FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER’S THESIS

Title:	Implementation of a generalized version of a system for discriminant chronicles mining
Student:	Bc. Radek Buša
Supervisor:	prof. Ing. RNDr. Martin Holeňa, CSc.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2020/21

Instructions

1. Get thoroughly acquainted with discriminant chronicles mining as described in the paper Dauxais et al., 2017, provided by the supervisor.
2. Get familiar with the system for discriminant chronicles mining implemented by the authors of the provided paper.
3. Analyze which changes in the system will be needed to generalize discriminant chronicles mining from integer scalar inputs to real-valued vector inputs.
4. Design an extension of the original system for discriminant chronicles mining, incorporating the proposed generalization.
5. Implement the designed system.
6. Propose and perform suitable testing methods for the implemented system.
7. Apply your implementation to real-valued vector input data provided by the supervisor.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 21, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Implementation of a generalized version of a system for discriminant chronicles mining

Bc. Radek Buša

Department of Software Engineering
Supervisor: prof. RNDr. Ing. Martin Holeňa, CSc.

April 29, 2020

Acknowledgements

In the first place, I would like to thank my supervisor prof. RNDr. Ing. Martin Holeňa, CSc. who proved to be a patient and erudite mentor.

Next, I would like to thank my family — mother Jana, father Petr, grandmother Jana, grandfather Frantisek, great-grandmother Jaroslava and great-grandfather Josef — for their unconditional love, endless support and a deep inspiration for all those 24 years of my life.

Last but not least, sincere thanks to all policemen, firefighters, military men, medical staff, scientists, politicians and everyone else who helped to save our country from massive COVID-19 pandemic outbreak in Spring 2020.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on April 29, 2020 . . . .

Citation of this thesis

Buša, Radek. *Implementation of a generalized version of a system for discriminant chronicles mining.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Tato práce se věnuje úpravě existujícího systému pro vytěžování odlišných průběhů událostí z dat (discriminant chronicles mining) tak, aby byl schopen přijímat vícedimenzionální vstupní data. Upravený systém bude následně použit na reálná data, která se týkají růstu monokrystalů.

Klíčová slova vytěžování znalostí z dat, vytěžování vzorů z dat, vytěžování odlišných průběhů událostí z dat, vytěžování pravidel z dat, automatizované testování software, refaktorování, Python, C++

Abstract

This thesis is dedicated to modifying an existing system for discriminant chronicles mining, resulting in a system for discriminant chronicles mining capable of handling multi-dimensional input data. The modified system will be applied to real-world data concerning crystal growth.

Keywords data mining, pattern mining, discriminant chronicles mining, rules mining, automated software testing, refactoring, Python, C++

Contents

List of Figures

List of Tables

Listings

Introduction

State-of-the-art

Discriminant chronicles mining is a field of data mining which helps to gain insight into temporal data by mining temporal patterns called *chronicles* which are considered significant in the context of the input dataset.

Currently implemented solutions for performing such type of data mining task are, however, limited in terms of accepted input data.

Motivation

I chose this topic because mining discriminant chronicles is at the time of writing this thesis in its outset and extending the possibilities of the existing system for discriminant chronicles mining might help this field of pattern mining become more prominent among other methods for pattern mining.

Thesis Goals and Result

The goals of this thesis are stated in the following paragraphs:

To introduce the reader to the theory of discriminant chronicles, discriminant chronicles mining and algorithms for discriminant chronicles mining.

To review a system implementing an algorithm for discriminant chronicles mining and extend it in order to accept multi-dimensional input data and mine discriminant chronicles containing multi-dimensional constraints.

To propose a suitable testing methodology and perform testing following this methodology for ensuring the system extension works properly.

To apply the modified system to multi-dimensional input data concerning crystal growth and interpret the resulting discriminant chronicles containing multi-dimensional constraints.

Chapter 1

Problem Faced

This chapter will introduce the reader to the problem of discriminant temporal pattern mining.

Section 1.1 will present the field of temporal pattern mining – its characteristics, practical usage and basic temporal pattern metrics.

Section 1.2 will introduce theoretical foundations important for grasping algorithmic concepts of an algorithm introduced in the subsequent section.

Section 1.3 will introduce the aforementioned algorithm for mining discriminant patterns proposed in [1] and describes its conceptual basis.

1.1 State-of-the-art

Temporal pattern mining is a research field of data science that studies algorithms used for extracting interesting patterns from temporal data and finds its usage in various fields of human research – medical data [2], sign language research [3], human behavior schemata in computer systems [3] and many more. [1]

Temporal pattern mining methods are discriminated by the types of patterns they extract:

- *Sequential patterns* are patterns that take order of the events into account,
- *Temporal rules* and *Chronicles* are patterns that are based on inter-event durations,
- *Time interval patterns* are patterns that bear inter-event durations and timestamps. [1]

According to [1], not all temporal patterns obtained using temporal pattern mining methods might be suited well for solving the problems they are supposed to solve. Therefore, *temporal pattern discriminancy* is an important qualitative property of temporal patterns.

Assume a dataset consisting of temporal data split into two disjoint parts. *Discriminant temporal patterns* are temporal patterns that occur frequently for the first part of temporal data (usually called *positive temporal data*) but are not frequently present in the second part of temporal data (usually called *negative temporal data*) [1].

1.2 Theoretical Foundations

Yann Dauxais et *al.* in their paper *Discriminant chronicles mining: Application to care pathways analytics* [1] thoroughly sum up theoretical introduction to the topic:

Assume that $\mathbb{E}$ is a set of event types totally ordered by $<_{\mathbb{E}}$, $\mathbb{T} \subseteq \overline{\mathbb{R}}$ is a temporal domain and $\mathbb{L} = \{+, -\}$ is a label set.

An *event* is a couple (e, t) where $e \in \mathbb{E}$ and $t \in \mathbb{T}$.

A *sequence* is a tuple $\langle SID, \langle (e_1, t_1), (e_2, t_2), \ldots, (e_n, t_n) \rangle, L \rangle$ where:

- SID is a sequence index,
- $L \in \mathbb{L}$ is a sequence label,
- $\langle (e_1, t_1), (e_2, t_2), \ldots, (e_n, t_n) \rangle$ is a finite sequence of events ordered by $\prec$ defined as $\forall i, j \in \hat{n} : (e_i, t_i) \prec (e_j, t_j) \iff t_i < t_j \vee (t_i = t_j \wedge e_i <_{\mathbb{E}} e_j)$ where $<$ denotes a traditional *less-than* relation between two elements of $\overline{\mathbb{R}}$.

A *temporal constraint* is a tuple $\langle e_1, e_2, t^-, t^+ \rangle$, also denoted as $e_1[t^-, t^+]e_2$ where:

- $e_1, e_2 \in \mathbb{E}$ and $t^-, t^+ \in \mathbb{T}$ and $t^- \leq t^+$ where $\leq$ denotes a traditional *less-than or equal to* relation between two elements of $\overline{\mathbb{R}}$.

A temporal constraint $e_1[t^-, t^+]e_2$ is said *satisfied* by a couple of events $((e, t), (e', t'))$ if and only if $e = e_1 \wedge e' = e_2 \wedge t' - t \in \langle t^-, t^+ \rangle$[1].

A *chronicle* is a couple $(\mathcal{E}, \mathcal{T})$ where:

- $\mathcal{E} = \{\!\{ e_1, e_2, \ldots, e_n \}\!\}$[2], $e_i \in \mathbb{E}$, $i \in \hat{n}$ is an ordered multiset of event types where $\forall i, j : 1 \leq i < j \leq n$, $e_i \leq_{\mathbb{E}} e_j$,
- $\mathcal{T} = \{ e_1[t^-, t^+]e_2 | e_1, e_2 \in \mathcal{E}, e_1 \leq_{\mathbb{E}} e_2 \}$ is a temporal constraint set.

[1]$\langle a, b \rangle$ represents a closed interval from a to b.

[2]$\{\!\{ e_1, e_2, \ldots, e_n \}\!\}$ represents a multiset.

1.2.1 Chronicle Occurrences

Assume that s is a sequence of n events and $\mathcal{C} = (\mathcal{E} = \{\!\{e'_1, e'_2, \ldots, e'_m\}\!\}, \mathcal{T})$, $m \in \mathbb{N}$ is a chronicle.

An *occurrence* of chronicle $\mathcal{C}$ in sequence $s = \langle (e_1, t_1), (e_2, t_2), \ldots, (e_n, t_n) \rangle$ is a subsequence $\tilde{s} = \langle (e_{f(1)}, t_{f(1)}), (e_{f(2)}, t_{f(2)}), \ldots, (e_{f(m)}, t_{f(m)}) \rangle$, such that:

- $f : \hat{m} \longmapsto \hat{n}$ is an injective function,
- $\forall i : e'_i = e_{f(i)}$,
- $\forall i, j : t_{f(j)} - t_{f(i)} \in \langle a, b \rangle$ where $e'_i[a, b]e'_j \in \mathcal{T}$.

Chronicle $\mathcal{C}$ *occurs* in sequence s, $\mathcal{C} \in s$ if and only if there exists at least one occurrence of $\mathcal{C}$ in s.

The *support* of a chronicle $\mathcal{C}$ in a sequence set $\mathcal{S}$, defined as $\operatorname{supp}(\mathcal{C}, \mathcal{S}) = |\{s \in \mathcal{S} | \mathcal{C} \in s\}|$, is the number of sequences of $\mathcal{S}$ in which $\mathcal{C}$ occurs.

Given *minimal support threshold* denoted as $\sigma_{min} \in \mathbb{N}$[1], chronicle $\mathcal{C}$ is *frequent* if and only if $\operatorname{supp}(\mathcal{C}, \mathcal{S}) \geq \sigma_{min}$.

Let $\mathcal{S}$ be a sequence set, $l \in \mathbb{L}$ a label and SID an arbitrary sequence index. *Sequence set label* $\mathcal{L}(\mathcal{S})$ is defined for $|\mathcal{S}| > 0$ as follows: $\mathcal{L}(\mathcal{S}) = l \iff \forall s \in \mathcal{S}, e_i \in \mathbb{E}, t_i \in \mathbb{T}, i \in \hat{n} : s = \langle SID, \langle (e_1, t_1), (e_2, t_2), \ldots, (e_n, t_n) \rangle, l \rangle$.

1.2.2 Discriminant Chronicles

Assume that $\mathcal{S}^-$ and $\mathcal{S}^+$ are two sequence sets fulfilling $\mathcal{S}^+ \cup \mathcal{S}^- = \mathcal{S} \wedge \mathcal{S}^+ \cap \mathcal{S}^- = \emptyset$, $\sigma_{min} \in \mathbb{N}$ minimal support threshold and $g_{min} \in \langle 1, \infty)$ *minimal growth threshold.*

Chronicle $\mathcal{C}$ is *discriminant* for sequence set $\mathcal{S}^+$ with respect to sequence set $\mathcal{S}^-$ if and only if $\operatorname{supp}(\mathcal{C}, \mathcal{S}^+) \geq \sigma_{min} \wedge \operatorname{supp}(\mathcal{C}, \mathcal{S}^+) \geq g_{min} \cdot \operatorname{supp}(\mathcal{C}, \mathcal{S}^-)$.

Chronicle $\mathcal{C}$ is *discriminant* for sequence set $\mathcal{S}^-$ with respect to sequence set $\mathcal{S}^+$ if and only if $\operatorname{supp}(\mathcal{C}, \mathcal{S}^-) \geq \sigma_{min} \wedge \operatorname{supp}(\mathcal{C}, \mathcal{S}^-) \geq g_{min} \cdot \operatorname{supp}(\mathcal{C}, \mathcal{S}^+)$.

Growth rate denoted as $g(\mathcal{C}, \mathcal{S})$ is defined as follows:

$$g(\mathcal{C}, \mathcal{S}) = \begin{cases} \frac{\operatorname{supp}(\mathcal{C}, \mathcal{S}^+)}{\operatorname{supp}(\mathcal{C}, \mathcal{S}^-)} & \text{for } \operatorname{supp}(\mathcal{C}, \mathcal{S}^-) > 0 \\ +\infty & \text{otherwise.} \end{cases}$$

[1] $\mathbb{N} = \{1, 2, 3, \ldots\}$.

1.3 Discriminant Chronicles Mining

DCM (Discriminant Chronicles Mining) is a data-mining task yielding a set of chronicles discriminant for $\mathcal{S}^+$ with respect to $\mathcal{S}^-$ with the following inputs [1]:

- $\mathcal{S}^-$, $\mathcal{S}^+$ sequence sets where $\mathcal{L}(\mathcal{S}^-) = -$ and $\mathcal{L}(\mathcal{S}^+) = +$ respectively,
- σ_{min} minimal support threshold,
- g_{min} minimal growth threshold.

The σ_{min} minimal support threshold parameter serves for pruning unfrequent chronicles which are considered insignificant – discriminant chronicles with the same event multiset and similar temporal constraints are, according to [1, p. 5], considered redundant and thus insignificant for the result.

1.3.1 DCM Algorithm

The DCM algorithm is an algorithm for discriminant chronicles mining proposed by Yann Dauxais et *al.* in [1, p. 5], mining an incomplete set of discriminant chronicles considered by [1] as meaningful for $\mathcal{S}^+ \subset \mathcal{S}$, determined by user-supplied argument values σ_{min} and g_{min}. The pseudocode of the algorithm proposed in [1] is portrayed in Listing 1.1.

```
DCM(pos, neg, fmin, gmin) {
  M := extractMultiSet(pos, fmin).
  C := emptySet().

  for (m of M) {
    if (supp(pos, {m,tinf}) > (gmin * supp(neg, {m,tinf}))) {
      C.add({m,tinf}). // adds a discriminant chronicle
                       // without temporal constraints
    }
    else {
      for (t of extractDTC(pos, neg, m, fmin, gmin)) {
        C.add({m,t}). // adds a discriminant chronicle
                      // with temporal constraints
      }
    }
  }

  return C.
}
```

Listing 1.1: DCM pseudocode

The meaning of symbols in the DCM pseudocode depicted in Listing 1.1 is as follows:

The `pos` parameter represents $\mathcal{S}^+$ – sequence set containing sequences with positive label (i.e. where $\mathcal{L}(\mathcal{S}^+) = +$),

The `neg` parameter represents $\mathcal{S}^-$ – sequence set containing sequences with negative label (i.e. where $\mathcal{L}(\mathcal{S}^-) = -$),

The `fmin` parameter represents σ_{min} – minimal support threshold,

The `gmin` parameter represents g_{min} – minimal growth threshold,

The `tinf` constant represents $\mathcal{T}_\infty$, i.e. a set of temporal constraints with all bounds equal to ∞,

The `extractMultiSet(...)` function extracts a set of frequent multisets of event types for $\mathcal{S}^+$, given σ_{min}, further explained in Section 1.3.1.1,

The `extractDTC(...)` function elaborated in Section 1.3.1.2 extracts discriminant temporal constraints from given frequent multiset and sequence sets, given σ_{min} and g_{min},

The `M` data structure represents a set of frequent multisets,

The `C` data structure represents a set of discriminant chronicles.

The branching statement in Listing 1.1 containing the compound condition `supp(pos, {m,tinf}) > (gmin * supp(neg, {m,tinf}))` is used to check whether given frequent multiset without further specific temporal constraints is discriminant (a chronicle $(\mathcal{E}, \mathcal{T}_\infty)$ is equivalent to $\mathcal{E}$ in terms of temporal relations). If the given condition is true, no discriminant temporal constraints are mined using the `extractDTC(...)` function because it would only yield specialized cases of $(\mathcal{E}, \mathcal{T}_\infty)$ which are considered redundant [1].

1.3.1.1 Frequent Multiset Extraction

The `extractMultiSet(...)` function extracts a set of frequent multisets from a given sequence set and user-supplied minimal support threshold (σ_{min}).

It applies a regular *frequent itemset mining algorithm* where an event type $a \in \mathbb{E}$ occurring n times in a sequence is encoded by n items $I_1^a, I_2^a, \ldots, I_n^a$. An intermediate *frequent itemset* of size m denoted as $(I_{i_k}^{e_k})_{1 \leq k \leq m}$ is extracted from the supplied sequence set and is further transformed into the resulting multiset. [1]

The last phase of the algorithm incorporates converting each frequent itemset $(I_{i_k}^{e_k})_{1 \leq k \leq m}$ to a multiset containing an event e_k exactly i_k times, $1 \leq k \leq m$, while ignoring the itemsets containing two items $I_{i_k}^{e_k}$ and $I_{i_l}^{e_l}$ where $e_k = e_l$ and $i_k \neq i_l$ – such itemsets are considered redundant. [1]

1.3.1.2 Discriminant Temporal Constraints Mining

The `extractDTC(...)` function is used to mine discriminant temporal constraints from given frequent multiset $\mathcal{E} = \{\!\{a_1, a_2, \ldots, a_n\}\!\}$, sequence sets $\mathcal{S}^+$ and $\mathcal{S}^-$, $\mathcal{S} = \mathcal{S}^+ \cup \mathcal{S}^-$, $\mathcal{S}^+ \cap \mathcal{S}^- = \emptyset$ and with user-defined parameters σ_{min} and g_{min}.

