system, a Central Control Unit (CCU) identifies which fault occurs according to the data/events fed from on-board sensors, and sends corresponding fault codes to Man Machine Interface (MMI) for display, and moreover, in response to identified faults, sends the corresponding fault-handling strategies to sub-system control units such as Traction Control Unit (TCU) and Brake Control Unit (BCU) for deceleration or halt.

Event stream processing (ESP) is a fairly recent topic. However, state-of-the-art ESP systems [2-4] may be not good enough for reactive applications since they focus on the detection of composite events and pattern matches with ignorance of the action processing about how to execute actions triggered by the events in well-defined contexts and situations. Some works [5,6] address the combination of event stream processing with active rules, i.e., Event-Condition-Action (ECA) rules [7] to bridge the gap between current ESP practice and reactive applications, but these approaches undergo unexpected latency of response to occurring events since they use query-based complex event processing realized with passive logic programming systems. To overcome this disadvantage, [8] addresses using an inference engine based on ECA rules to provide automatic reasoning service which allows for reasoning over events, conditions and actions uniformly. However, all above works [5, 6, 8] on combining ECA rules with ESP systems ignore the timing constraints held by high-level reactive applications.

The contribution of this paper is to propose a real-time reasoning system based on active rules, i.e., event-condition-action (ECA) rules, called RTRS, to make automatic decisions about how to react to continuously arriving events so that the deadlines of inference delay for rules can be met as much as possible.

The rest of the paper is organized as follows: in section 2, we describe the architecture of RTRS; Section 3 addresses one of key infrastructure for RTRS – VSTE, which is a visual specification tool for ECA rules; In section 4, an event-oriented inference

---

1 Copyright is held by the authors. This work is based on an earlier work: RACS’12 Proceedings of the 2012 ACM Research in Applied Computation Symposium, Copyright 2012 ACM 978-1-4503-1492-3/12/10. http://doi.acm.org/10.1145/2401603.2401696.
algorithm with timing constraints, called RTEIA, which is the core of RTRS, is presented; Section 5 evaluates RTRS via a series of experiments; Section 6 states conclusions and future works.

2. SYSTEM OVERVIEW

Figure 2 shows the architecture of RTRS. RTRS is based on an open rule base. It has two parts, i.e., rule manager and real-time inference engine. The rule manager is responsible to specify, add, delete, modify and store event-condition-action (ECA) rules. It provides a visual rule specification tool, called VSTE, by which the user is able to specify ECA rules in a natural way. VSTE combines the visualization of ECA rules and the XML-based representation as a single tool. The core of this tool is a visual ECA rule language, called VECAS which has a compatible syntax structure with XML. When the user specifies ECA rules with VSTE, the graphical representations of VECAS will be automatically transformed into the form of XML in the background.

![Figure 2. Architecture of RTRS.](image)

The real-time inference engine takes a set of external events (event stream) as inputs and derives corresponding actions taken to react to these events within given deadline according to ECA rules in the rule base. The core of the real-time inference engine is an event-oriented inference algorithm with timing constraints, called RTEIA which exploits heuristic searching on a rule graph to fire the corresponding rules. In this case, a set of predefined ECA rules in the rule base are represented as a rule graph that is a directed graph whose nodes represent events, conditions or actions in ECA rules. Two nodes are connected via a directed edge if one of them needs to transfer information to another. A directed edge starts from a node which creates the information and terminates at a node which receives the information. The node sending out a directed edge is called “child node” while the node to which a directed edge directs is called “parent node”. Furthermore, a node with zero in-degree is called entrance node; it represents a primitive event. A node with zero out-degree is called exit node; it represents an action.

Meanwhile, the purpose of heuristic searching is to find a path from a specific entrance node to an exit node so that the time consumed for traveling along this path is as short as possible and the exit node representing the action in an ECA rule with more urgent inference delay deadline can be found preferentially. This allows RTEIA to make the rule with higher priority fired preferentially. Thus, a heuristic function is defined for each event/condition node in the rule graph. The search starts from a specific entrance node. During the search, the expected path will be expanded with a node selected via the value of the heuristic function. The details of RTEIA will be addressed in section 4.

3. VSTE—A VISUAL SPECIFICATION TOOL FOR ECA RULES

3.1 VECAS—Visual Specification Language for ECA Rules

3.1.1 Modeling

The model behind the language constructs of VECAS can be represented as an augmented graph:

\[ GR = (VR, ER, Attr(VR), M(ER)) \]

\( VR \) is a set of vertices. \( VR = VR_E \cup VR_C \cup VR_A \), where \( VR_E \) is a set of event vertices, \( VR_C \) is a set of condition vertices and \( VR_A \) is a set of action vertices. Each element in these three sets represents an event, a condition and an action respectively. The details of the events, conditions and actions supported by VECAS can be found in [9] and are skipped in this paper.

\( ER \) is a set of edges. Each edge is a connection between different types of vertices. VECAS supports following three types of connections: EC connection (The relationship between an event and a condition; It means that when the event is detected, the condition will be examined.), CA connection (The link between a condition and an action; It means that when the condition is examined to be satisfied, the action will be executed.) and RR connection (The triggering relationship between two ECA rules; It means that execution of one rule may trigger execution of another.).

\( Attr(VR) \) is a set of attributes applied to each vertex to describe its critical characteristics. The attribute sets of each type of event, condition and action are addressed in [9] and are skipped in this paper.

\( M(ER) \) is a set of constraints on each edge. It describes the characteristics of each connection. These constraints represent coupling modes between events, conditions and actions [7].

3.1.2 Syntax Structure

Based on the model addressed in section 3.1.1, the syntax of VECAS can be graphically represented as Figure 3.

![Figure 3. Syntax of VECAS.](image)

In Figure 3, an event/condition is graphically represented by a square associated with a set of attributes. A single square is used for a primitive event/condition while nested squares are used for a composite event/condition. An action is represented as a rounded
rectangle associated with corresponding attributes. The detailed graphical notations for events, conditions and actions can be found in our previous work [9] and are skipped since they are not the emphases for this paper. The $EC$ connection is graphically represented as a line with an arrow. The symbols “I”, “D” and “T” are constraints on the $EC$ connection. They represent three coupling modes [7] between an event and a condition. The $C4$ connection is graphically represented as a dash with an arrow. The symbols “I”, “D”, “T” and “T” are constraints imposed on the $C4$ connection. They represent four coupling modes [7] between a condition and an action. Furthermore, the $RR$ connection is graphically represented as a bold line with an arrow.

3.1.3 Easy-to-use

Easy-to-use is important for a specification language. It includes two aspects: (1) The language should be easy to be understood by users; and (2) The language should be easy to be converted into XML whereas to reduce the complexity of language compiling.

For the first aspect of easy-to-use, we can observe that the syntax of VECAS shown in Figure 3 is very straightforward and reveals the structure of an ECA rule in a nature way. It vividly describes the constituents of an ECA rule (i.e., event, condition and action) by three graphical blocks and the relationships among these three components by lines between corresponding graphical blocks. This straightforward syntax of VECAS allows users to easily understand the structure of an ECA rule without special trainings.

For the second aspect of easy-to-use issue, we can observe that the syntax structure in Figure 3 actually has a tree structure shown in Figure 4. Here, assume rule0 is an ECA rule. In Figure 4, rule0 is the root node. It has three sub-nodes “composite event 2”, “composite condition 0” and “action 0”. These three sub-nodes represent the composite event, composite condition and action that constitute rule0. In this context, a node of composite event or composite condition has several sub-nodes representing its constituent events or conditions. The sub-nodes can be further decomposed into low-level sub-nodes until leaf-nodes representing primitive events or primitive conditions. Since XML is a well-known standard to encode tree-structured information, the VECAS with syntax shown in Figure 3 are easy to be converted into XML.

3.2 SPECIFICATION TOOL IMPLEMENTATION

VSTE consists of three parts, i.e., ECA rule editor, syntax checker and compiler. The user can define specific ECA rules with visual specification language VECAS via the rule editor. When ECA rules are defined, the syntax checker will be activated to check the syntax correctness for these ECA rules. Moreover, the compiler is responsible to convert VECAS representations of defined ECA rules into corresponding XML representations. Figure 5 shows the architecture of VSTE.

![Figure 5. Architecture of visual specification tool VSTE.](image)

### 3.2.1 ECA Rule Editor

VSTE provides a graphical rule editor for users to define ECA rules with VECAS. The Graphical User Interface (GUI) of the rule editor is shown in Figure 6.