Assume that e_i and e_j are events. $A_{e_i \to e_j} \in \mathbb{T}$ represents an inter-event duration between e_i and e_j.

Let $\mathcal{E} \in \mathbb{M}$ be a frequent event multiset of size m, $\mathcal{D}$ a relational dataset, EID an arbitrary example index, SID an arbitrary sequence index and $l \in \mathbb{L}$ a sequence label. For each occurrence of $\mathcal{E}$ in $\mathcal{S}$, a row also called an *example* of form $\langle EID, SID, (\mathcal{A}_{e_i \to e_j})_{1 \le i,j \le m, e_i <_{\mathbb{E}} e_j}, l \rangle$ is added into the relational dataset $\mathcal{D}$ [1]. An example of a relational dataset is illustrated in Table 1.1. Note that the EID values are unique within the relational dataset, whereas the SID values do not need to be unique.

EID	SID	$\mathcal{A}_{A \to B}$	$\mathcal{A}_{B \to C}$	$\mathcal{A}_{A \to C}$	Label
1	1	1	-4	6	$+$
2	1	9	5	3	$+$
3	2	1	4	3	$+$
4	3	-3	8	0	$-$

Table 1.1: Relational dataset example

After populating the relational dataset $\mathcal{D}$, a *numerical rule learning algorithm* will induce numerical rules while taking minimal growth threshold g_{min} into account [1].

Each such induced rule is an implication, containing conjunctions of form $\mathcal{A}_{e_i \to e_j} \ge x \wedge \mathcal{A}_{e_i \to e_j} \le y$ where $x, y \in \mathbb{T}$ and $e_i, e_j \in \mathcal{E}$ in its premise and a label $l \in \mathbb{L}$ in its conclusion [1]. The x and y temporal values are called *rule values* and the symbols $\mathcal{A}_{e_i \to e_j}$ representing durations between specified events in a rule are called *rule significands.*

Such rules of form $\mathcal{A}_{e_i \to e_j} \ge x \wedge \mathcal{A}_{e_i \to e_j} \le y$ are transformed into temporal constraints of form $\langle e_i, e_j, x, y \rangle$, i.e. $e_i[x, y]e_j$, which are added into the temporal constraint set of the resulting discriminant chronicle [1].

1.3.2 DCM Algorithm Implementation

Based on the *DCM* algorithm proposed in Section 1.3.1, the authors of [1] implemented a proof-of-concept tool for mining discriminant chronicles. It utilizes the RIPPER*k* algorithm proposed in [4] for inducing numerical rules as explained in Section 1.3.1.2. The original source code of the *DCM* implementation is available at [5]. The implementation will further be discussed in Chapter 3. Note that the system also supports frequent chronicles mining and discriminant episodes mining which are out of assignment scope of this thesis and will not be elaborated further.

Along with the implementation of the *DCM* algorithm, a tool used for generating classification rules named *DC-PBC (Discriminant Chronicles Pattern-Based Classification)* was implemented by the authors of the aforementioned *DCM* algorithm. The source codes of both of these projects are available at [6].

1.4 Conclusion

This chapter presented an introduction to the field of temporal pattern mining along with its theoretical basis and also introduced the reader to the *DCM* algorithm proposed by Yann Dauxais et *al.* in [1] along with its important concepts of operation.

Chapter 2

Towards the Generalization of Discriminant Chronicles Mining

This chapter will introduce new theoretical foundations building on the original theoretical foundations for discriminant chronicles mining elaborated in Section 1.2, for generalizing the system for multi-dimensional input data.

Section 2.1 will state the reasons why the generalization of the existing system for discriminant chronicles mining is desired.

Section 2.2 will introduce new generalized definitions to the theoretical basis as introduced in Section 1.2.

2.1 Motivation

As described in the previous chapter, the original system along with its theoretical foundations is able to mine discriminant chronicles containing intervals of extended real numbers.

The aforementioned scalar domain might be constraining in areas where the input data are multi-dimensional, for instance.

A real-world example of such situation is mining classification rules based on temperatures recorded by multiple sensors during a crystal growth process in a furnace. In that case, the temperatures recorded at a particular time can be represented by a vector of real numbers [7].

2.2 Conceptual Changes

In contrast to the theoretical foundations introduced in Chapter 1, the theoretical concepts described in the following sections will be changed.

2.2.1 Event Types

Event type set $\mathbb{E}$ will not be totally ordered in contrast to the original definition as introduced in Section 1.2.

2.2.2 Domain

The temporal domain $\mathbb{T}$ as introduced in Section 1.2 will lose its temporal semantics and will be extended to vectors of extended real numbers of arbitrary length, i.e. $\mathbb{T} \subseteq \overline{\mathbb{R}}^d$.

2.2.3 Sequence Event Order

The domain $\mathbb{T} \subseteq \overline{\mathbb{R}}^d$ can't be totally ordered, so the original relation $\prec$ imposing total order to events as sequence elements will be undefined.

2.3 New Concepts

2.3.1 Hyperrectangle Test

Assume $\vec{a} = (a_1, a_2, \ldots, a_d), \vec{b} = (b_1, b_2, \ldots, b_d) \in \overline{\mathbb{R}}^d$ and $n \in \mathbb{N}$. *Hyperrectangle* $\mathfrak{R}(\vec{a}, \vec{b})$ is a generalization of rectangle in d-dimensional space, represented by a cartesian product of d orthogonal closed intervals $\langle a_1, b_1\rangle \times \langle a_2, b_2\rangle \times \ldots \times \langle a_d, b_d\rangle$.

Let $\vec{a}, \vec{b}, \vec{c} \in \overline{\mathbb{R}}^d$. *Hyperrectangle test*, denoted by $\Subset$ is defined as follows:

$$\vec{c} \Subset \mathfrak{R}(\vec{a}, \vec{b}) \iff \forall i \in \hat{d} : c_i \in \langle a_i, b_i\rangle.$$

Geometrical interpretation of hyperrectangle test means that if $\vec{c} \Subset \mathfrak{R}(\vec{a}, \vec{b})$ holds, the point $\vec{c} \in \overline{\mathbb{R}}^d$ is located inside a d-dimensional body of hyperrectangle $\mathfrak{R}(\vec{a}, \vec{b})$ where $\vec{a}, \vec{b} \in \overline{\mathbb{R}}^d$.

2.3.2 Hyperrectangle Constraints

A *hyperrectangle constraint* is a tuple $\langle e_1, e_2, \vec{t_1}, \vec{t_2}\rangle$, also denoted as $e_1[\![\vec{t_1}, \vec{t_2}]\!]e_2$ where $e_1, e_2 \in \mathbb{E}$ and $\vec{t_1}, \vec{t_2} \in \mathbb{T}$.

A hyperrectangle constraint $e_1[\![\vec{t_1}, \vec{t_2}]\!]e_2$ is said *satisfied* by a couple of events $((e, \vec{t}), (e', \vec{t'}))$ if and only if $e = e_1 \wedge e' = e_2 \wedge \vec{t'} - \vec{t} \Subset \mathfrak{R}(\vec{t_1}, \vec{t_2})$.

2.3.3 Multi-dimensional Chronicles

A *multi-dimensional chronicle* is a couple $(\mathcal{E}, \mathcal{T})$ where:

- $\mathcal{E} = \{\!\{e_1, e_2, \ldots, e_n\}\!\}$, $e_i \in \mathbb{E}$, $i \in \hat{n}$ is a multiset of event types,
- $\mathcal{T} = \{e_1[\![\vec{t_1}, \vec{t_2}]\!]e_2 | e_1, e_2 \in \mathcal{E}\}$ is a hyperrectangle constraint set.

2.3.4 Multi-dimensional Chronicle Occurrences

Assume that s is a sequence of n events and $\mathcal{C} = (\mathcal{E} = \{\!\{e'_1, e'_2, \ldots, e'_m\}\!\}, \mathcal{T})$, $m \in \mathbb{N}$ is a chronicle.

An *occurrence* of chronicle $\mathcal{C}$ in sequence $s = \langle (e_1, \vec{t_1}), (e_2, \vec{t_2}), \ldots, (e_n, \vec{t_n}) \rangle$ is a subsequence $\tilde{s} = \langle (e_{f(1)}, \vec{t}_{f(1)}), (e_{f(2)}, \vec{t}_{f(2)}), \ldots, (e_{f(m)}, \vec{t}_{f(m)}) \rangle$, such that:

- $f : \hat{m} \longmapsto \hat{n}$ is an injective function,
- $\forall i : e'_i = e_{f(i)}$,
- $\forall i, j : \vec{t}_{f(j)} - \vec{t}_{f(i)} \in \mathfrak{R}(\vec{a}, \vec{b})$ where $e'_i[\![\vec{a}, \vec{b}]\!]e'_j \in \mathcal{T}$.

2.3.5 Discriminant Constraint Mining

Let $\mathbb{M}$ be a set of frequent event multisets, $\mathcal{E} \in \mathbb{M}$ be a frequent event multiset of size m, $\mathcal{D}$ a relational dataset, EID an arbitrary example index, SID an arbitrary sequence index and $l \in \mathbb{L}$ a sequence label. Analogically as for the non-generalized definition, for each occurrence of $\mathcal{E}$ in $\mathcal{S}$, a row also called an *example* of form $\langle EID, SID, (\mathcal{A}_{e_i \to e_j})_{1 \le i,j \le m}, l \rangle$ is added into the relational dataset $\mathcal{D}$. An example of such relational dataset is illustrated in Table 2.1.

EID	SID	$\mathcal{A}_{A\to B}$	$\mathcal{A}_{B\to C}$	$\mathcal{A}_{A\to C}$	Label
1	1	$(1, 2.4)$	$(-4.6, 5.1)$	$(6.3, 0)$	$+$
2	1	$(0, 0)$	$(9.6, -7.4)$	$(0, 1.9)$	$+$
3	2	$(1.6, -2)$	$(4, 7.8)$	$(3.9, -1.5)$	$+$
4	3	$(-3.6, 0.9)$	$(8.4, -4.9)$	$(0, 1.5)$	$-$

Table 2.1: Generalized relational dataset example

2.4 Conclusion

This chapter presented the motivation why the generalization of discriminant chronicles mining is desired and also introduced the new theoretical basis needed for implementing a generalized version of the system for discriminant chronicles mining.

Chapter 3

Current Implementation State

This chapter will discuss the current state of the system.

Section 3.1 will introduce the technological aspects of the system.

Section 3.2 will state current features of the system and implementation-imposed limitations of the system.

Section 3.3 will explain the high-level modular structure of the system and communication schemata between system components.

Sections 3.4 and 3.5 will elaborate the building blocks of each individual component of the system – packages, modules, classes, methods and attributes.

Section 3.6 will state technical debt in the aforementioned projects in terms of code issues.

3.1 Technologies Used

3.1.1 Python

Python is an interpreted object-oriented programming language with dynamic typing known for its simple, concise and compact syntax. Python code is also portable between supported platforms in the sense that it is not necessary to recompile for the target platform. [8, p. 3]

3.1.2 C++

C++ is a high-level general-purpose compiled programming language backwards compatible with C programming language. As opposed to C, it provides features for writing object-oriented code which when written well, makes the source code of the programs shorter, easier to understand and easier to maintain. [9, p. 13]

3.1.3 scikit-learn

scikit-learn (sklearn) is a set of tools for predictive data analysis usable in Python environments, including implementation of classifiers, regressors, clustering algorithms, data preprocessing algorithms, etc. [10]

3.1.4 WEKA

WEKA (Waikato Environment for Knowledge Analysis) is a collection of ML (Machine Learning) algorithms and data preprocessing tools implemented in Java with companion GUI (Graphical User Interface) used for experimental data mining – i.e. comparing different ML methods on given input data. This toolkit also features a CLI (Command Line Interface) for batch ML task processing. [11, p. 7]

3.1.5 RIPPERk

RIPPER*k* is a rule learning algorithm proposed as a modification of IREP rule learning algorithm with lower error rates and faster rule induction for large datasets by W. W. Cohen in [4], implemented in K&R (Kernighan and Ritchie) C, an early version of C programming language.

3.1.6 CMake

”CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles.” [12]

3.1.7 Doxygen

Doxygen is a configurable tool for generating documentation from source code – both annotated using specific comment syntax and unannotated. Doxygen is also able to visualize various kinds of relationships in the code, such as dependency graphs, class diagrams and function call diagrams. [13]

3.2 System Features

The system in its initial version [6] as implemented by Yann Dauxais can mine discriminant chronicles from user-supplied sequence sets and classify them afterwards, yielding a list of discriminant chronicles and classification results.

The implementation of the system is constrained to work with a temporal domain $\mathbb{T} \subset \mathbb{Z}$ because the implementation uses the `int` type for representing elements of $\mathbb{T}$. [5] Conceptual temporal domain of the proposed algorithm as stated in [1, p. 3] is $\mathbb{T} \subseteq \overline{\mathbb{R}}$.

3.3 High-Level System Structure

The system consists of three components as depicted in Figure 3.1 cooperating together to accomplish the desired task. Briefly:

1. The component *DC-PBC (Discriminant Chronicle Pattern-Based Classification)* is invoked by the user using its CLI with supplied input data and other parameters regarding the mining task and classification rules generation further described in its `README` file [6].

2. After data parsing and preprocessing, *DC-PBC* invokes *DCM* executable – *DCM* mines discriminant chronicles from the dataset using an implementation of the `DCM` algorithm as described in Section 1.3.1.

3. After the mining task is completed, if specified using a CLI argument, *DC-PBC* generates classification rules from the results of the mining task accomplished by *DCM* – either by invocation of an external classifier from the *WEKA* component, or by using a Python classifier implementation from *scikit-learn.*

4. Both the mining results and classification rules validation statistics if present are written to systematically named output directory structure further described in the `README` of *DC-PBC* located in [6].

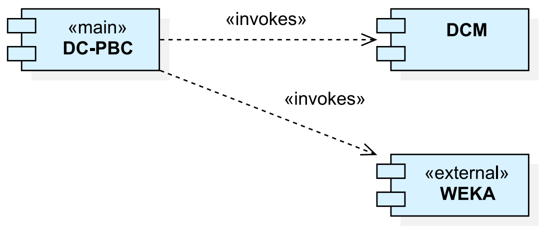

Figure 3.1: Components of the system

The communication schema of individual components described in the previous paragraph is depicted in Figure 3.2.

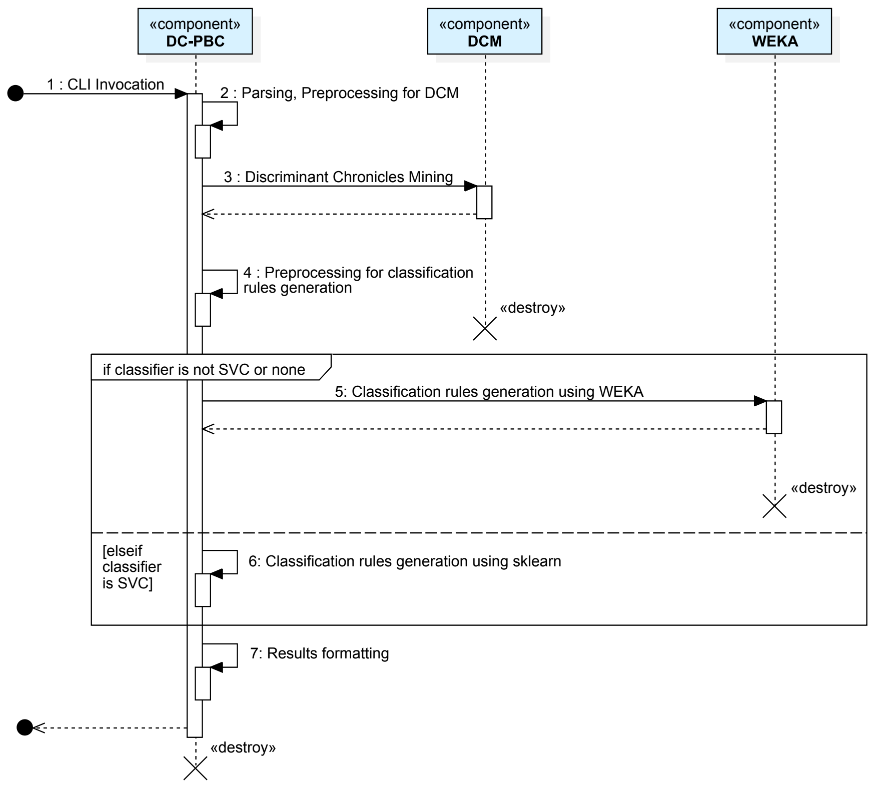
«component»
DC-PBC
«component»
DCM
«component»
WEKA
1 : CLI Invocation
2 : Parsing, Preprocessing for DCM
3 : Discriminant Chronicles Mining
4 : Preprocessing for classification rules generation
«destroy»
if classifier is not SVC or none
5: Classification rules generation using WEKA
«destroy»
[elseif classifier is SVC]
6: Classification rules generation using sklearn
7: Results formatting
«destroy»

Figure 3.2: Component communication diagram

3.3.1 Component Integration

The components *DC-PBC* and *DCM* are integrated in two ways:

1. *DC-PBC* hands the mining task over to *DCM* using simple executable invocation with CLI parameter passing – this is done for each fold as specified by the `--fold` CLI parameter and each label of the input in order to mine both chronicles discriminant for $\mathcal{S}^+$ with respect to $\mathcal{S}^-$ and chronicles discriminant for $\mathcal{S}^-$ with respect to $\mathcal{S}^+$,

2. *DCM* passes the results of the mining tasks back to *DC-PBC* using file-based integration – that means *DC-PBC* parses the result of the mining task executed by *DCM* back to its internal data representation and further generates the classification rules.

The components *DC-PBC* and *WEKA* are integrated in a similar manner as the previous couple of components, i.e.:

1. *DC-PBC* passes the classification rules generation task over to *WEKA* using executable invocation with temporary input file encoded in the `.arff` format *WEKA* accepts,
2. *WEKA* passes the results of the classification rules generation back to *DC-PBC* using file-based integration.

3.4 DC-PBC Implementation Structure

The Python implementation of the *DC-PBC* project consists of application entrypoint file `main.py` which is callable using CLI, a file `GLOBAL.py` containing paths to the executables of the *WEKA* and *DCM* components and a file `functions_p3.py` containing three classes and several functions described in the following sections.

Classes

The class `Chronicle` represents a chronicle (i.e. a couple $(\mathcal{E}, \mathcal{T})$ as elaborated in Section 1.2) and also provides the functionality to parse the output of *DCM* mining task into an instance of this class.

The class `McDataset` represents an input dataset consisting of sequences and also provides the functionality to parse input files.

The class `Dataset` provides functionalities similar to the functionality of `McDataset`, however, is not consumed by any other code.

Functions

The aforementioned file contains numerous functions. The responsibilities of the functions are briefly:

- classification rules generation using *SVC (Support Vector Classifier)* from *scikit-learn* (functions `fit_svm`, `mc_classify`),
- classification rules generation using numerous classification methods from *WEKA* along with *WEKA* input formatting and output parsing (functions `mc_generate_arff`, `generate_arff`, `mc_classify`, `classify`, `extract_res_classify`, `extract_precision`, `extract_recall`),
- classification rules validation statistics file formatting and writing (`compute_res_classify`),

and a few other utility functions.

3.5 DCM Implementation Structure

The C++ project *DCM* consists of a top-level package containing application entry file `main.cpp` and seven packages (implemented as statically linked CMake sublibraries), each with a different responsibility. Package dependencies across the project are illustrated in Figure 3.3.

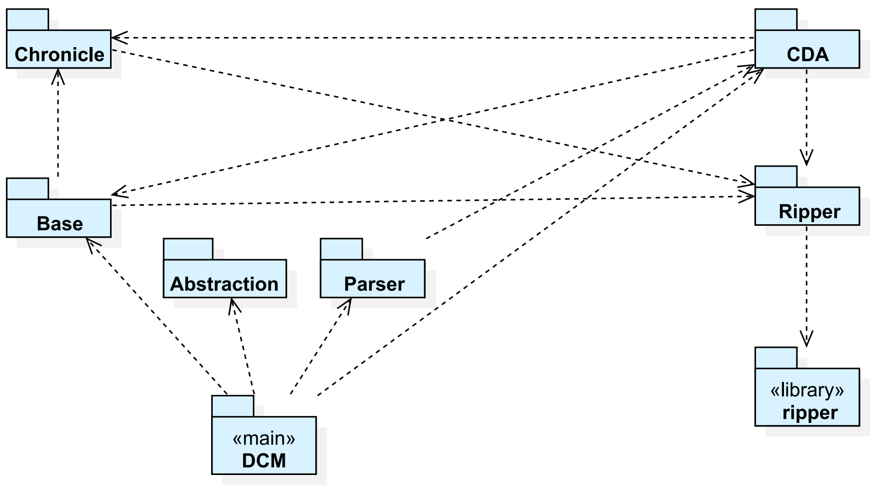

Figure 3.3: Dependencies among packages in the DCM project

The following sections will briefly describe each of the packages.

3.5.1 Abstraction

This package consists of just a single class – `Abstraction` which contains purely static methods, i.e. *is basically a collection of utility functions* used for preprocessing input data for discriminant chronicles mining using the *DCM* component. The class exposes a single method named `generatePathways` which does the aforementioned task.

3.5.2 Base

This package contains three classes - `Event`, `Base` and `VerticalEventBase`.

The class `Event` is a utility class used in algorithms contained within the methods of `Base` and `VerticalEventBase` classes – i.e. a data structure consisting of an event, start time and end time. In addition to these properties, the class also implements a comparison operator `<` (less-than).

The class `Base` is an abstract class abstracting `VerticalEventBase<T>`, declaring its interface with virtual abstract methods and defining numerous common non-virtual utility methods.

The class `VerticalEventBase<T>` which extends the class `Base` is a class representing storage for sequences with a diverse set of responsibilities, containing several implemented methods for:

- retrieving occurrences – subsequences of input sequence sets further described in Section 1.2,
- retrieving episodes for frequent chronicles mining and discriminant episodes mining,
- determining chronicle support,
- extracting the set of frequent multisets from the input sequence set – i.e. the data structure represented by the symbol `M` as illustrated in Listing 1.1,

and other utility methods.

3.5.3 CDA

This package consists of a single class called `CDA`.

The class `CDA` encapsulates the `DCM` algorithm elaborated in Section 1.1 and algorithms for frequent chronicles mining and discriminant episodes mining along with their subroutines and state.

3.5.4 Chronicle

This package consists of five classes - `Chronicle`, `EMSet`, `MSetOcc`, `TC` and `TCIterator`.

The class `EMSet` represents an event multiset, along with its additional properties, such as frequency of the multiset representing computed support of $(\mathcal{E}, \mathcal{T}_\infty)$ as elaborated in Section 1.3.1.

The class `Chronicle` represents a chronicle – a couple $(\mathcal{E}, \mathcal{T})$ as defined in Section 1.2.

The class `MSetOcc` represents an occurrence for given frequent event multiset – a subsequence of a sequence from the input sequence set.

The class `TC` represents a temporal constraint as described in Section 1.2.

The class `TCIterator` is a wrapper around a shared pointer to an instance of `TC` class. Despite its name, the implementation does not conform to any of the standardized *C++ STL (Standard Template Library)* iterator concepts, just implementing the operators `->`, unary `*`, `=`, `==` and custom methods `isNull` and `isPunct`.

3.5.5 Parser

This package is a collection of functions and is responsible for parsing input files and aside from functions intended for printing model classes provides a total of three different parsers.

The function `CSVParser(...)` parses an input in CSV (Comma-Separated Values) format if the program receives an input file with file extension `.csv`.

The function `LineParser(...)` parses tabular data separated by spaces if the input file has other extension than `.csv` and internal file structure consists of one sequence per line.

The function `TXTParser(...)` parses tabular data separated by spaces if the input file has other extensions than `.csv` and internal file format is *IBM format* (enforced by supplying the `-u` CLI parameter to the program).

3.5.6 Ripper

This package serving as an implementation-specific C++ wrapper around the RIPPER*k* third-party K&R C library described in the next section consists of a class `Rule` and a collection of utility functions used for initializing the algorithm and transforming data to and from the algorithm implementation.

The class `Rule` is a model class representing a rule as an output from the rule induction procedures of RIPPER*k* algorithm. The class represents a data structure consisting of:

- rule label – an element of $\mathbb{L}$ as defined in Section 1.3.1.2,
- rule significand array as described in Section 1.3.1.2,
- an array of interval bounds for each rule significand as described in Section 1.3.1.2,
- success and error rates (i.e. counts of samples correctly and incorrectly covered by the rule) returned from the RIPPER*k* algorithm used later for determining resulting discriminant chronicle support.

3.5.7 ripper

This third-party library provides a K&R C Programming Language implementation of the RIPPER*k* algorithm by W. W. Cohen introduced in [4].

3.6 Technical Debt

In the software engineering field, *"technical debt is the debt that accrues when you knowingly or unknowingly make wrong or non-optimal design decisions"*, the authors of [14] state in the chapter about technical debt of their book *Refactoring for Software Design Smells: Managing Technical Debt.*

Major design issues found after the evaluating source codes of the *DC-PBC* and *DCM* components are stated in the following sections.

3.6.1 General Code Issues

The following code issues were identified in both projects:

- The code occasionally contains unused imports, dependencies and functions/methods which when removed, will simplify the structure of the project and reusability of individual packages.
- Variables in the code often do not follow self-descriptive naming conventions which are perceived as a general good practice [14] - e.g. variables named `flag`, `tmp`, etc. This issue *greatly reduces code understandability and thus makes the code closed for changes and closed for extensions.*
- Some functions are very long in terms of LoCs (Lines of Code) and complex in terms of cyclomatic complexity, causing significant cognitive load [14] on the reader of the code.

3.6.2 DCM Issues

- Data types in the source files are not symbolic – i.e. their usage does not convey their meaning – e.g. the `int` data type might represent an event type or a temporal domain element. *This will present a significant challenge* when designing the generalized version of the system.
- The project contains several so-called god classes – classes with multiple responsibilities, essentially breaking the Single Responsibility Principle [14] of maintainable object-oriented code.
- Formal parameter lists of some methods are excessively long and contain boolean parameters to alter method behavior in runtime which is considered a code smell [14] concerning method invocation statement readability – namely `CDA::run` accepts 15 formal parameters and five of them are boolean.
- Some methods in numerous classes have unnecessary `public` access modifiers that should be `private` or `protected` in order to simplify public interface complexity of their owner classes.

3.7 Conclusion

This chapter described the structure of the system for discriminant chronicles mining – first its high-level structure and component communication schemata, then each individual component by describing their modules, packages, classes, methods and attributes.

Chapter 4

Analysis and Design

This chapter will introduce the changes to be implemented into the original version of the system as elaborated in Chapter 3.

Sections 4.1 and 4.2 will state the goals and non-goals of the generalization from a software implementation standpoint.

Sections 4.3 and 4.4 will propose particular changes to all components of the system, elaborated in-depth.

4.1 Goals of the Generalization

F1: The system will be able to process datasets and produce results with domain elements as real vectors of arbitrary length as introduced in Chapter 2.

F2: The system will support a new input file format further described in Section 4.4.3.

F3: The generalization itself should be implemented in a style that allows further easy customization of the domain.

4.2 Non-goals of the Generalization

N1: The generalized version of the system will not support Frequent Chronicles Mining and Discriminant Episodes Mining which are out of thesis assignment scope.

N2: The system will be able to parse only files containing tabular space-separated values (the *one-sequence-per-line format*) and the new file format stated in the requirement F2.

4.3 DCM Change Analysis

As discussed in Section 3.6, the implementation in the current as-is state is closed to extensions due to numerous code issues. To make the system easier to maintain and extend in the future for further general use-cases, the prerequisites to implementing the requirements stated in Section 4.1 will include code refactoring.

4.3.1 Refactoring

Girish Suryanarayana et *al.* in their book [14] state that *"performing refactoring is the primary means of repaying technical debts"*. As stated in Section 3.6, both *DC-PBC* and *DCM* have significant amounts of technical debt. Thus refactoring at least the most serious code issues will be needed to simplify implementing further extensions to the system.

4.3.1.1 Project Structure Refactoring

In order to simplify modular structure of both projects, unused imports, dead code, overly relaxed access modifiers and ambiguous package dependency links will be removed. The simplified structure of the project after refactoring is depicted in Figure 4.1.

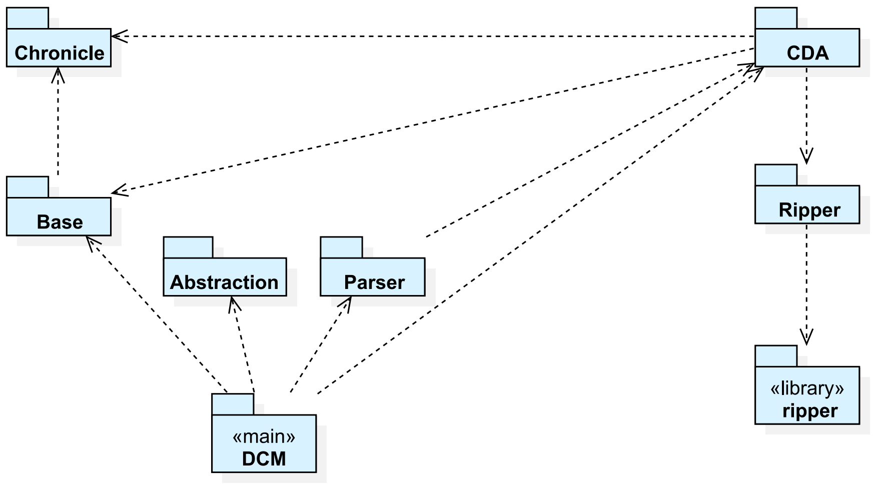

Figure 4.1: Package dependencies in the *DCM* project after refactoring

4.3.1.2 Symbolic Type Refactoring

As stated in Section 3.6, some variable types used in the code of the *DCM* project are not semantically named. In order to simplify further extensions to the system, the occurrences of the `int` type representing temporal domain elements will be changed to a type alias called `DomainElement`. The type alias approach was chosen over class inheritance because defining abstract operators with polymorphic return value types is impossible to achieve in C++.

Operators

Based on code occurrences of the `int` variables representing original temporal domain elements, any types further aliased by the `DomainElement` type alias will need to specify and implement the following set of binary operators with the following semantics:

- arithmetic operators `+`, `-` for adding/subtracting two realizations of the `DomainElement` alias,
- equality operator `==` for equality checks between two realizations of the `DomainElement` alias.

Some code occurrences also require using the `>`, `<` and `<=` operators. However, as stated in Chapter 2, temporal domain elements can not be totally ordered in general.

The occurrences of `>`, `<` and `<=` operators will be deleted or rewritten to code that does not utilize comparison operators between two `DomainElement` realizations. The statements involving sorting of `DomainElement` realizations utilizing the comparison operators transitively will also need to be deleted or rewritten. Such changes will have further impact on several aspects of `DCM` functionality and will be elaborated in the next section.

Substitution of Comparison Operators and Sorting

All operations involving comparing particular objects aliased by the alias `DomainElement` in manners of interval tests will be substituted with generalized method call `bool isInsideBounds(lowerBound, upperBound)`, representing a test whether given `DomainElement` is between `lowerBound` and `upperBound`, both included and both realizations of `DomainElement`.

When pruning redundant rules, two semi-infinite interval checks using the `>` operator in the package `CDA` in the file `CDA.cpp` on lines 40 and 42 happen. Such substitution will be dependent on internal structure and other characteristics of future `DomainElement` realizations, thus the whole code block will be extracted into a separate procedure in the `Base` package called `pruneRedundantIntervals`, intended to be reimplemented for each realization of the `DomainElement` alias. Its parameters are:

`significand` – an index of *significand* in an *example* of a relational dataset further elaborated in Section 4.3.2.4, `outputIntervals` – an ordered collection of intervals indexed by *significand* indices,

`example` – an *example* of the relational dataset.

When retrieving occurrences, event sequence sorting in the package `Base` in the file `VerticalEventBase.hh` on line 321 happens. After evaluating the rest of the code, the sequence sorted was not used anywhere. Deleting this sorting statement will be safe and will not affect *DCM* operation.

After the parsing stage of *DCM* completes, there is a sorting statement in the package `Abstraction` in `Abstraction.hh` on line 66 which sorts individual temporal elements sharing the same event type and the same input sequence. Deleting this sorting statement implies that *the dataset consisting of the DCM input files (*`pos.dat` *and* `neg.dat`*) must have all input sequence events sorted chronologically from left to right.* In situations where this does not hold, *manual data preprocessing should take place.*

Substitution of Value Literals

At numerous places in the source codes, the `int` variables representing temporal domain elements were initialized by numerical values (namely `0`, `1`, `-1`, and special values conceptually representing `inf`/`-inf`).

In initialization statements where the scalar values are initialized to value `0`, calls to the factory method `DomainElementFactory::zero()` will be introduced to *substitute the number* 0 *with its generalization dependent on the particular realization of the* `DomainElement` *alias*, such as zero vectors, zero matrices, etc.

In printing statements where special values representing $\infty/-\infty$ values occur, calls to methods `DomainElementFactory::positiveInfinity()` and `DomainElementFactory::negativeInfinity()` will be introduced to *substitute given values with their generalizations* with similar intents as described in the previous paragraph.

Other values, in particular `1` and `-1`, are used in initialization statements in the part of *DCM* responsible for mining frequent chronicles and discriminant episodes that are out of the scope of this thesis and thus will simply be removed.

Final Contracts

The final set of operators and methods the realizations of the `DomainElement` alias and the implementation of the `DomainElementFactory` class should implement are listed in Table 4.1 and 4.2.