![Figure 6. GUI for editing ECA rules.](image)

The GUI includes four areas, i.e., navigator area, outline area, rule edit area, and property define area. In the navigator area, the name of project and the files contained in this project are listed. When the user defines a group of rules, a project containing several files is created. A group of dependent rules are saved into one file under this project. The outline area is responsible to display the overall picture of selected group of ECA rules defined by the user. It has two views: one view is textual view which lists all components in selected ECA rules by texts; another view is graphical view which displays the overall imagine of selected ECA rules. A set of graphical components is provided in the rule edit area. Users can drag corresponding graphical components into the edit area to draw ECA rules according to the syntax of VECAS addressed in section 3.1.2. There are four types of graphical components, i.e., event components, condition components, action components and connection components. The detailed graphical notations for each graphical component can be found in our previous work [9]. In the property defining area, the user can define the attributes associated with each graphical block in defined ECA rules. When the syntax checking is applied to selected ECA rules, the syntax error messages will be also shown in this area.
3.2.2 Syntax Checker
The syntax checker is used to ensure the syntax correctness of ECA rules defined with VECAS. Following criteria are applied to syntax checking:

- The necessary parts should be defined. This means when user defines an ECA rule with VECAS, the event part and action part of this rule should be defined while the condition part is optional depending on specific applications.

- The necessary attributes should be defined. This means that for an event/condition/action, all their associated attributes should be properly defined.

- Number of constituent events/conditions should be right for a composite event/condition. For example, assume the attribute of number of events for an Any event [9] is defined as three. Thus, three sub-squares representing three constituent events should be depicted.

- Number of attributes should be right for an event/condition/action. This means the number of their associated attributes defined by the users should be consistent with the number of attributes in corresponding attribute sets for each type of event/condition/action.

- Value of attributes for event/condition/action should be defined in correct form. For example, when the user defines attribute of constrained time value for a time constrained sequence event[9], the value of this attribute should be an integer larger than 0.

3.2.3 Compiler
Compiler is a key infrastructure of VSTE, which is responsible to convert VECAS representations of ECA rules into corresponding XML-based representations. Via the compiling, a corresponding XML file is created for VECAS codes of a specific set of dependent ECA rules. This XML file has a root element. The name of the root is defined as “VECAS” indicating the file is compiled from VECAS codes. There are two main sections in the file, i.e., Module section and Rule section. The Module section is used to define reused elements in XML descriptions. The Rule section is used to define a set of dependent ECA rules. The framework of this XML file is shown in Figure 7.

![Figure 7. Framework of compiled XML file.](image)

The core of a compiler is a mapping mechanism which is used to create the Rule section in the compiled XML file. It includes two aspects, i.e., component mapping and attribute mapping. In component mapping, the compiler identifies each single ECA rule among a group of dependent graphical ECA rules and creates a sub-element called “ruleitem” for each identified ECA rule in the Rule section. Then, the compiler will search all graphical blocks in each ECA rule and convert graphical blocks representing events, conditions and actions into corresponding event sub-element, condition sub-element and action sub-element of each “ruleitem” element. Figure 8 shows the framework of the Rule section in a compiled XML file created via the component mapping.

![Figure 8. Framework of Rule section.](image)

In attribute mapping, the compiler examines attributes associated with each graphical block in a graphical ECA rule and maps them into the attributes with the same names for corresponding event sub-element, condition sub-element and action sub-element of a “ruleitem” element in the Rule section.

4. RTEIA—AN EVENT-ORIENTED INFERENCE ALGORITHM WITH TIMING CONSTRAINTS

4.1 Timing Constraints
$R$ is a set of predefined ECA rules fed into the inference engine, where $R = \{r_1, r_2, ..., r_n\}$. $A$ is a set of actions defined in rules. $A = \{a_1, a_2, ..., a_n\}$, where action $a_i$ is defined in rule $r_i (1 \leq i \leq n)$ and is executed immediately when $r_i$ is fired. (An ECA rule is fired when its event part is matched by the occurring events and its condition part is evaluated to be true.) We assume that each
rule \( r_i (1 \leq i \leq n) \) has a unique action, and action \( a_i (1 \leq i \leq n) \) executes on dedicated processor in the high-level application.

In a real-time reactive application, for each rule \( r_i (r_i \in R) \), after all events associated with rule \( r_i \) arrive at the system, the action \( a_i (a_i \in A) \) should be completed within a given deadline. Assume \( RT(r_i) \) denotes response time for rule \( r_i \), which is the elapse time from all events associated with rule \( r_i \) are detected to the time action \( a_i \) is completed. \( Id(r_i) \) is the inference delay for rule \( r_i \), which is defined as the elapse time from the time all events associated with \( r_i \) are detected to the time \( r_i \) is fired. Thereby, for \( \forall r_i \in R \), \( Id(r_i) \leq D'_i \), where \( D'_i \) is the deadline of the inference delay for rule \( r_i \). Here, \( D_1 \) is the deadline of the response time for rule \( r_1 \) and \( C(a_i) \) is the execution time of action \( a_i \). They are determined by the specific application requirements.