Unit	Responsibility
operator +	adds two DomainElements together
operator -	subtracts two DomainElements
operator ==	compares two DomainElements for equality
operator <<	inserts a DomainElements into a stream for printing
method bool isInsideBounds(lo, up)	checks if the value of given DomainElement is between lo and up, both of type DomainElement

Table 4.1: DomainElement contract

Method	Responsibility
zero	constructs a DomainElement generalizing the 0 value
positiveInfinity	constructs a DomainElement generalizing the ∞ value
negativeInfinity	constructs a DomainElement generalizing the $-\infty$ value

Table 4.2: DomainElementFactory class contract

4.3.2 Changes Related to Requirements

4.3.2.1 RealVector Class

Related to symbolic type refactoring as stated in the previous section is an implementation of vector data type `RealVector` representing a vector of arbitrary length with real numbers as its elements – i.e. $t \in \overline{\mathbb{R}}^d$. The type alias `DomainElement` will refer to this class. Both the `RealVector` class and the `DomainElement` alias will be placed into a new package called `Domain` in the *DCM* project.

Class Hierarchy

The class `RealVector` will extend the class `std::vector<double>` from C++ STL in order to be easily handled like a homogenous vector of floating-point numbers. Because of utilizing the inheritance, the class will offer standard `std::vector` methods, such as `push_back`, `clear`, etc.

Although inheriting from library-supplied classes such as `std::vector` is often seen as an unsafe antipattern, the utilization of inheritance, in this case, is not unsafe because the `RealVector` class will not redefine the original functionality and implementation details of its parent class `std::vector`, thus any changes made to C++ STL in the future will not break the functionalities of the `RealVector` class and vice versa.

The class hierarchy described in the paragraphs above is depicted in Figure 4.2.

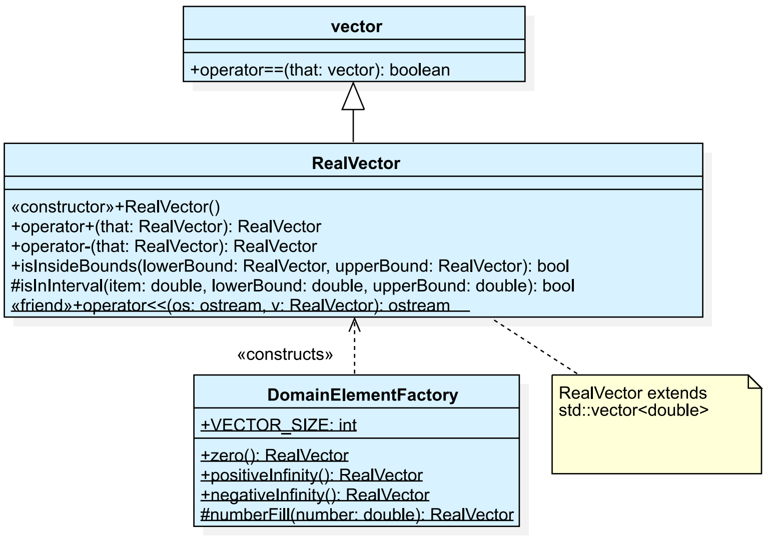
vector
+operator==(that: vector): boolean
RealVector
«constructor»+RealVector()
+operator+(that: RealVector): RealVector
+operator-(that: RealVector): RealVector
+isInsideBounds(lowerBound: RealVector, upperBound: RealVector): bool
#isInInterval(item: double, lowerBound: double, upperBound: double): bool
«friend»+operator<<(os: ostream, v: RealVector): ostream
«constructs»
DomainElementFactory
+VECTOR_SIZE: int
+zero(): RealVector
+positiveInfinity(): RealVector
+negativeInfinity(): RealVector
#numberFill(number: double): RealVector
RealVector extends std::vector<double>

Figure 4.2: Temporal domain element class hierarchy (DCM)

Operators

As required for each realization of the `DomainElement` alias, the `RealVector` class will define the operators `+` and `-` and will inherit the operator `==` from `std::vector` which already implements the desired functionality for comparing two vectors. Specific operator definitions are listed below.

Given two vectors $\vec{a} = (a_1, a_2, \dots, a_d)$ and $\vec{b} = (b_1, b_2, \dots, b_d)$, $\vec{a}, \vec{b} \in \overline{\mathbb{R}}^d$:

- The operator `+` will perform vector addition, i.e. the operation denoted as $\vec{a} + \vec{b}$ will denote the vector $(a_1 + b_1, a_2 + b_2, \dots, a_d + b_d) \in \overline{\mathbb{R}}^d$.
- The operator `-` will perform vector subtraction, i.e. the operation denoted as $\vec{a} - \vec{b}$ will denote the vector $(a_1 - b_1, a_2 - b_2, \dots, a_d - b_d) \in \overline{\mathbb{R}}^d$.
- The operator `==` will perform *equality* vector comparison, i.e. the comparison denoted as $\vec{a} = \vec{b}$. $\vec{a} = \vec{b} \iff a_1 = b_1 \wedge a_2 = b_2 \wedge \dots \wedge a_d = b_d$.

Hyperrectangle Test Implementation

The method `bool isInsideBounds(lowerBound, upperBound)` will perform the hyperrectangle test as defined in Section 2.3.1.

Constructor

The parameterless constructor of `RealVector` will create a blank vector with the size equal to zero, ready to be initialized with desirable elements if needed, using standard `std::vector` methods.

DomainElementFactory Implementation

The methods `zero()`, `positiveInfinity()` and `negativeInfinity()` of the `DomainElementFactory` class will construct instances of `RealVector`, such that its size equals to a static integer member variable `VECTOR_SIZE` of the class `DomainElementFactory` discussed below and all its elements are equal to 0, ∞ and $-\infty$ respectively.

Aside from the `zero()`, `positiveInfinity()` and `negativeInfinity()` factory methods in the `DomainElementFactory` class declared in Section 4.3.1.2, a new static class member variable called `VECTOR_SIZE` will be added to the class, representing the size of vectors the system will operate with.

Aware of the risks of using public static class members which share their flaws with global variables, this approach was chosen because:

- Each run of the mining task will utilize vectors of unified size and will not change it after being set in the parsing phase of the algorithm.
- Every possible realization of the `DomainElement` alias produced by the aforementioned factory methods `zero()`, `positiveInfinity()` and `negativeInfinity()` might require different production interface (e.g. for a matrix *row count* and *column count*, for a vector *size*), however *will in every case need to generalize a zero or positive/negative infinity value.* The usage of a static member will:
  1. *unify the interfaces* of the factory methods to be parameterless for every possible realization of the `DomainElement` alias,
  2. *deminish the impact on formal parameter lists of numerous caller methods* that would otherwise require a change involving adding a parameter representing the vector size or different information needed for producing the desired $0/\infty/-\infty$ value generalizations for further extensions, such as matrix row/column count.

The `DomainElementFactory` class will be placed into the `Domain` package in the *DCM* project.

Stream Insertion Operator Function

In addition to the aforementioned operators, implementing a binary operator `<<` for stream insertion used for printing the values to the output will also be needed. The operator will be implemented as a function because C++ imposes limits on left-hand side and right-hand side operand types when implementing operator methods.

The function will be declared with `friend` modifier, allowing it to access the `private` and `protected` members of `RealVector` objects.

The function will format the vector as a comma-separated list of its elements enclosed in `<` and `>`.

For instance, for a `RealVector` representing a vector $(1, 2, 3, 4, 5)$, the formatted version of it will be `<1,2,3,4,5>`.

Redundant Interval Pruning

As introduced in Section 4.3.1.2, the procedure `pruneRedundantIntervals` will be altered – its original code will be wrapped into a `for` statement called repeatedly component-wise, and altered to prune insignificant intervals for each component of vector doubles in supplied vector interval set represented by the `outputIntervals` parameter. Listing 4.1 illustrates the original body of the function as extracted in Section 4.3.1.2 and Listing 4.2 illustrates the updated body of the function.

```
void pruneRedundantIntervals(
        unsigned int significand,
        std::vector<std::vector<DomainElement>>& outputIntervals,
        const std::vector<DomainElement>& example) {

    if (outputIntervals[significand][0] > example[significand]) {
        outputIntervals[significand][0] = example[significand];
    }
    else if (outputIntervals[significand][1] < example[significand]) {
        outputIntervals[significand][1] = example[significand];
    }
}
```

Listing 4.1: Original `pruneRedundantIntervals` function body

```
void pruneRedundantIntervals(
    unsigned int significand,
    std::vector<std::vector<DomainElement>>& outputIntervals,
    const std::vector<DomainElement>& example) {

  for (int v = 0; v < DomainElementFactory::VECTOR_SIZE; ++v) {
    if (outputIntervals[significand][0][v] > example[significand][v]) {
      outputIntervals[significand][0][v] = example[significand][v];
    }
    else if (outputIntervals[significand][1][v] < example[significand][v]) {
      outputIntervals[significand][1][v] = example[significand][v];
    }
  }
}
```

Listing 4.2: Generalized `pruneRedundantIntervals` function body

The `pruneRedundantIntervals` procedure will be changed according to the description in the paragraph above and moved to a separate C++ module `interval_pruning.hh` located in the `Domain` package.

4.3.2.2 DCM CLI Changes

Associated with the change involving implementation of `RealVector` class described in the next subsection, the d fulfilling $d \in \mathbb{N}$ in temporal domain $\overline{\mathbb{R}}^d$, later referred to as *vector size* should be specifiable by the user using a new, mandatory CLI parameter `-s/--vecsize`, standing for *vector size*.

This parameter will be stored into the `VECTOR_SIZE` static member variable of the `DomainElementFactory` class.

The `-e/--episode`, `-a/--all_different` and `-n/--not_calc_freq` parameters related to frequent chronicles mining and discriminant episodes mining will be removed as the aforementioned mining methods are out of the scope of this thesis.

4.3.2.3 DCM Input Parser Changes

The parser functions `CSVParser`, `TXTParser` and related code calling them will be removed from the DCM project based on the non-goal N2 introduced in Section 4.2.

The parser function `LineParser` will be changed accordingly to be able to parse variable-length vector domain inputs. Further analysis of changes to the input file format will be discussed in the following sections.

Changes to Input File Format

The input file format will be derived from the current state of the format used for `LineParser`, a parser parsing files with the *one-sequence-per-line format.*

```
A 15 B 23 C 29 D 37
A 15 B 16 C 28 D 29
A 15 B 16 C 25 D 26
A 15 C 21 B 22 D 28
```

Listing 4.3: Old file format example

```
A 15.0 15.0 B 23.0 23.0 C 29.0 29.0 D 37.0 37.0
A 15.0 15.0 B 16.0 16.0 C 28.0 28.0 D 29.0 29.0
A 15.0 15.0 B 16.0 16.0 C 25.0 25.0 D 26.0 26.0
A 15.0 15.0 C 21.0 21.0 B 22.0 22.0 D 28.0 28.0
```

Listing 4.4: Changed file format example

4.3.2.4 Rule Induction Process Changes

At the moment, the Ripper C++ wrapper used for rule induction induces rules with scalar values from input datasets containing scalar values as described in Section 1.3.1.2. The process of rule induction in the original implementation is illustrated in Figure 4.3.

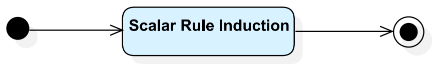

Figure 4.3: Original rule induction process

This will be changed so that the wrapper will accept an input dataset containing `RealVector` objects as inter-event durations and will produce a ruleset with rule values of `RealVector` type.

Because the `ripper` third-party library does not accept datasets with vectors as features nor produces rulesets consisting of vector values, the numerical rule induction will be preceded by a *global preprocessing phase*, followed by a *local indexing phase* and a *global postprocessing phase*. The scalar rule induction algorithm which will be unchanged will be called repeatedly componentwise inbetween. High-level overview of the new rule induction process is illustrated in Figure 4.4.

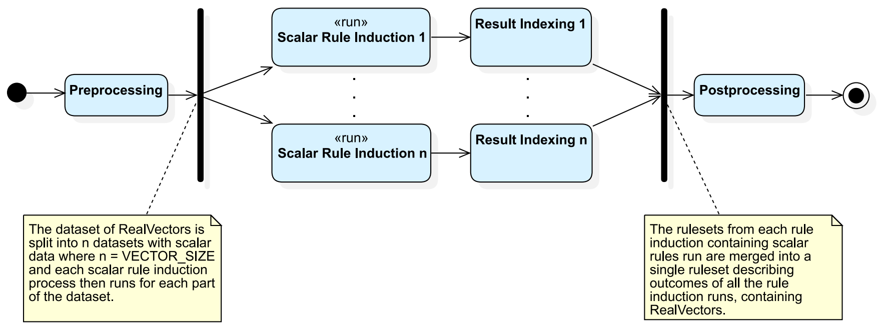
Preprocessing
«run»
Scalar Rule Induction 1
«run»
Scalar Rule Induction n
Result Indexing 1
Result Indexing n
Postprocessing
The dataset of RealVectors is split into n datasets with scalar data where n = VECTOR_SIZE and each scalar rule induction process then runs for each part of the dataset.
The rulesets from each rule induction containing scalar rules run are merged into a single ruleset describing outcomes of all the rule induction runs, containing RealVectors.

Figure 4.4: New rule induction process

Note that the input dataset containing the `RealVector` values is generated in earlier stages of the DCM algorithm, based on its corresponding frequent event multiset $\mathcal{E}$ and input sequence set $\mathcal{S}$ as elaborated in Section 1.3.1.2.

New Data Structures

The `Rule` class representing a rule from the original implementation as described in Section 3.5.6 will be removed.

A new class `SingleRule` representing a rule obtained from a single RIPPER*k* rule induction run will be introduced. Rule values in this class will be represented by values of type `double`.

A new class `MultiRule` representing a rule produced by the global postprocessing phase, representing a result of multiple rule induction runs combined will be introduced and will be used as the final rule induction wrapper output. Rule values in this class will be represented by objects of the `RealVector` class.

Both classes – `SingleRule` and `MultiRule` will share the same interface encapsulated into the class `GeneralRule<T>` partly derived from the original `Rule` class in order to maintain code compatibility. The `SingleRule` class will, in addition to the `GeneralRule<T>` class, store a set of *EID*s correctly/incorrectly covered by a given rule using its member variables `correctExamples` and `incorrectExamples` respectively. The exact class hierarchy and class interfaces are depicted in Figure 4.5.

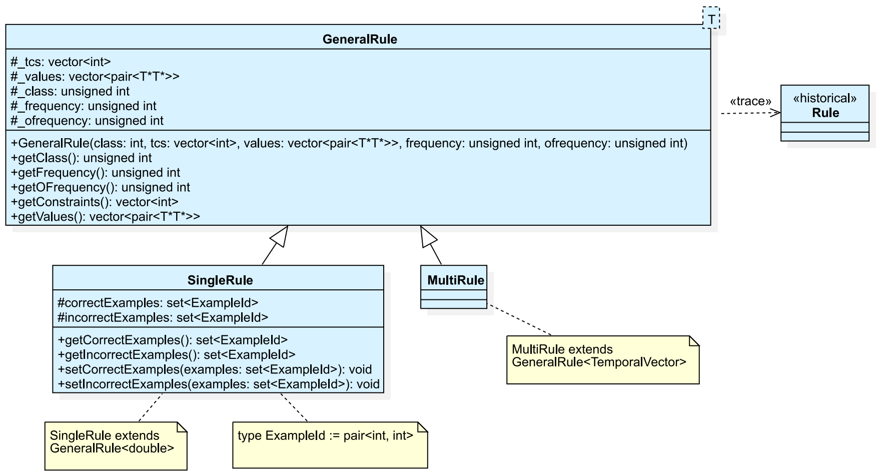
T
GeneralRule
#_tcs: vector<int>
#_values: vector<pair<T*T*>>
#_class: unsigned int
#_frequency: unsigned int
#_ofrequency: unsigned int
+GeneralRule(class: int, tcs: vector<int>, values: vector<pair<T*T*>>, frequency: unsigned int, ofrequency: unsigned int)
+getClass(): unsigned int
+getFrequency(): unsigned int
+getOFrequency(): unsigned int
+getConstraints(): vector<int>
+getValues(): vector<pair<T*T*>>
«trace»
«historical»
Rule
SingleRule
#correctExamples: set<ExampleId>
#incorrectExamples: set<ExampleId>
+getCorrectExamples(): set<ExampleId>
+getIncorrectExamples(): set<ExampleId>
+setCorrectExamples(examples: set<ExampleId>): void
+setIncorrectExamples(examples: set<ExampleId>): void
MultiRule
MultiRule extends GeneralRule<TemporalVector>
SingleRule extends GeneralRule<double>
type ExampleId := pair<int, int>

Figure 4.5: Rule class hierarchy

Method	Responsibility
`getConstraints`	retrieves the significands of the rule
`getValues`	retrieves the temporal constraints of the rule
`getClass`	retrieves the label of the rule
`getFrequency`	retrieves the number of examples correctly covered by the rule
`getOFrequency`	retrieves the number of examples incorrectly covered by the rule
`getCorrectExamples`	retrieves the set of EIDs correctly covered by the rule
`getIncorrectExamples`	retrieves the set of EIDs incorrectly covered by the rule
`setCorrectExamples`	changes the set of EIDs correctly covered by the rule
`setIncorrectExamples`	changes the set of EIDs incorrectly covered by the rule

Table 4.3: `GeneralRule<T>`, `SingleRule` and `MultiRule` class methods

All three classes – `SingleRule`, `MultiRule` and `GeneralRule<T>` will be implemented to the package `Ripper` where the RIPPER*k* C++ wrapper code resides.

Rule Induction Preprocessing

The preprocessing phase will split the input relational dataset containing inter-event durations as `RealVector` objects with vector size equal to n into n relational datasets with `double` floating point number values as features. The aforementioned transformation named `SplitDataset` is depicted in Listing 4.5.

```
SplitDataset(dataset) {
  D := [].

  for (i of {1, ..., VECTOR_SIZE}) {
    newTable := emptyTable().

    for (row of dataset) {
      newRow := emptyTableRow().

      newRow[sid] := row[sid].
      newRow[label] := row[label].

      for (vector of row[durations]) {
        if (vector != NULL) {
          newRow[durations].push(vector[i]).
        }
        else {
          newRow[durations].push(NULL).
        }
      }

      newTable.push(newRow). // EIDs are preserved by maintaining
                             // the same index order
    }

    D.push(dataset).
  }

  return D.
}
```

Listing 4.5: `SplitDataset` algorithm pseudocode

The meaning of symbols in the pseudocode depicted in Listing 4.5 is as follows:

The `dataset` parameter represents the generalized relational dataset as defined in Section 2.3.5 (an example of such dataset is illustrated in Table 4.4), with individual rows indexed by *EID*, containing *SID*, *label* and an ordered collection of *inter-event durations* as `RealVector` objects indexed by *SignificandIndex* illustrated in Table 4.7 – the data structure implementing the dataset is illustrated in Table 4.8,

The `newRow` data structure represents a table row with the structure same as the rows of the `dataset` parameter, except that the inter-event durations are represented by the `double` value type,

The `newTable` data structure represents a table representing a scalar dataset – i.e. an ordered collection of table rows, again indexed by *EID* – examples of such datasets are illustrated in Table 4.5 and Table 4.6,

The `D` data structure represents an ordered collection of relational datasets (here represented by tables) with `double` floating point numbers as features, maintaining the same inter-event duration column set as the input table supplied to the `dataset` parameter. The RIPPER*k* algorithm will use these datasets for scalar rule induction.

An example of operation of the `SplitDataset` algorithm is depicted in the tables below – the relational dataset as input to the preprocessing phase depicted in Table 4.4 is split into two relational dataset parts called *slices* depicted in Table 4.5 and Table 4.6.

EID	SID	$\mathcal{A}_{A\to B}$	$\mathcal{A}_{B\to C}$	$\mathcal{A}_{A\to C}$	Label
1	1	$(1, 2.4)$	$(-4.6, 5.1)$	$(6.3, 0)$	$+$
2	1	$(0, 0)$	$(9.6, -7.4)$	$(0, 1.9)$	$+$
3	2	$(1.6, -2)$	$(4, 7.8)$	$(3.9, -1.5)$	$+$
4	3	$(-3.6, 0.9)$	-	-	$-$

Table 4.4: Relational dataset before preprocessing

EID	SID	$\mathcal{A}_{A\to B}$	$\mathcal{A}_{B\to C}$	$\mathcal{A}_{A\to C}$	Label
1	1	1	-4.6	6.3	$+$
2	1	0	9.6	0	$+$
3	2	1.6	4	3.9	$+$
4	3	-3.6	-	-	$-$

Table 4.5: Relational dataset slice for the first rule induction run

The `Ripper` wrapper will run the scalar numerical rule induction as described in Section 1.3.1.2 for each such relational dataset slice separately and will index its results into a rule map using the `IndexRuleset` algorithm described in the next section.

Due to as-is implementation data structure constraints, the `SplitDataset` transformation will be implemented into the `ripper::run` function while also utilizing calls to a new procedure called `ripper::splitDatasetPart`.

EID	SID	$\mathcal{A}_{A\to B}$	$\mathcal{A}_{B\to C}$	$\mathcal{A}_{A\to C}$	Label
1	1	2.4	5.1	0	+
2	1	0	−7.4	1.9	+
3	2	−2	7.8	−1.5	+
4	3	0.9	-	-	−

Table 4.6: Relational dataset slice for the second rule induction run

Significand	$\mathcal{A}_{A\to B}$	$\mathcal{A}_{B\to C}$	$\mathcal{A}_{A\to C}$
SignificandIndex	1	2	3

Table 4.7: Index mapping used for the implementation of relational datasets

#	sid	durations		label
		SignificandIndex	duration	
1	1	1	`RealVector:<1,2.4>`	`pos`
		2	`RealVector:<-4.6,5.1>`	
		3	`RealVector:<6.3,0>`	
		SignificandIndex	duration	
2	1	1	`RealVector:<0,0>`	`pos`
		2	`RealVector:<9.6,-7.4>`	
		3	`RealVector:<0,1.9>`	
		SignificandIndex	duration	
3	2	1	`RealVector:<1.6,-2>`	`pos`
		2	`RealVector:<4,7.8>`	
		3	`RealVector:<3.9,-1.5>`	
		SignificandIndex	duration	
4	3	1	`RealVector:<-3.6,0.9>`	`neg`
		2	`NULL`	
		3	`NULL`	

Table 4.8: Implementation of the relational dataset from Table 4.4

Rule Indexing

The rule indexing phase realized by the new `IndexRuleset` algorithm introduced below will store the rules produced by all scalar rule induction runs into a shared data structure depicted in Table 4.11. Such indexation will allow the algorithm of the postprocessing phase to access the data more optimally.

Pseudocode of the `IndexRuleset` algorithm is depicted in Listing 4.6.

```
IndexRuleset(ruleMap&, scalarRuleset, runIndex) {
  // IndexRuleset mutates the ruleMap parameter
  // value, thus does not return anything

  for (scalarRule of scalarRuleset) {
    ruleMap[scalarRule.significands][runIndex] := scalarRule.
  }
}
```

Listing 4.6: `IndexRuleset` algorithm pseudocode

The meaning of symbols in the `IndexRuleset` pseudocode depicted in Listing 4.7 is as follows:

The `ruleMap` parameter represents a mutable reference to the output rule map with rule significand set as keys and maps with run numbers assigned to rules as values – an example of such rule map is illustrated in Table 4.11,

The `scalarRuleset` parameter represents an ordered collection of scalar rules,

The `runIndex` parameter represents ordinal number of scalar rule induction run,

The `scalarRule` variable represents a rule produced by a scalar rule induction run with `runIndex`-th dataset part as input – its only important member variable, for now, is `significands` – an ordered collection of rule significands as defined in Section 1.3.1.2.

Tables 4.9 and 4.10 depict the results of two scalar rule induction runs. Such results will be subsequently stored into a rule map illustrated in Table 4.11, later passed to the `ruleMap` parameter of the `MergeRulesets` algorithm introduced in the next section.

The $s(+)$ and $s(-)$ table header identifiers in Tables 4.9, 4.10, 4.11, 4.12 and 4.13 denote sets of examples represented by *EID*s correctly and incorrectly covered by the given rule.

Note that the results of scalar rule induction illustrated below are intended to illustrate the effect of the `MergeRulesets` algorithm and thus as such do not correspond to the input dataset slices illustrated in Tables 4.5 and 4.6.

Rule	$s(+)$	$s(-)$
$\mathcal{A}_{A\to B} \geq -2.4 \wedge \mathcal{A}_{A\to B} \leq 1.3 \implies +$	$\{1,2,3,4\}$	$\emptyset$
$\mathcal{A}_{A\to B} \leq 6 \wedge \mathcal{A}_{B\to C} \geq 0 \wedge \mathcal{A}_{B\to C} \leq 1 \implies +$	$\{1,2\}$	$\{3,4\}$

Table 4.9: Ruleset produced by the first run of rule induction

The `IndexRuleset` algorithm will be implemented into the `indexRuleset` procedure in the `Ripper` package.

Rule	$s(+)$	$s(-)$
$\mathcal{A}_{A\to B} \geq 0 \implies +$	$\{1,2,3\}$	$\{4\}$

Table 4.10: Ruleset produced by the second run of rule induction

Significand Set	Run Results			
	Run No.	Run Rule	$s(+)$	$s(-)$
$\{\mathcal{A}_{A\to B}\}$	1	$\mathcal{A}_{A\to B} \geq -2.4 \wedge \mathcal{A}_{A\to B} \leq 1.3 \implies +$	$\{1,2,3,4\}$	$\emptyset$
	2	$\mathcal{A}_{A\to B} \geq 0 \implies +$	$\{1,2,3\}$	$\{4\}$
	Run No.	Run Rule	$s(+)$	$s(-)$
$\{\mathcal{A}_{A\to B}, \mathcal{A}_{B\to C}\}$	1	$\mathcal{A}_{A\to B} \leq 6 \wedge \mathcal{A}_{B\to C} \geq 0 \wedge \mathcal{A}_{B\to C} \leq 1 \implies +$	$\{1,2\}$	$\{3,4\}$
	2	-	-	-

Table 4.11: Rule map containing run rulesets from Table 4.9 and 4.10

Rule Induction Postprocessing

After each scalar rule induction run completes and their results are stored into the rule map as described in the previous section, the rulesets will be merged using the `MergeRulesets` algorithm proposed in Listing 4.7.

```
MergeRulesets(ruleMap) {
  R := emptySet().

  for (runsRules of ruleMap) {
    // merge rules from all runs sharing the same
    // rule significand sets and input EID sets
    multiRule := MergeRules(runsRules).

    R.add(multiRule).
  }

  return R.
}
```

Listing 4.7: `MergeRulesets` algorithm pseudocode

The meaning of symbols in the `MergeRulesets` pseudocode depicted in Listing 4.7 is as follows:

The `ruleMap` parameter represents a table as a result of rule induction over the same set of *EID*s with rule significand set as keys and tables with run numbers assigned to rules as values – an example of such map is illustrated in Table 4.11,

The `runsRules` variable is a key-value pair representing a single item of the `ruleMap` collection, The `R` data structure represents the final set of rules with `RealVector` objects as values.

```
MergeRules(runsRules) {
  outRule := MultiRule().
  outRule.label := +. // RIPPERk runs in as-is implementation state
                      // produce only rules with positive label
  outRule.significands := runsRules.key. // runsRules.key contains
                                         // the significands array

  for (si := significand.index of runsRules.key) {
    outRule.values[si].lowerBound := negativeInfinity().
    outRule.values[si].upperBound := positiveInfinity().
  }

  outCorrectExamples := emptySet().
  outIncorrectExamples := emptySet().

  for ((runIndex, runRule) of runsRules.value) {
    MergeValues(
      runRule.significands,
      outRule.values,
      runRule.values,
      runIndex
    ).

    if (outCorrectExamples.isEmpty()) {
      outCorrectExamples := runRule.correctExamples.
    }
    else {
      outCorrectExamples := intersect(
        outCorrectExamples,
        runRule.correctExamples
      ).
    }

    if (outIncorrectExamples.isEmpty()) {
      outIncorrectExamples := runRule.incorrectExamples.
    }
    else {
      outIncorrectExamples := intersect(
        outIncorrectExamples,
        runRule.incorrectExamples
      ).
    }
  }

  outRule.correctExampleCount := outCorrectExamples.size().
  outRule.incorrectExampleCount := outIncorrectExamples.size().

  return outRule.
}
```

Listing 4.8: `MergeRules` procedure pseudocode

The meaning of symbols in the `MergeRules` pseudocode depicted in Listing 4.8 is as follows:

The `runsRules` parameter is a key-value pair where the key is an ordered collection of rule significands and the value is a key-value pair consisting of run number as key and run rule as value,

The `outRule` variable represents an output rule which contains the merged RIPPER*k* C++ wrapper run results – its member variables are:

- `label` – a label assigned to the rule located in rule conclusion as defined in Section 1.3.1.2,
- `significands` – an ordered collection of rule significands as defined in Section 1.3.1.2,
- `correctExampleCount` – a number representing the count of correctly covered relational dataset examples by the rule,
- `incorrectExampleCount` – a number representing the count of incorrectly covered relational dataset examples by the rule,
- `values` – an ordered collection of rule values for each significand from the rule signicicand collection located in the `significands` member variable,

The `negativeInfinity()` and `positiveInfinity()` functions represent the vectors $(-\infty, \dots, -\infty)$ and $(\infty, \dots, \infty)$, of desired vector size – the functions will be implemented into the `DomainElementFactory` class and the size of the vectors constructed will be equal to `VECTOR_SIZE`,

The `outCorrectExamples` variable represents the set of correctly covered relational dataset examples by the merged rule, represented by *EID*s,

The `outIncorrectExamples` variable represents the set of incorrectly covered relational dataset examples by the merged rule, represented by *EID*s,

The `runIndex` variable represents the ordinal number of scalar rule induction run,

The `runRule` variable represents a rule produced by a scalar rule induction run with `runIndex`-th dataset part as input – its member variables are:

- `label` – a label assigned to the rule located in rule conclusion as defined in Section 1.3.1.2,
- `significands` – an ordered collection of rule significands as defined in Section 1.3.1.2,
- `correctExamples` – a set of correctly covered relational dataset examples by the rule, represented by *EID*s,

- `incorrectExamples` – a set of incorrectly covered relational dataset examples by the rule, represented by *EID*s,
- `values` – an ordered collection of rule values for each significand from rule signicicand collection located in the `significands` member variable.

```
MergeValues(significands, resultValues&, inputValues, runIndex) {
  // MergeValues mutates the resultValues parameter
  // value, thus does not return anything

  for (si := significand.index of significands) {
    resultValues[si].lowerBound[runIndex] := CoalesceValue(
      inputValues[si].lowerBound,
      BoundType.LOWER_BOUND
    ).
    resultValues[si].upperBound[runIndex] := CoalesceValue(
      inputValues[si].upperBound,
      BoundType.UPPER_BOUND
    ).
  }
}
```

Listing 4.9: `MergeValues` procedure pseudocode

The meaning of symbols in the `MergeValues` pseudocode depicted in Listing 4.9 is as follows:

The `significands` parameter represents an ordered collection of rule significands,

The `resultValues` parameter represents a mutable reference to an ordered collection of merged rule values for each rule significand,

The `inputValues` parameter represents an ordered collection of scalar rule induction output rule values for each rule significand,

The `runIndex` parameter represents ordinal number of a scalar rule induction run,

The `si` variable represents an index of a significand in the `significands` collection,

The `BoundType` enumeration models a type of interval bound and has got two members: `LOWER_BOUND` representing lower bound of an interval and `UPPER_BOUND` representing upper bound of an interval.

```
CoalesceValue(ruleValue?, boundType) {
  if (boundType == BoundType.LOWER_BOUND) {
    return NEGATIVE_INFINITY if ruleValue == undefined else ruleValue.
  }
  else if (boundType == BoundType.UPPER_BOUND) {
    return POSITIVE_INFINITY if ruleValue == undefined else ruleValue.
  }
}
```

Listing 4.10: `CoalesceValue` procedure pseudocode

The meaning of symbols in the `CoalesceValue` pseudocode depicted in Listing 4.10 is as follows:

The `ruleValue` parameter represents a scalar number which might or might not be defined,

The `boundType` parameter represents interval bound type and can be of `BoundType` enumeration type described in symbol explanation for the `MergeValues` procedure,

The `NEGATIVE_INFINITY` constant represents a numeric value for negative infinity, i.e. $-\infty$,

The `POSITIVE_INFINITY` constant represents a numeric value for positive infinity, i.e. ∞.

Table 4.13 illustrates the result of running the `MergeRulesets` algorithm with data illustrated in Table 4.12 as its input.

Significand Set	Run Results			
	Run No.	Run Rule	$s(+)$	$s(-)$
$\{\mathcal{A}_{A\to B}\}$	1	$\mathcal{A}_{A\to B} \geq -2.4 \wedge \mathcal{A}_{A\to B} \leq 1.3 \implies +$	$\{1,2,3,4\}$	$\emptyset$
	2	$\mathcal{A}_{A\to B} \geq 0 \implies +$	$\{1,2,3\}$	$\{4\}$
	Run No.	Run Rule	$s(+)$	$s(-)$
$\{\mathcal{A}_{A\to B}, \mathcal{A}_{B\to C}\}$	1	$\mathcal{A}_{A\to B} \leq 6 \wedge \mathcal{A}_{B\to C} \geq 0 \wedge \mathcal{A}_{B\to C} \leq 1 \implies +$	$\{1,2\}$	$\{3,4\}$
	2	-	-	-

Table 4.12: Rule map as an input to the `MergeRulesets` algorithm

Merged Rule	$\lvert s(+)\rvert$	$\lvert s(-)\rvert$
$\mathcal{A}_{A\to B} \geq (-2.4, 0) \wedge \mathcal{A}_{A\to B} \leq (1.3, \infty) \implies +$	$\lvert\{1,2,3\}\rvert = 3$	$\lvert\emptyset\rvert = 0$
$\mathcal{A}_{A\to B} \leq (6, \infty) \wedge \mathcal{A}_{B\to C} \geq (0, -\infty) \wedge \mathcal{A}_{B\to C} \leq (1, \infty) \implies +$	$\lvert\{1,2\}\rvert = 2$	$\lvert\{3,4\}\rvert = 2$

Table 4.13: Rules extracted using `MergeRulesets` from rule map in Table 4.12

The `MergeRulesets` algorithm will be implemented into the `mergeRulesets` function and the `mergeRules`, `mergeValues` and `coalesceValue` procedures implement the `MergeRules`, `MergeValues` and `CoalesceValue` procedures proposed in Listing 4.8, 4.9 and 4.10 respectively.