The problem to be solved by RTEIA is formally described as follows: Given an infinite set of events dynamically arriving at the inference engine, \( E = \{ e_1, e_2, ..., e_m \} \); Given a set of predefined rules, \( R = \{ r_1, r_2, ..., r_n \} \) and a set of actions \( A = \{ a_1, a_2, ..., a_n \} \) defined in rules. For each action \( a_i (1 \leq i \leq n) \), it is immediately executed when \( r_i (1 \leq i \leq n) \) is fired. The goal is to find an inference approach which is able to fire rules in \( R \) so that the corresponding actions in \( A \) can be executed to react to events in \( E \), where for each rule \( r_i (1 \leq i \leq n) \), \( Id(r_i) \leq D'_i \).

### 4.2 Overview of RTEIA

The main idea of RTEIA is to fire rules based on their priorities so that the deadlines of inference delay for rules can be met as much as possible. In this case, each rule is assigned with a priority related to time costs consumed on firing this rule and the deadline of inference delay for this rule. The rule with higher priority will be fired preferentially during the inference.

This idea is materialized by the iterative and heuristic searching on a rule graph. The inference process is converted into a process of finding the path from the entrance nodes to an exit node on the rule graph to fire corresponding rule. The search starts from the corresponding entrance nodes representing primitive events that match with event instances in the event stream. When an exit node is found, the rule associated with this node is fired.

To make the rule with higher priority fired preferentially, a heuristic function is defined for each event node to guide the search on the rule graph. In this case, the heuristic function is well designed to guide the search to preferentially fire the rule which consumes less time to be fired and has earlier deadline of the inference delay. The details of RTEIA will be addressed in the following sections.

### 4.3 Modeling ECA Rules as a Rule Graph

A set of ECA rules fed into the inference engine is modeled as a rule graph. A rule graph, denoted as \( G \), is a directed acyclic graph (DAG): \( G = (V, E) \), where \( V = \{ v_1, v_2, ..., v_n \} \) are nodes and \( E = \{ < v_i, v_j > | v_i, v_j \in V \} \) are directed edges.

- **Node**: There are three types of nodes: event nodes, condition nodes and action nodes on the rule graph. An event node represents either a primitive event or a composite event defined in the ECA rule. Each event node representing a primitive event is an entrance node. Each event node representing a composite event has two child nodes representing two sub-events that constitute this composite event. A condition node represents a primitive condition and is connected to an event node. An action node is connected to a condition node. It is an exit node and represents a rule action of the corresponding ECA rule. Recall that each ECA rule has a unique action. Thereby, an ECA rule is associated with a unique exit node on the rule graph.

- **Edge**: A directed edge is expressed as \( < v_i, v_j > (i, j \geq 1, i \neq j) \), where \( v_i \) will send a token to \( v_j \).

Figure 9 is the rule graph for the rule set described in Table 1.

#### Table 1. An ECA rule set

<table>
<thead>
<tr>
<th>Rule 1</th>
<th>Rule 2</th>
<th>Rule 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>On ( E_1 \land E_2 \lor E_6 )</td>
<td>On(On(( E_1 \land E_2 ))( \land (E_3 \land E_4) ))</td>
<td>On(On(( E_2 \land E_3 ))( \land (E_1 \land E_2) \land E_5 ))</td>
</tr>
<tr>
<td>If ( X &gt; 1 )</td>
<td>If ( Y &lt; 6 )</td>
<td>If ( Z &gt; 10 )</td>
</tr>
<tr>
<td>Do ( A_1 )</td>
<td>Do ( A_2 )</td>
<td>Do ( A_3 )</td>
</tr>
</tbody>
</table>

Figure 9. Example of rule graph.

### 4.4 Heuristic Searching on the Rule Graph

At the beginning, the expected path is null. When the first event instance \( e \) in the event stream arrives at the inference engine, the entrance node representing the primitive event associated with \( e \) is activated and the iterative search starts from it. When a node is activated, a feasibility check is performed to decide if the event/condition represented by this node occurs/is satisfied. If the node passes the feasibility check, it will set its computation token and send the copy of the computation token to its all parent nodes.