All the functions and procedures will be implemented into the `Ripper` package, alongside the original functions used as a wrapper around the `ripper` K&R C third-party library.

RIPPER Input Data Format Changes

The scalar rule induction implemented by the `ripper` third-party library operating on scalar values will newly accept a collection of floating point numbers represented by the `double` data type instead of a collection of `int` values.

4.4 DC-PBC Change Analysis

DC-PBC will undergo only the *substitution of value literals* refactoring – other refactoring techniques analogous to the techniques used in *DCM* component refactoring will not be utilized because:

- the structure of the project is simple – consisting of only 3 Python modules as stated in Section 3.4,
- changes to the *DC-PBC* codebase regarding the replacement of `int` variables by a custom temporal domain type is straight-forward in loosely typed languages without required variable types such as Python.

4.4.1 Substitution of Value Literals

Substitution of value literals in the *DC-PBC* component will be straight-forward – the values ∞ and $-\infty$ used for initializing several variables will be replaced by calls to the `positive_infinity` and `negative_infinity` methods of the class `DomainElementFactory` introduced in the next section.

4.4.2 RealVector Class

Analogous to the `RealVector` class proposed for *DCM* component in Section 4.3.2.1, this class with the same name will represent a vector of arbitrary length with real numbers as its elements – i.e. $t \in \overline{\mathbb{R}}^d$.

Contrary to the C++ implementation of the `RealVector` class, this class will not have any abstraction declaration such as type alias because Python is loosely typed and the codebase of *DC-PBC* itself does not utilize optional type annotations – adding type annotations would be counterproductive.

In addition to the `RealVector` class, a class `DomainElementFactory` analogous to the one introduced in Section 4.3.2.1 will be implemented, with the same intents and design considerations considering its `VECTOR_SIZE` static member variable as its *DCM* counterpart. Class diagram of the aforementioned classes is depicted in Figure 4.6.

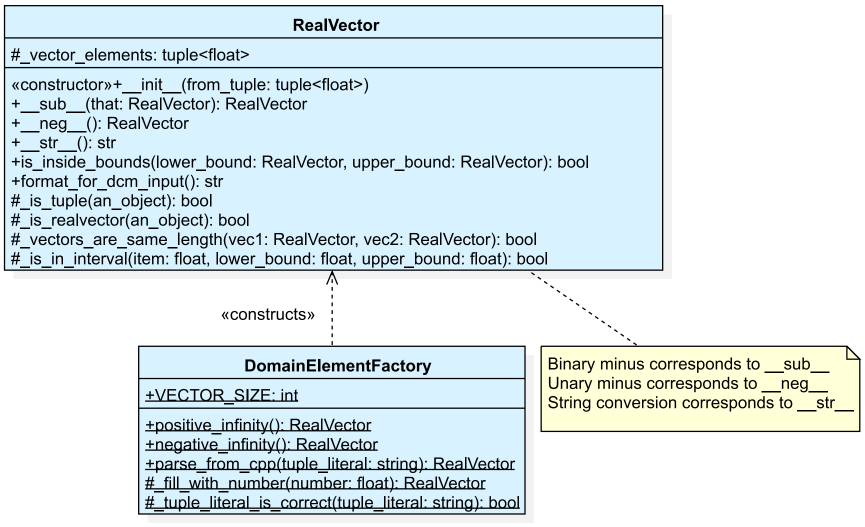
RealVector
#_vector_elements: tuple<float>
«constructor»+__init__(from_tuple: tuple<float>)
+__sub__(that: RealVector): RealVector
+__neg__(): RealVector
+__str__(): str
+is_inside_bounds(lower_bound: RealVector, upper_bound: RealVector): bool
+format_for_dcm_input(): str
#_is_tuple(an_object): bool
#_is_realvector(an_object): bool
#_vectors_are_same_length(vec1: RealVector, vec2: RealVector): bool
#_is_in_interval(item: float, lower_bound: float, upper_bound: float): bool
«constructs»
DomainElementFactory
+VECTOR_SIZE: int
+positive_infinity(): RealVector
+negative_infinity(): RealVector
+parse_from_cpp(tuple_literal: string): RealVector
#_fill_with_number(number: float): RealVector
#_tuple_literal_is_correct(tuple_literal: string): bool
Binary minus corresponds to __sub__
Unary minus corresponds to __neg__
String conversion corresponds to __str__

Figure 4.6: Temporal domain element class hierarchy (DC-PBC)

Method	Responsibility
`__sub__`	subtracts two `RealVector`s
`__neg__`	negates the `RealVector`
`__str__`	converts the `DomainElement` to a string serialization
`format_for_dcm_input`	converts the `DomainElement` to a string representation used as DCM input
`is_inside_bounds(a, b)`	checks if the `RealVector` denoted as $\vec{c}$ fulfills the hyperrectangle test, i.e. $\vec{c} \in \mathfrak{R}(\vec{a}, \vec{b})$ where `a` and `b` are of type `RealVector`

Table 4.14: `RealVector` class interface

Both `RealVector` and `DomainElementFactory` classes will be implemented in a python module `vectors.py`.

Operators

The operator set implemented in the `RealVector` class is based on code occurrences of operators that were used with the previous representation of temporal domain elements – i.e. `int` values. Specific algebraic operator and method definitions are listed below.

Given two vectors $\vec{a} = (a_1, a_2, \ldots, a_d)$ and $\vec{b} = (b_1, b_2, \ldots, b_d)$, $a, b \in \overline{\mathbb{R}}^d$:

- The operator *binary* `-` will perform vector subtraction, i.e. the operation denoted as $\vec{a} - \vec{b}$ and will represent a vector $(a_1 - b_1, a_2 - b_2, \ldots, a_d - b_d) \in \overline{\mathbb{R}}^d$.

- The operator *unary* `-` will perform vector negation, i.e. the operation denoted as $-\vec{a}$ and will represent a vector $(-a_1, -a_2, \ldots, -a_d) \in \overline{\mathbb{R}}^d$.

Substitution of Comparison Operators

The only occurrences of comparison operators in the source code of *DC-PBC* component are interval checks. Similarily to the substitution of interval checks in the *DCM* component, interval checks in the *DC-PBC* component were substituted by calls to the `RealVector.is_inside_bounds` method implementing the hyperrectangle test as defined in Section 2.3.1.

String Conversion Methods

The `RealVector` class will implement two different methods for converting an instance of the class to a string:

- The `__str__()` magic method definition will convert the instance to a string representation suitable for output file formatting,

- The `format_for_dcm_input()` method will convert the instance to a string representation used for DCM input file formatting before the discriminant chronicles mining phase as described in Section 3.3.

4.4.3 DC-PBC Input Parser Changes

Because the main entry point for the whole process of discriminant chronicles mining and subsequent classification will be *DC-PBC*, this project will undergo more changes to its parsing code in order to comply with the goal F2 stated in Section 4.1.

AS-IS Input File Format Change

The original parser located in the `McDataset::load_db_line(...)` method will be altered to accept a file format the same as the file format proposed for the *DCM* component in Section 4.3.2.3.

New File Format

A new file format accepted by DC-PBC will be based on *Matlab CSV export format* (using the `writematrix` function) of the datasets provided by the supervisor of the thesis.

The datasets represented in this format will be split into a total of 3 files. Dataset file structure and file content description is depicted in Figure 4.7.

```
<dataset directory> .......................... the dataset directory
 |__ input.csv ...................... the file containing sequence events
 |__ satisfactory.csv ............... the file containing sequence labels
 |__ temperatures.csv ....... the file containing sequence temporal data
```

Figure 4.7: New file format dataset directory structure

Listing 4.11, 4.12 and 4.13 depict internal formats of particular input files listed in Figure 4.7. Worth noting that values in `input.csv` are not separated by any separator, thus limiting event type names to single-character names.

```
ABCD
BCDA
CDAB
DABC
```

Listing 4.11: `input.csv` data example

```
1
1
0
0
```

Listing 4.12: `satisfactory.csv` data example

```
15.0,15.0,23.0,23.0,29.0,29.0,37.0,37.0
15.0,15.0,16.0,16.0,28.0,28.0,29.0,29.0
15.0,15.0,16.0,16.0,25.0,25.0,26.0,26.0
15.0,15.0,21.0,21.0,22.0,22.0,28.0,28.0
```

Listing 4.13: `temperatures.csv` data example

The datasets represented in this format will be converted to the modified as-is input file format (proposed in Section 4.3.2.3) using a new class `NewFileFormatConverter` implementing the conversion. The converted input files will be placed into the `temp/dataset/` directory located in the project root and both the classification and mining process will continue with the converted representation.

The `NewFileFormatConverter` class will also automatically infer input temporal element vector length for the rest of the process, so the `-s/--vecsize` CLI argument will not need to be specified when using this input file format.

```
A 15.0 15.0 B 23.0 23.0 C 29.0 29.0 D 37.0 37.0
B 15.0 15.0 C 16.0 16.0 D 28.0 28.0 A 29.0 29.0
```

Listing 4.14: positive sequence set after conversion

```
C 15.0 15.0 D 16.0 16.0 A 25.0 25.0 B 26.0 26.0
D 15.0 15.0 A 21.0 21.0 B 22.0 22.0 C 28.0 28.0
```

Listing 4.15: negative sequence set after conversion

New File Format Converter

As discussed in the previous section, the class `NewFileFormatConverter` responsible for parsing and converting the new file format will be placed into a Python module named `new_file_format_converter.py`.

4.4.4 DC-PBC CLI Changes

Analogically to DCM CLI changes stated in Section 4.3.2.2, a new CLI parameter `-s/--vecsize` will be introduced in order to specify domain element vector size, again stored into the `DomainElementFactory.VECTOR_SIZE` static member variable. For temporal element vector size equal to 1 or when using the new file format (see the paragraph below), the parameter will be non-mandatory.

The parameter `-e/--episode` related to discriminant episodes mining will be removed as it is out of the scope of this thesis.

Related to the new file format as discussed in Section 4.4.3, the program will accept a new CLI flag `-l/--legacy` which will use the changed as-is file format for input data instead of the new file format proposed in Section 4.4.3. This file format will be used mainly for regression testing the system on the datasets provided with the implementation by its authors in [6].

Note that for each usage of the legacy file format (i.e. when specifying the `-l/--legacy` CLI flag) where domain element vector has size greater than 1, a CLI parameter `-s/--vecsize` must be defined too.

4.5 Conclusion

This chapter discussed the changes to be implemented into the original version of the system – beginning with stating the high-level description of implementation goals and non-goals, then elaborating particular changes to each component in depth.

Chapter 5

Testing

This chapter will discuss software testing performed on the generalized version of the system for discriminant chronicles mining.

Section 5.1 will briefly introduce the reader to high-level principles of intended testing methodology further elaborated in subsequent sections.

Section 5.2 will state the methods utilized in the testing methodology for testing the system.

Sections 5.3 and 5.4 will elaborate unit tests applied to individual system units – listing performed assertions and introducing additional changes to the code of the system for increasing code testability.

Section 5.5 will elaborate component tests applied to individual system components – discussing test kinds and particular test cases.

Section 5.6 will elaborate integration tests applied to the system as a whole – again, discussing test kinds and particular test cases.

Finally, Section 5.7 will state the results of the testing performed on the system and additional fixes performed to fix the bugs found in the system by testing.

5.1 Testing Methodology

”As software takes on ever more vital functions in life-critical and mission-critical applications and in applications that carry massive financial stakes, it becomes increasingly important to ensure that software products fulfil their function with a high degree of dependability” [15]. Choosing appropriate software testing methodology consisting of complementing software testing techniques will help ensure that the system as a whole is dependable upon.

The original implementation proposed by Yann Dauxais et *al.* available at [5] and [6] did not include any tests as in traditional software testing manners – thus *it is assumed that the implementations of the DCM and DC-PBC projects* *in its original state were tested by its author and are correct* – any assertions concerning backward compatibility and regression testing will be compared to the output of the original implementation.

In order to test the changes implemented in the scope of this thesis, the most affected parts of the system will be tested using the following approaches:

- unit tests will be applied for newly created code units,
- regression and generalization component testing will be applied to individual system components (*DCM*, *DC-PBC*),
- regression and generalization integration testing will be applied to the system as a whole.

All the tests will be automated using scripts in order to allow implementing any changes to the system faster in the future.

5.2 Testing Methods

5.2.1 Unit Testing

Unit tests are short tests initially written by software implementors in the development phase of the software development cycle, performing assertions over software units to comply with given *deterministic specification* [15].

Worth noting, the assertions in unit tests are usually and if possible written to test *solely* the responsibilities of given software unit – i.e. the unit test should test a single class/a single method. Any further integrations and linkings to/with other software units should be disbanded prior to writing the unit tests. Examples of techniques realizing such functional isolation of software units include *mocking*, *stubbing* and *faking*. Utilizing such techniques allows the classes to be tested more rigorously and also makes the tests less fragile when implementing changes in unrelated classes.

5.2.2 Component Testing

Component testing is a type of software testing where individual system *components are tested separately*, i.e. without integrating the components of the system together. Such isolation of software components is usually achieved by the technique called *mocking* – providing files imitating an input from another component to the tested component.

Contrary to unit testing, functional isolation is applied to components instead of individual classes or methods.

5.2.3 Integration Testing

Integration testing tests the system when all of its components are integrated together. Contrary to the deterministic nature of specifications used for unit testing, here the requirements for testing might contradict each other or not even cover the whole spectrum of system functionalities [15].

Integration testing is the final stage of the testing methodology proposed in Section 5.1, thus no functional isolation will be applied to the components in this type of testing.

5.2.4 Regression Testing

According to [15], regression testing takes place in the maintenance phase of the software development cycle to ensure that when software requirements or configuration changes, the system will be tested to check that the changes did not break existing functionalities of the system.

5.3 DCM Unit Testing

Unit testing will be applied to new classes and significantly changed parts of the code of *DCM*:

- The `RealVector` class introduced in Section 4.3.2.1,
- The `DomainElementFactory` class introduced in Section 4.3.2.1,
- The `GeneralRule<T>` and `SingleRule` classes introduced in Section 4.4.3,
- The `extractSingleData`, `run` and `extractMultiRule` functions located in the `ripper` package proposed in Section 4.3.2.4.

The `catch2` C++ unit testing framework available from [16] will be used for defining the unit tests, formulating their assertions and compiling into an executable invoked using a system command line interpreter of choice.

The unit testing assertions for the *DCM* component listed in the following sections will test only public class members and methods. Private and protected members and methods of the classes are not by design of the C++ programming language accessible, thus will be tested by creating stimuli for public members and methods that will test them transitively.

All the unit tests will be placed into the `tests` package in the *DCM* project and can be built independently of the main `DCM` executable.

5.3.1 RealVector Unit Testing

This class is not coupled to any other class, thus is well testable as-is, without applying any specific unit testing technique.

Unit testing assertions for the `RealVector` class are listed in Table 5.1.

Units	Assertions
constructors	It creates a blank `RealVector` when using the parameterless constructor.
`operator+`	It adds two `RealVector`s together correctly.
`operator-`	It subtracts two `RealVector`s correctly.
`isInsideBounds`	It performs a hyperrectangle test correctly. It works correctly for points on hyperrectangle bounds. It works correctly for some coordinates outside of hyperrectangle bounds. It works correctly for all coordinates outside of hyperrectangle bounds. It works correctly for hyperrectangle bounds containing `inf`/`-inf` values.
`operator<<`	It formats a `RealVector` containing integers correctly. It formats `RealVectors` containing `inf`/`-inf` values correctly. It formats a `RealVector` containing floating point numbers correctly. It formats a `RealVector` containing very small floating point numbers using scientific notation.

Table 5.1: `RealVector` class unit testing assertions (DCM)

5.3.2 DomainElementFactory Unit Testing

This factory class is coupled to the `RealVector` class because it produces instances of it. However, assuming that the assertions need to be made over the produced `RealVector` instances themselves, no further mocking or other unit testing technique will be applied in the testing process.

Unit testing assertions for the `DomainElementFactory` class are listed in Table 5.2.

Units	Assertions
`zero`	It constructs a `RealVector` with all zeroes correctly. It reacts to `VECTOR_SIZE` value changes properly.
`positiveInfinity`	It constructs a `RealVector` with all infinity values correctly. It reacts to `VECTOR_SIZE` value changes properly.
`negativeInfinity`	It constructs a `RealVector` with all negative infinity values correctly. It reacts to `VECTOR_SIZE` value changes properly.

Table 5.2: `DomainElementFactory` class unit testing assertions (DCM)

5.3.3 Rule Classes Unit Testing

Both classes are not coupled to any other class, thus are well testable as-are, without applying any specific unit testing technique.

Because the `SingleRule` class extends the already separately tested class `GeneralRule<double>`, while adding two protected members and its corresponding public getters and setters, only the aforementioned getters and setters will be tested.

Unit testing assertions for the classes `GeneralRule<T>` and `SingleRule` are listed in Table 5.3 and Table 5.4.

Units	Assertions
constructor	It constructs an instance of `GeneralRule<int>` correctly.
`getClass`, `getConstraints`, `getValues`, `getFrequency`, `getOFrequency`	All the getters access object properties properly.

Table 5.3: `GeneralRule` class unit testing assertions

Units	Assertions
`setCorrectExamples`, `setIncorrectExamples`	All the setters mutate object properties properly.
`getCorrectExamples`, `getIncorrectExamples`	All the getters access object properties properly.

Table 5.4: `SingleRule` class unit testing assertions

5.3.4 RIPPER Wrapper Procedures Unit Testing

Unit testing assertions for the new procedures of the `Ripper` C++ wrapper namespace are listed in Table 5.5.

Units	Assertions
`indexRuleset`	It stores the rules correctly for the significand set $\{\mathcal{A}_{A\to B}\}$.
	It stores the rules correctly for the significand set $\{\mathcal{A}_{A\to B}, \mathcal{A}_{B\to C}\}$.
	It does not store given rule into the rule map if no rule is present for given run.
`mergeRulesets`	It creates a correct number of resulting rules.
	It assigns the `pos` label to all resulting rules.
	It assigns a distinct significand set for each resulting rule.
	It merges rule values correctly.
	It merges rule values correctly for some run rules missing.
	It computes the count of correctly/incorrectly covered examples correctly.

Table 5.5: Ripper procedures unit testing assertions

5.4 DC-PBC Unit Testing

Unit testing will be applied to all classes newly introduced to the codebase of *DC-PBC*:

- `RealVector` introduced in Section 4.4.2,
- `DomainElementFactory` introduced in Section 4.4.2,
- `NewFileFormatConverter` introduced in Section 4.4.3.

The `unittest` unit testing framework bundled with Python 3 will be used for defining the unit tests, formulating their assertions and running the tests using a CLI.

All the unit tests will be placed into the `tests` directory in the *DC-PBC* project.

5.4.1 RealVector Unit Testing

This model class is not coupled to any other dependency in terms of standard library calls or calls to methods of any other class, so it is testable well as-is.

Unit testing assertions for the `RealVector` class are listed in Table 5.6.

5.4.2 DomainElementFactory Unit Testing

Similarily to its DCM counterpart, no mocking or other unit testing technique will be performed and the assertions will be made over the produced `RealVector` objects.

Unit testing assertions for the `DomainElementFactory` class are listed in Table 5.7.

5.4.3 NewFileFormatConverter Unit Testing

Opposed to the previous classes, this class is tightly coupled to the standard Python library by calling the `os.open` and `os.path.isfile` methods used for working with the file system. Such coupling makes the class difficult to test in performance-oriented environments, so the methods `_open_file`, `_write_output` and `_is_file` substituting direct calls to functions `os.open` and `os.path.isfile` will be introduced to the class.

Along with the `_open_file`, `_write_output` and `_is_file` methods introduced in the previous paragraph, a new optional constructor parameter `mock_input_files` and new instance member variables `_mock_infiles` and `_mock_outfiles` will be introduced to the class in order to allow for input and output file mocking, resulting in simplified unit testing assertions and thus better testability of the `NewFileFormatConverter` class.

Units	Assertions
`__init__`	It constructs a `RealVector` from a tuple correctly.
`_is_in_interval`	It performs closed interval check correctly. It works correctly for edge values. It works correctly for `inf`/`-inf` bound values. It raises an exception when interval bounds are not properly ordered. It does not raise an exception for lower bound equal to upper bound.
`__sub__`	It subtracts two `RealVector`s correctly.
`is_inside_bounds`	It performs a hyperrectangle test correctly. It works correctly for points on hyperrectangle bounds. It works correctly for some coordinates outside of hyperrectangle bounds. It works correctly for all coordinates outside of hyperrectangle bounds. It works correctly for hyperrectangle bounds containing `inf`/`-inf` values.
`__neg__`	It negates all `RealVector` elements correctly.
`_is_tuple`	It correctly marks a tuple literal as tuple. It raises an exception when supplied a dictionary. It raises an exception when supplied an instance of `RealVector`.
`_is_realvector`	It correctly marks a `RealVector` object as `RealVector` instance. It raises an exception when supplied a tuple. It raises an exception when supplied a dictionary.
`_vectors_are_same_length`	It works correctly for two `RealVector`s of the same length. It raises an exception for two `RealVector`s of different length.
`format_for_output`, `__str__`	It formats a `RealVector` containing whole numbers correctly. It formats `RealVector`s containing `inf`/`-inf` values correctly. It formats a `RealVector` containing floating point numbers correctly.

Table 5.6: `RealVector` class unit testing assertions (DC-PBC)

Units	Assertions
`positive_infinity`	It produces a vector of positive infinity values correctly. It reacts to `VECTOR_SIZE` value changes properly.
`negative_infinity`	It produces a vector of negative infinity values correctly. It reacts to `VECTOR_SIZE` value changes properly.
`_tuple_literal_is_correct`	It marks well-formed tuple literals as correct. It raises an exception for malformed tuple literals.
`parse_from_cpp`	It parses tuple literals containing whole numbers correctly. It parses tuple literals containing `inf`/`-inf` values correctly. It parses tuple literals containing floating point numbers correctly.

Table 5.7: `DomainElementFactory` class unit testing assertions (DC-PBC)

Unit testing assertions for the `NewFileFormatConverter` class are listed in Table 5.8.

Units	Assertions
`convert_new_to_old`	It converts positive sequences to old format correctly.
	It converts negative sequences to old format correctly.
	It infers vector size correctly.
	It raises an exception for fluctuating number of items within file lines.
`_check_same_file_length`	It raises an exception when input files have differing line count.
`_check_file_presence`	It raises an exception when some input files are missing in the dataset.
`_parse_input_line`	It parses a line of `input.csv` correctly.
	It parses a line correctly when in a file with LF line endings.
	It parses a line correctly when in a file with CRLF line endings.
	It parses a line with trailing whitespace correctly.
	It does not raise an exception for event types named using metacharacters.
`_parse_satisfactory_line`	It parses a line of `satisfactory.csv` correctly.
	It parses a line correctly when in a file with LF line endings.
	It parses a line correctly when in a file with CRLF line endings.
	It raises an exception for a line not containing the ”0” or ”1” character.
	It raises an exception for a line containing multiple characters.
`_parse_temperatures_line`	It parses a line of `temperatures.csv` correctly.
	It parses a line correctly when in a file with LF line endings.
	It parses a line correctly when in a file with CRLF line endings.
	It raises an exception for a line containing incorrectly formatted number.
`_convert_output_list_to_str`	It serializes a list of arbitrary stringified values correctly.
	It serializes an empty list to empty line.
`_get_input_filename`	It produces correct input file paths for known file types.
	It raises an exception for unknown file type.
`_get_output_filename`	It produces correct output file paths for known sequence labels.
	It raises an exception for unknown sequence label.

Table 5.8: `NewFileFormatConverter` class unit testing assertions

5.5 Component Testing

Unit testing itself as elaborated in Section 5.3 and Section 5.4 will certainly prevent errors in the newly introduced code units, however, will not prove itself sufficient for verifying overall correct functionality of individual system components – *DCM* and *DC-PBC* alone.

Testing both components of the system separately will ensure that each component works correctly on its own *without errors propagating from one component to another.*

Particular approaches used for testing system components separately are elaborated in the following sections.

5.5.1 DCM Component Testing

5.5.1.1 Regression Testing

The testing will be set up as a set of equality assertions between the output of the generalized version of the system and testing data based on the output of the original version of the system while supplying both versions the same arguments and the same dataset. The list of test cases is illustrated in Table 5.9. Note that the $\min(|\mathcal{E}|)$ and $\max(|\mathcal{E}|)$ parameters represent the minimal and maximal chronicle event multiset size and when mining chronicles discriminant for $\mathcal{S}^+$ with respect to $\mathcal{S}^-$, *relative minimal support threshold* $f_{min} \in \langle 0, 1\rangle$ is defined as $f_{min} = \frac{\sigma_{min}}{|\mathcal{S}^+|}$.

Dataset	Arguments
`proportionality`	$f_{min} = 0.25, g_{min} = 2.0, \min(\lvert\mathcal{E}\rvert) = 2, \max(\lvert\mathcal{E}\rvert) = 5$
`ECG`	$f_{min} = 0.5, g_{min} = 5.0, \min(\lvert\mathcal{E}\rvert) = 2, \max(\lvert\mathcal{E}\rvert) = 5$
`BIDE-D/unix`	$f_{min} = 0.5, g_{min} = 2.0, \min(\lvert\mathcal{E}\rvert) = 2, \max(\lvert\mathcal{E}\rvert) = 3$
`BIDE-D/blocks`	$f_{min} = 0.5, g_{min} = 20.0, \min(\lvert\mathcal{E}\rvert) = 5, \max(\lvert\mathcal{E}\rvert) = 5$
`BIDE-D/asl-bu`	$f_{min} = 0.5, g_{min} = 1.0, \min(\lvert\mathcal{E}\rvert) = 6, \max(\lvert\mathcal{E}\rvert) = 8$
`BIDE-D/context`	$f_{min} = 0.9, g_{min} = 7.0, \min(\lvert\mathcal{E}\rvert) = 7, \max(\lvert\mathcal{E}\rvert) = 9$

Table 5.9: DCM component regression test cases

Component Testing Helper Tool

Because the output format of the generalized version differs slightly from the output format of the original version of *DCM*, a tool called `dcm_differ` will be implemented using Python programming language, allowing for automated component testing while respecting the differences between the output of the original and the generalized version of *DCM*.

The main features of `dcm_differ` are:

- equality assertions among output chronicles,
- tolerating minor changes in the output format of the generalized version, namely the mandatory `<` and `>` characters in `RealVector` serialization as specified in Section 4.3.2.1,
- simple generalization testing (tolerating duplicated vector items for generalization testing as specified in Section 5.5.1.2),
- tolerating different order of chronicles in the resulting chronicle output list (introduced by the changes described in 4.3.1.2).

The `dcm_differ` testing tool will accept the following CLI parameters:

- `--old <filename>` (mandatory) will specify the output of the original implementation saved to a file,
- `--new <filename>` (mandatory) will specify the output of the new implementation saved to a file,
- `--check-equivalence` (non-mandatory) will turn on the generalization testing feature as described in the previous bullet list.

5.5.1.2 Generalization Testing

For the needs of testing the generalization code itself, a new dataset called `proportionality_2D` will be derived from the `proportionality` dataset.

Both datasets will share the same event types and counts of positive and negative sequences. The `proportionality_2D` dataset will have vector size equal to 2 and its events will have its two domain vector elements duplicated and equal to the same value as its equivalents in the original dataset.

An example of the generalization step between the `proportionality` and `proportionality_2D` datasets containing *equivalent* sequences is illustrated in Listing 5.1 and Listing 5.2.

```
A 15 B 23 C 29 D 37
A 15 B 16 C 28 D 29
A 15 B 16 C 25 D 26
A 15 C 21 B 22 D 28
```

Listing 5.1: `proportionality` dataset example

```
A 15 15 B 23 23 C 29 29 D 37 37
A 15 15 B 16 16 C 28 28 D 29 29
A 15 15 B 16 16 C 25 25 D 26 26
A 15 15 C 21 21 B 22 22 D 28 28
```

Listing 5.2: Equivalent `proportionality_2D` dataset example

With such changes to the input dataset, the resulting chronicles are expected to have *the same multisets, the same support and equivalent hyperrectangle constraints* (equivalent in terms of duplicated vector elements) compared to the original output. The test cases used for testing the generalization are listed in Table 5.10.

The `dcm_differ` testing tool introduced in Section 5.5.1.1 will be used for constructing equivalence assertions for this kind of tests.

Datasets	Arguments
`proportionality`, `proportionality_2D`	$f_{min} = 0.25, g_{min} = 2.0, \min(\lvert\mathcal{E}\rvert) = 2, \max(\lvert\mathcal{E}\rvert) = 5$
`proportionality`, `proportionality_2D`	$f_{min} = 0.7, g_{min} = 2.0, \min(\lvert\mathcal{E}\rvert) = 3, \max(\lvert\mathcal{E}\rvert) = 6$
`proportionality`, `proportionality_2D`	$f_{min} = 0.1, g_{min} = 30.0, \min(\lvert\mathcal{E}\rvert) = 1, \max(\lvert\mathcal{E}\rvert) = 3$

Table 5.10: DCM component generalization test cases

5.5.2 DC-PBC Component Testing

Because the *DC-PBC* component requires input from the *DCM* component, input file mocking functionality will be implemented into the *DC-PBC* component as described in Section 5.2.2. Schematics depicting the *DC-PBC* component in original mode and input mocking mode are illustrated in Figure 5.1 and 5.2, respectively.

A new parameter `--component-debug-in <fold_1_neg>,<fold_1_pos>, ...,<fold_k_neg>,<fold_k_pos>` for *DC-PBC* CLI will implement the *DCM* input mocking mode. Each file represents a single mock input from the *DCM* component for each fold and each label (`pos`/`neg`). The number of folds for a given test run will be inferred automatically as `k`. Thus the number of mock files must be even and the mock files must be supplied to the parameter in exact order.

The new `--disable-randomness` CLI parameter will, when present, make all stochastic elements in the component deterministic – all random number generators will produce a single fixed value and dataset shuffling in dataset fold splitting phase will not be executed so the testability of the component will be deterministic between individual test runs.

To simplify performing assertions over *DC-PBC* outputs, the new non-mandatory `--out <dir>` CLI argument will be implemented to specify the output directory for *DC-PBC* results containing mining and classification results.

Note that, as described in Section 3.3, the *DC-PBC* component is also integrated with third-party *WEKA* component while utilizing it in the classification process. The *WEKA* component will not be mocked while testing the *DC-PBC* component because it has the same responsibility as the internal *sklearn SVC* classifier, essentially being part of the *DC-PBC* component.

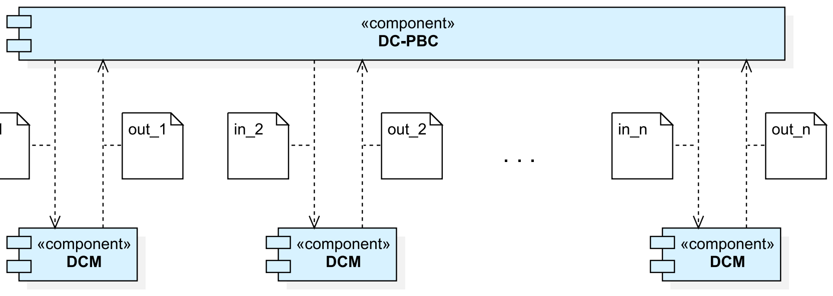

Figure 5.1: DC-PBC and DCM component integration schema

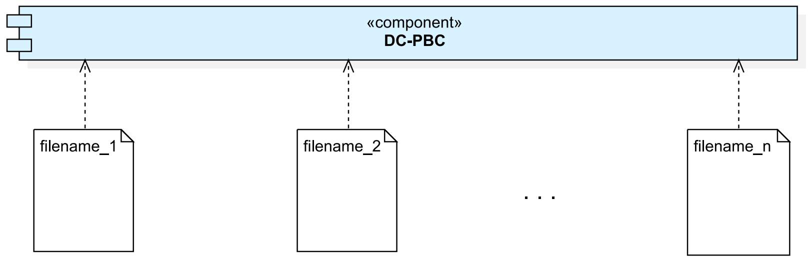

Figure 5.2: DC-PBC in DCM input mocking mode

5.5.2.1 Testing Data

The data for *DC-PBC* component testing will be prepared by modifying the original implementation of *DC-PBC* so that it will reveal its *DCM* component inputs. The patch for the original implementation used to generate mock files is listed in Appendix B.

Modified DCM mock inputs (with `<` and `>` added to `RealVector` serialization), CLI parameters and accordingly modified DC-PBC outputs will be grouped into test cases and prepared for automatic execution. The standard Unix `diff` tool will be used for *DC-PBC* output assertions.

5.5.2.2 Regression Testing

Both *DC-PBC* implementations – the original implementation and the generalized implementation will be supplied the `proportionality` dataset supplied with the original implementation. Particular test cases for regression testing of the *DC-PBC* component are illustrated in Table 5.11. Note that the n parameter represents a number limiting the total count of resulting chronicles per class and the k parameter represents growth rate bias, both elaborated in the `README` file bundled with the original implementation.

Datasets	Arguments
`proportionality`	$f_{min} = 0.25, g_{min} = 2.0, \min(\lvert\mathcal{E}\rvert) = 2, \max(\lvert\mathcal{E}\rvert) = 5,$ $\#\text{folds} = 1, \text{classifier} = \text{SVC}, n = 90, k = 1$
`proportionality`	$f_{min} = 0.4, g_{min} = 20.0, \min(\lvert\mathcal{E}\rvert) = 1, \max(\lvert\mathcal{E}\rvert) = 3,$ $\#\text{folds} = 1, \text{classifier} = \text{none}, n = 90, k = 1$
`proportionality`	$f_{min} = 0.25, g_{min} = 2.0, \min(\lvert\mathcal{E}\rvert) = 2, \max(\lvert\mathcal{E}\rvert) = 5,$ $\#\text{folds} = 1, \text{classifier} = \text{none}, n = 90, k = 0$
`proportionality`	$f_{min} = 0.3, g_{min} = 4.0, \min(\lvert\mathcal{E}\rvert) = 1, \max(\lvert\mathcal{E}\rvert) = 3,$ $\#\text{folds} = 5, \text{classifier} = \text{J48}, n = 90, k = 1$

Table 5.11: DC-PBC component regression test cases

5.5.2.3 Generalization Testing

In addition to the steps described in Section 5.5.2.1, the original *DCM* input mock data and *DC-PBC* output data from a task with `proportionality` dataset as input will be extended to two vector elements with duplicated values. The input dataset for the generalized component will be the new dataset `proportionality_2D`. Particular test cases for generalization testing of the DC-PBC component are illustrated in Table 5.12.