At this time, heuristic functions of the parent nodes that receive the computation tokens are computed. The parent node having the smallest value of the heuristic function is selected to be activated and is added into the expected path. The search moves to the next level. If the check fails, the search will go back to the previous level. A node with the next smallest value of the heuristic function in this search level is selected to be activated and added into the expected path. Above process will iterate until an exit node is found, or no more backtracking is possible. After that, the entrance node representing the primitive event that matches with the next arriving event instance in the event stream is activated and all above process will be repeated. Furthermore, during the iterative searching process, as soon as an exit node is activated, the rule associated with this exit node is exported by the inference engine as a reasoning result; this rule is also fired at the same time and the action represented by this exit node is immediately executed.

#### 4.4.1 Heuristic Function

The heuristic function considers two factors that impact satisfaction of deadlines of inference delay for rules during the searching on rule graph: (i) the time consumed on finding an exit node; and (ii) the order of finding exit nodes. The former
represents a conjunction event, is the arriving event, the occurrence of are not null, the is “TRUE” or , denoted as representing a disjunction event, It is described by a tuple: is the composite event is an entrance node representing a primitive event, the representing a sequence event, are not null and represents a condition, (The recent context is used as an event coupling mode during the composite event detection on the node, i.e., only the most recent occurrence of each sub event is taken into account of deciding whether the corresponding composite event occurs and computing the parameters of the corresponding composite event [7]), each time when a node passes the feasibility check, its computation token is updated to record the most recent mediate reasoning results. In this case, if node represents a composite event, the mediate reasoning result generated by is the composite event instance for this composite event derived via the feasibility check. If node is an entrance node representing a primitive event, the mediate reasoning result generated by is the arriving event instance for this primitive event; If node represents a condition, the mediate reasoning result generated by is “TRUE” or “FALSE”.

<table>
<thead>
<tr>
<th>Node</th>
<th>Detection Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>node representing a conjunction event</td>
<td>If both ( CL^v ) and ( CR^v ) are not null, the occurrence of this composite event is detected.</td>
</tr>
<tr>
<td>node representing a disjunction event</td>
<td>If either ( CL^v ) or ( CR^v ) is not null, the occurrence of this composite event is detected.</td>
</tr>
<tr>
<td>node representing a sequence event</td>
<td>If both ( CL^v ) and ( CR^v ) are not null and ( \theta_k ) is an exit node on the rule graph. ( \alpha_k ) is an action represented by ( \theta_k ). ( D_k ) is the deadline of response time for the rule associated with ( \theta_k ). ( C(\alpha_k) ) is the execution time of ( \alpha_k ). Apparently, ( D_k - C(\alpha_k) ) is the deadline of inference delay for the rule associated with ( \theta_k ), ( n ) is the number of exit nodes on the rule graph. ( W ) is the weight used to adjust the importance of two terms in the heuristic function.</td>
</tr>
</tbody>
</table>

\[
H(v) = \text{MIN}_{1 \leq k \leq n} \left( T(v, \theta_k) + W \ast (D_k - C(\alpha_k)) \right)
\]

After that, the node will send the copy of its computation token to its parent nodes. When an event node receives a token from its child node, it will save it as a checking token for this child node to fulfill the future event detections. At this time, the old checking token for this child node will be overwritten by the new one according to recent context.

4.4.3 Detecting Composite Events

When an event node is activated, it should perform a feasibility check to detect if the composite event represented by this node
occurs based on its two checking tokens. RTEIA only consider three types of composite events, i.e., conjunction (∧), disjunction (∨) and sequence (→) [9]. Recall each checking token is the copy of computation token hold by the corresponding child node. The detailed methods for detecting the composite event represented by an event node are listed in Table 2. Here, x is the left child node of node V; y is the right child node of node V; \( CL_x \) is the checking token for x; \( CR_y \) is the checking token for y.

4.5 Algorithm Descriptions

\[
\text{While } ES \neq \emptyset \{
\begin{array}{l}
\quad (1) \text{Set the expected path, denoted as } P, \text{ to be null}
\quad (2) \text{Activate the corresponding entrance node } Ext(E_i), \text{ for } E_i
\quad (3) \text{CurrentNode} = Ext(E_i)
\quad (4) \text{Send the copy of computation token to all parent nodes of } \text{CurrentNode}
\quad (5) \text{Count the number of parent nodes for CurrentNode (This number is denoted as ParentNu(CurrentNode))}
\quad (6) \text{Compute heuristic functions } H \text{ for all parent nodes of CurrentNode}
\quad (7) \text{For each parent node, get the copy of computation token and save it as checking token for the child node corresponding to CurrentNode}
\quad (8) \text{Select one of parent node of CurrentNode with smallest value of } H \text{ for each parent node of CurrentNode}
\quad (9) \text{ParentNu(CurrentNode)} = \text{ParentNu(CurrentNode)} - 1
\quad (10) \text{ActiveNode} = \text{SP}
\quad (11) \text{Activate ActiveNode; check if the event represented by ActiveNode occurs; If it occurs or is satisfied, Occ=true; otherwise, Occ=false}
\quad (12) \text{If (Occ==true)}
\quad \quad (12.1) \text{CurrentNode, ActiveNode > is added into } P
\quad \quad (12.2) \text{CurrentNode = ActiveNode}
\quad \quad \text{Else}
\quad \quad (12.3) \text{If ParentNu(CurrentNode)>0}
\quad \quad \quad \text{Select one of parent node with the next smallest value of } H \text{ for CurrentNode (this node is denoted as SP'); ActiveNode = SP'}
\quad \quad \quad \text{ParentNu(CurrentNode)} = \text{ParentNu(CurrentNode)} - 1
\quad \quad \quad \text{Go to (11)}
\quad \quad \text{Else}
\quad \quad \text{If the backtracking is possible}
\quad \quad \quad \text{Backtrack to the previous level and set CurrentNode to be the selected node in the previous level; Select one of parent node with the next smallest value of } H \text{ for CurrentNode (this node is denoted as SP’); ActiveNode = SP’}
\quad \quad \quad \text{ParentNu(CurrentNode)} = \text{ParentNu(CurrentNode)} - 1
\quad \quad \quad \text{Go to (11)}
\quad \quad \text{Else}
\quad \quad \text{break}
\quad \text{Repeat (4) – (12), until an exit node, i.e., an action node is found; If an exit node is found, the rule associated with this exit node is fired;}
\end{array}
\]

Figure 10. Descriptions of RTEIA.

5. EVALUATIONS

In simulation studies, we are interested in how many times the rules are fired within deadlines of their inference delay. Thus, we use the number of times that rules are fired within the deadlines of their inference delay, denoted as NAFS, as a metric to evaluate the performance of RTEIA. Higher NAFS means higher performance. Specifically, since RTEIA is the core of the inference engine which dominates the performance of RTRS, in the simulation studies, we evaluate NAFS of RTEIA and compare it to Depth-First (DF) algorithm. Here, DF algorithm is same to RTEIA except that depth-first policy is applied to guide the search on the rule graph.

5.1 Simulation Setup

We shall generate ECA rule set by randomly generating a rule graph as follows:

1. Generate M entrance nodes for the rule graph and initiate the alternative node set with these M entrance nodes.
2. Randomly choose two nodes from the alternative node set and randomly take one node as the left child node and another as the right child node. Generate a parent node for these two selected nodes. Here, the type of the composite event represented by this parent node is randomly chosen from ‘conjunction’, ‘disjunction’ and ‘Sequence’.
3. Add the new created parent node into the alternative node set.
4. Repeat (2) – (3) until the number of event nodes on the rule graph reaches ENnu.
5. For each event node which has no parent node on the rule graph, generate a condition node to serve as its parent node.
6. For each condition node on the rule graph, generate an action node to serve as its parent node.

Assume \( \text{Rand}(x, y) \) denotes a random number between \( x \) and \( y \). For each exit node \( \theta \), \( D(\theta) = \text{Rand}(\text{Min}_D, \text{Max}_D) \) and \( C(\theta) = \text{ER} \times \text{Rand}(\text{Min}_E, \text{Max}_E) \), where \( D(\theta) \geq C(\theta) \) is the deadline of response time for the rule associated with \( \theta \); \( C(\theta) \text{ is the execution time of the action represented by } \theta \). To make NAFS of RTEIA easy to be observed, we use execution time ratio (denoted as ER) to adjust the random number created between \( \text{Min}_E \) and \( \text{Max}_E \). When ER increases, the execution time of an action is increased. Here, the values of \( \text{Max}_D, \text{Min}_D, \text{Max}_E \) and \( \text{Min}_E \) are 100, 1, 90, 30 respectively.

We shall generate event instances for each primitive event defined in ECA rules with Poisson arrivals. Each point on each curve in the following figures is the average of 10 experiments, each carried out for a sufficiently long time (2000 event instances arrival). The rule graph and the seed were varied across different experiments, and each experiment was repeated using the same rule graph and seed for RTEIA and Depth-First (DF) algorithm.

5.2 Effect of the Number of Rules

Firstly, we study the relationship between the number of event nodes on the rule graph and the number of rules. From Figure 11, we find that the number of rules is in linear direct proportion to the number of event nodes on the rule graph. More event nodes mean more ECA rules will be included in the generated rule set. Based on this conclusion, we can show the effect of the number of
rules in a rule set fed into the inference engine of RTRS on NAFS of RTEIA by analyzing the relationship between the number of event nodes on the generated rule graph and NAFS of the proposed algorithm. Figure 11 shows the simulation results.

![Figure 11. Relationship between the number of event nodes on rule graph and the number of rules.](image)

Figure 11. Relationship between the number of event nodes on rule graph and the number of rules.

Figure 12 uses logarithmic coordinate to show the effect of the number of rules in a rule set fed into the inference engine of RTRS. It shows that increasing the number of event nodes on the rule graph (i.e., increasing the number of rules in the rule set) results in the decrease of NAFS of RTEIA. We also find RTEIA is not sensitive to the number of rules if the number of event nodes on the rule graph varies within the same order of magnitude. Here, we use equation (5.1) to compute the value of $W$ in the heuristic function.

![Figure 12. Effect of the number of rules.](image)

Figure 12. Effect of the number of rules.

When the number of event nodes on the rule graph is smaller than 1000, NAFS of RTEIA is always higher than that of DF algorithm. However, the difference between NAFS of RTEIA and DF algorithm is more and more inconspicuous with the increase of the number of event nodes on the rule graph. This result demonstrates that the heuristic searching policy used by RTEIA is effective to improve the number of times that rules are fired within the deadlines of their inference delay especially when the number of event nodes on the rule graph is less than 1000 (i.e., the number of rules is less than 333).

5.3 Effect of Action’s Execution Time

Figure 13, Figure 14 and Figure 15 show effect of action’s execution time on NAFS of RTEIA under 50 event nodes, 200 event nodes and 500 event nodes on the rule graph. Here, logarithmic coordinate is used to show the results. We also use equation (5.1) to compute the value of $W$ in the heuristic function.

From Figure 13, Figure 14 and Figure 15, we observe that before a certain point of execution time ratio, NAFS of RTEIA is very stable. This means when execution time of the action is small enough, the performance of RTEIA is not sensitive to the execution time of the action. However, after this point, NAFS of RTEIA is dramatically reduced with the increase of execution time ratio. This means when the execution time of the action is large to certain extent, RTEIA will be sensitive to the execution time of the action and increasing execution time of the action will greatly decrease NAFS of RTEIA.

![Figure 13. Effect of execution time of action (The Number of Event nodes on rule graph is 50).](image)

Figure 13. Effect of execution time of action (The Number of Event nodes on rule graph is 50).

![Figure 14. Effect of execution time of action (The Number of Event nodes on rule graph is 200).](image)

Figure 14. Effect of execution time of action (The Number of Event nodes on rule graph is 200).

![Figure 15. Effect of execution time of action (The number of event nodes on rule graph is 500).](image)

Figure 15. Effect of execution time of action (The number of event nodes on rule graph is 500).

Meanwhile, the turning point of execution time ratio differs under different number of event nodes on the rule graph. In this case, more event nodes, i.e., more rules in the rule set results in larger value of turning point. Specifically, the turning points of execution time ratio under 50 event nodes, 200 event nodes and 500 event nodes are 0.05, 0.06 and 0.07 respectively. Furthermore, for various execution time ratios, i.e., various execution times of the actions, NAFS of RTEIA is always higher than that of DF algorithm. However, this advantage disappears when execution time ratio is larger than 0.1.

5.4 Estimating the Value of Weight

The reasonable value of $W$ will benefit the performance of RTEIA. We denote the first term in heuristic function $H$ as $F_1$ and the second term as $F_2$. The criteria for setting the value of $W$ is to make these two terms $F_1, F_2$ be on the same order of magnitude so that the corresponding two factors, i.e., the estimated time...
consumed on firing a rule and the order of firing rules can be sufficiently considered during the heuristic search on the rule graph.

To estimate the value of \( W \), we study the relationship between the number of event nodes on the rule graph and the length of the longest path on the rule graph. Figure 16 shows that the length of the longest path on the rule graph is in linear direct proportion to the natural logarithm of the number of event nodes on the rule graph, i.e., \( LP = 4.5701^{\ln(ENnu)} - 11.546 \), where \( LP \) represents the length of longest path on the rule graph and \( ENnu \) is the number of event nodes on the rule graph. Thus, without loss of the generality, considering the value of \( F_i \) depends on the length of longest path on the rule graph and \( WT \) function value is between 0 and 2, we use following equation to estimate the value of \( W \):

\[
W = \frac{(4.5701^{\ln(ENnu)} - 11.546)^*N}{\sum_{i=1}^{n}(D(\theta_i)-C(\theta_i))}
\]

(5.1)

![Figure 16. Relationship between the number of event nodes on rule graph and length of the longest path on the rule graph.](image)

Here, \( n \) is the number of exit nodes on the rule graph, \( \theta_i \) is an exit node (1 \( \leq i \leq n \)). The value of \( n \) can be known when the rule set fed into the inference engine of RTEIA is determined. \( D(\theta_i) \) is the deadline of response time for the rule associated with \( \theta_i \); \( C(\theta_i) \) is the execution time of the action represented by \( \theta_i \). These two parameters are determined by the specific application requirements.

### 6. CONCLUSIONS AND FUTURE WORKS

This paper proposes a real-time reasoning system based on active rules (i.e., event-condition-action rules), called RTEIA, to make automatic decisions about how to react to continuously arriving events (i.e., event streams) so that the deadlines of inference delay for rules can be met as much as possible. A series of simulations are conducted to evaluate the performance of RTEIA.

However, RTEIA – the core of the inference engine of RTRS, is a “best effort” algorithm that only tries its best to satisfy the deadlines of inference delay for rules but cannot provide the guarantee. It can be only used for the reactive applications with soft real-time constraints where missing deadlines of inference delay for rules will not induce serious consequences. An on-line checking mechanism should be developed to integrate with RTEIA so as to ensure the satisfaction of deadlines of inference delay for rules. We will also extend the types of composite events considered by RTEIA. These composite events include time constrained sequence event, between event, any event etc [9]. Furthermore, we will enhance the expressiveness of VSTE by improving graphical representations of VECAS and allowing VECAS to describe the fuzzy temporal information.

### 7. ACKNOWLEDGMENTS

This work is supported by National Science Foundation of China under Grant No. 61232013, Institute of Software, Chinese Academy of Sciences (CAS) under Grant No. ISCAS2009-JQ03 and State Grid Corporation of China under Grant No. JSKF-2011-85.

### 8. REFERENCES


ABOUT THE AUTHORS:

Ying Qiao received her PhD degree in computer science from the Institute of Software (IOS), Chinese Academy of Sciences, in 2001. She is currently an associate professor at IOS. She worked as research associate at US National Research Council from 2002 to 2005. Her research interests include real-time scheduling, real-time intelligence and software engineering. She is the author (co-author) of more than 30 papers.

Qinan Han received his Bachelor degree from the University of Zhejiang, China in 2010. He is currently a graduate student at the Institute of Software, Chinese Academy of Sciences. His research interests include real-time system and multi-agent system.

Hongan Wang received his PhD degree in computer science from the Institute of Software (IOS), Chinese Academy of Sciences, in 1999. He is currently a professor and the director of Intelligence Engineering Lab at IOS. His research interests include real-time intelligence, real-time database and real-time scheduling. He is the author (co-author) of more than 100 papers. He had two patents and obtained National Science and Technology Progress Award, CAS (Chinese Academy of Sciences) Science and Technology Progress Award.

Xiang Li received his PhD degree in computer science from the Institute of Software (IOS), Chinese Academy of Sciences, in 2011. He is currently a post-doc in Tsinghua University, China. His research interests include real-time reasoning and service computing.
Kang Zhong received his PhD degree from the Institute of Software, Chinese Academy of Sciences in 2011. He spent one and a half years as an international researcher at the Brunel University, UK. He is currently an assistant professor at the Institute of Software, Chinese Academy of Sciences. His research interests include real-time system and real-time intelligence.

Keming Zhang received his Master degree from the University of Sheffield, UK in 2009. He is currently an engineer at the Institute of Software, Chinese Academy of Sciences. His research interests include real-time system and real-time reasoning.

Jian Liu received his Ph.D degree from Xi’an University of Technology in Electrical Engineering in 1997. He is currently a professor and the chief engineer of Shanxi Electric Power Research Institute. He has published 12 books and more than 100 papers. His research interests include power system analysis, control and planning.

Anxiang Guo received his master degree from Northeast Dianli University. He is currently a senior engineer and the director of technology department at Shanxi electric power Research Institute. His research interests include water treatment and electric power environmental protection.