Datasets	Arguments
`proportionality`, `proportionality_2D`	$f_{min} = 0.4, g_{min} = 20.0, \min(\lvert\mathcal{E}\rvert) = 1, \max(\lvert\mathcal{E}\rvert) = 3$ $\#\text{folds} = 1, \text{classifier} = \text{SVC}, n = 90, k = 1$
`proportionality`, `proportionality_2D`	$f_{min} = 0.4, g_{min} = 20.0, \min(\lvert\mathcal{E}\rvert) = 1, \max(\lvert\mathcal{E}\rvert) = 3$ $\#\text{folds} = 1, \text{classifier} = \text{none}, n = 90, k = 5$
`proportionality`, `proportionality_2D`	$f_{min} = 0.25, g_{min} = 2.0, \min(\lvert\mathcal{E}\rvert) = 1, \max(\lvert\mathcal{E}\rvert) = 3$ $\#\text{folds} = 2, \text{classifier} = \text{none}, n = 5, k = 1$

Table 5.12: DC-PBC component generalization test cases

5.6 Integration Testing

Testing the system as a whole will share test cases with the DC-PBC component, except that *DCM* input mocking using the `--component-debug-in` CLI argument will be disabled and real *DCM* component will be used for discriminant chronicles mining instead.

5.6.1 Regression Testing

Particular test cases for regression testing of the system are illustrated in Table 5.11.

5.6.2 Generalization Testing

Particular test cases for generalization testing of the system are illustrated in Table 5.12.

5.7 Testing Results

5.7.1 Unit Testing

The unit tests were written ex-post the implementation was completed – meaning the author did not follow *TDD (Test-Driven Development)* or *BDD (Behavior-Driven Development)* development methodologies utilizing tests in the implementation phase of the development cycle.

The assertions themselves as seen in Section 5.3 and Section 5.4 were constructed to partly reflect the examples illustrated in Chapter 4, capture important corner cases and test basic expected functionality.

After running the completed suite of unit tests, the testing uncovered some minor bugs – those bugs were fixed trivially afterwards, making the unit testing efforts profitable.

5.7.2 Component Testing

After successful implementation of the changes for each component as specified in Chapter 4 and after running unit tests for each of the components, component testing took place.

The component testing itself proved worth the effort because it uncovered some more critical, hard-to-find bugs which could not be covered by the unit tests directly.

The aforementioned bugs were removed using programming in the debugger and using internal data structure assertions in runtime between specified application breakpoints.

5.7.3 Integration Testing

Last but not least, integration testing was performed as the last of the three, testing the system as a whole to ensure it is dependable upon for both the mining and classification tasks. All the bugs found in the previous testing stages were fixed prior to executing the integration tests.

Thanks to the efforts made in the unit and component testing stages, the integration testing stage did not uncover any new bugs, signifying that the implementation did neither break the functionality of the original system nor the generalization worked incorrectly.

5.8 Conclusion

This chapter introduced a testing methodology utilizing various methods of software testing for verifying and validating the changes introduced to the system. The testing methodology proposed was later elaborated in subsequent sections. The last part of this chapter evaluated the results of the testing process.

Chapter 6

Proof of Concept

The last chapter of this thesis will demonstrate the abilities of this system on a dataset containing multi-dimensional input data.

Section 6.1 will describe the dataset used to demonstrate the functionality of the system.

Section 6.2 will impose qualitative and quantitative criteria on the resulting set of chronicles.

Section 6.3 will propose a new metric for determining the qualitative properties of chronicles.

Section 6.4 will introduce a set of metrics which will be used for mining parameter tuning in subsequent sections.

Section 6.5 will propose a tool which will generate parameter tuning metrics tables and extract chronicles with several qualitative properties.

Section 6.6 will describe the mining parameter tuning process and its outcomes.

Section 6.7 will list the resulting chronicles of the mining process.

Finally, Section 6.8 will discuss the characteristics of the resulting chronicles in the context of the input dataset.

6.1 Dataset

The dataset is based on real-world data introduced in [7], containing multi-dimensional inputs consisting of temperatures recorded by five sensors during a crystal growth process (i.e. $\mathbb{T} \subset \overline{\mathbb{R}}^5$) in a furnace and several event types representing clusters of two-dimensional vectors recording the power of two heaters inside the furnace.

The dataset consists of two sequence sets:

- $\mathcal{S}^+$, the positive sequence set, containing a total of 90 sequences in which the position of the solid/liquid interface is greater than 2500,
- $\mathcal{S}^-$, the negative sequence set, containing a total of 165 sequences in which the position of the solid/liquid interface is lower than 2500.

All the sequences in both sequence sets contain 20 events.

The dataset itself is encoded in the new *Matlab Export* file format as elaborated in Section 4.4.3.

6.2 Output Criteria

The resulting chronicle set $\mathbb{C}$ should contain about 20-30 elements and also should contain both chronicles discriminant for $\mathcal{S}^+$ with respect to $\mathcal{S}^-$ and chronicles discriminant for $\mathcal{S}^-$ with respect to $\mathcal{S}^+$.

Each chronicle $(\mathcal{E}, \mathcal{T}) \in \mathbb{C}$ should contain only a minimal number of hyperrectangle constraints of type $e[\![(-\infty, -\infty, -\infty, -\infty, -\infty), (\infty, \infty, \infty, \infty, \infty)]\!]e'$ where $e, e' \in \mathcal{T}$.

6.3 Chronicle Specificity

Assume that $\mathcal{C} = (\mathcal{E}, \mathcal{T})$ is a chronicle, $\mathbb{C}$ is a set of chronicles and $t_s \in \langle 0, 1\rangle$.

Chronicle specificity denoted as $\mathrm{s}(\mathcal{C})$ is defined as:

$$\mathrm{s}(\mathcal{C}) = \frac{|\{e[\![t, t']\!]e' | \{e[\![t, t']\!]e' \in \mathcal{T} \wedge [e \neq (-\infty, \dots, -\infty) \vee e' \neq (\infty, \dots, \infty)]\}|}{|\mathcal{T}|}.$$

Chronicle set $\mathbb{C}$ specific for a specificity threshold t_s denoted as $\mathrm{s}(\mathbb{C}, t_s)$ is defined as:

$$\mathrm{s}(\mathbb{C}, t_s) = \{\mathcal{C} | \mathcal{C} \in \mathbb{C} \wedge \mathrm{s}(\mathcal{C}) \geq t_s\}.$$

6.4 Metrics For Parameter Tuning

The metrics used for evaluating the convenience of parameters passed to the *DC-PBC* component are:

- $|\mathbb{M}|$ is the size of frequent multisets set as introduced in Section 1.3.1,

- $|\mathbb{E}|$ is the count of distinct frequent multisets which occurred in some discriminant chronicle of the resulting chronicle set $\mathbb{C}$ – i.e. $|\mathbb{E}| = |\{\mathcal{E}|(\mathcal{E},\mathcal{T}) \in \mathbb{C}\}|$,
- $\text{maxs}(\mathbb{C}) = \max\{\text{s}(\mathcal{C})|\mathcal{C} \in \mathbb{C}\}$ is the maximal specificity value found among the chronicles in $\mathbb{C}$,
- $|\text{s}(\mathbb{C}, t_s)|$ is the count of chronicles specific for t_s found in given resulting chronicle set $\mathbb{C}$.

6.5 Parameter Tuning Tools

For the needs of generating CSV files containing tables with parameter tuning metric values, a tool `chronicle_statgen` was developed in Python programming language while also allowing to extract specific chronicles given a specificity threshold t_s.

The CLI parameters of the `chronicle_statgen` tool are:

- `--chrofile` (mandatory) specifies the file containing all the chronicles in the output format of the *DC-PBC* component,
- `--logfile` (mandatory) specifies the file containing the log of the *DC-PBC* component in verbose mode using its `--verbose` CLI flag,
- `--vecsize` (mandatory) specifies the size of vectors in the output file of *DC-PBC*,
- `--category` (non-mandatory) is a parameter used for generating output table row captions,
- `--minspec` (non-mandatory) specifies the specificity threshold used for extracting specific chronicles set from the output file.

6.6 Parameters Tuning

The *DC-PBC* component will be used for mining the resulting set of chronicles as it produces both chronicles discriminant for $\mathcal{S}^+$ with respect to $\mathcal{S}^-$ and chronicles discriminant for $\mathcal{S}^-$ with respect to $\mathcal{S}^+$ as opposed to *DCM* which produces chronicles discriminant only for $\mathcal{S}^+$ with respect to $\mathcal{S}^-$.

The classification rules generation capabilities of the *DC-PBC* component will not be used in this scenario, thus only the parameters affecting the mining process itself will be tuned:

- $f_{min} \in \langle 0, 1\rangle$ implemented by the `--fmin` parameter represents *relative minimal support threshold* related to the *minimal support threshold* parameter of the `DCM` algorithm further elaborated in Section 1.3 as follows:

  - Assume $\mathcal{S}' \cup \mathcal{S}'' = \mathcal{S} \wedge \mathcal{S}' \cap \mathcal{S}'' = \emptyset$, $\sigma_{min} \in \mathbb{N}$. When mining chronicles discriminant for $\mathcal{S}'$ with respect to $\mathcal{S}''$, $f_{min} = \frac{\sigma_{min}}{|\mathcal{S}'|}$,

- $g_{min} \in \langle 1, \infty)$ implemented by the `--gmin` parameter represents the minimal growth rate threshold parameter of the `DCM` algorithm further elaborated in Section 1.3,
- $\min(|\mathcal{E}|) \in \mathbb{N}$ implemented by the `--mincs` parameter represents minimal chronicle event multiset size,
- $\max(|\mathcal{E}|) \in \mathbb{N}$ implemented by the `--maxcs` parameter represents maximal chronicle event multiset size.

In addition to the parameters stated in the previous paragraph, the specificity threshold f_s will be tuned for the `chronicle_statgen` tool in order to extract only a certain number of chronicles satisfying given specificity threshold value. The process of discriminant specific chronicles mining is depicted in Figure 6.1.

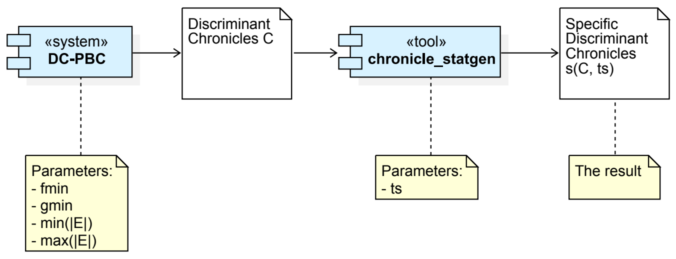

Figure 6.1: Resulting chronicle set mining

6.6.1 Chronicle Event Multiset Size

With f_{min}, g_{min} and t_s set to fixed values – $f_{min} = 0.2$, $g_{min} = 5000$ and $f_s = 0.3$, the $\min(|\mathcal{E}|)$ and $\max(|\mathcal{E}|)$ parameters were tuned while $\min(|\mathcal{E}|)$ was kept equal to $\max(|\mathcal{E}|)$.

After evaluating the $|\mathrm{s}(\mathbb{C}, 0.3)|$ metric in Table 6.1, the chronicle event multiset size was limited by choosing $\min(|\mathcal{E}|) = 2$ and $\max(|\mathcal{E}|) = 5$.

$\min(\vert\mathcal{E}\vert), \max(\vert\mathcal{E}\vert)$	$\vert\mathbb{M}\vert$	$\vert\mathbb{E}\vert$	$\max\mathrm{s}(\mathbb{C})$	$\vert\mathrm{s}(\mathbb{C}, 0.3)\vert$
2	86	81	1.000	10
3	242	238	1.000	36
4	545	541	0.500	18
5	1030	1025	0.300	1
6	1670	1666	0.133	0
7	2369	2366	0.048	0
8	2968	2967	0.036	0
9	3293	3293	0.028	0
10	3230	3230	0.000	0

Table 6.1: Tuning metrics for parameters $\min(|\mathcal{E}|)$ and $\max(|\mathcal{E}|)$, rouded to 3 decimal places

6.6.2 Minimal Support Threshold and Specificity Threshold

With $g_{min} = 5000$, $\min(|\mathcal{E}|) = 2$ and $\max(|\mathcal{E}|) = 5$, the parameters f_{min} with respect to f_s were tuned simultaneously – lower f_{min} values result in a higher number of chronicles and specific chronicles while the count of specific chronicles in the resulting chronicle set should be kept at about 20-30 as specified in Section 6.2.

f_{min}	$\vert\mathbb{M}\vert$	$\vert\mathbb{E}\vert$	$\max\mathrm{s}(\mathbb{C})$	$\vert\mathrm{s}(\mathbb{C}, 0.3)\vert$
0.1	3067	3049	1.0	208
0.3	1220	1216	1.0	21
0.5	558	554	1.0	6
0.7	292	291	1.0	2
0.9	145	144	1.0	2
0.95	79	79	1.0	1
0.96	60	60	1.0	1
0.97	60	60	1.0	1
0.98	60	60	1.0	1
0.99	52	52	1.0	1

Table 6.2: Tuning metrics for parameter f_{min} with $t_s = 0.3$

The parameter values that were chosen based on data in Table 6.2, 6.3 and 6.4 are $f_{min} = 0.1$ and $t_s = 0.7$, mainly because the count of specific chronicles is adequate to what was stated as required in Section 6.2.

6.6.3 Minimal Growth Threshold

With $f_{min} = 0.1$, $t_s = 0.7$, $\min(|\mathcal{E}|) = 2$ and $\max(|\mathcal{E}|) = 5$ fixed values, the g_{min} parameter was tuned while checking if the $|\mathrm{s}(\mathbb{C}, 0.7)|$ metric would raise.

f_{min}	$\|\mathbb{M}\|$	$\|\mathbb{E}\|$	maxs($\mathbb{C}$)	$\|\text{s}(\mathbb{C}, 0.5)\|$
0.1	3067	3049	1.0	84
0.3	1220	1216	1.0	7
0.5	558	554	1.0	3
0.7	292	291	1.0	1
0.9	145	144	1.0	1
0.95	79	79	1.0	1
0.96	60	60	1.0	1
0.97	60	60	1.0	1
0.98	60	60	1.0	1
0.99	52	52	1.0	1

Table 6.3: Tuning metrics for parameter f_{min} with $t_s = 0.5$

f_{min}	$\|\mathbb{M}\|$	$\|\mathbb{E}\|$	maxs($\mathbb{C}$)	$\|\text{s}(\mathbb{C}, 0.7)\|$
0.1	3067	3049	1.0	26
0.3	1220	1216	1.0	6
0.5	558	554	1.0	2
0.7	292	291	1.0	1
0.9	145	144	1.0	1
0.95	79	79	1.0	1
0.96	60	60	1.0	1
0.97	60	60	1.0	1
0.98	60	60	1.0	1
0.99	52	52	1.0	1

Table 6.4: Tuning metrics for parameter f_{min} with $t_s = 0.7$

g_{min}	$\|\mathbb{M}\|$	$\|\mathbb{E}\|$	maxs($\mathbb{C}$)	$\|\text{s}(\mathbb{C}, 0.7)\|$
1	3067	3054	1.0	12
5	3067	3049	1.0	22
50	3067	3049	1.0	25
500	3067	3049	1.0	26
5000	3067	3049	1.0	26
50000	3067	3049	1.0	26

Table 6.5: Tuning metrics for parameter g_{min}

The effect of g_{min} on the size of specific resulting chronicle set proved to be compliant with the assumption stated in the previous paragraph. The initial g_{min} value was sufficient for yielding an optimal amount of specific chronicles for $t_s = 0.7$, thus the value was left as-is – i.e. $g_{min} = 5000$.

6.7 Obtained Results

Based on the observations from Section 6.6, the system was invoked using the *DC-PBC* component entry point with the following set of arguments:

- `--mincs 2`,
- `--maxcs 5`,
- `--fmin 0.1`,
- `--gmin 5000`.

The `chronicle_statgen` tool was called subsequently with the argument `--minspec 0.7`.

The system produced a total of 26 specific discriminant chronicles – 18 of them were discriminant for $\mathcal{S}^+$ with respect to $\mathcal{S}^-$, while 8 of them were discriminant for $\mathcal{S}^-$ with respect to $\mathcal{S}^+$. Those chronicles are listed in Table 6.6 and 6.7.

Note that for a chronicle $\mathcal{C} = (\mathcal{E}, \mathcal{T})$, the table row caption E($\mathcal{C}$) represents the event type multiset $\mathcal{E}$ and the table row caption T($\mathcal{C}$) represents the temporal constraint set of the chronicle.

6.8 Results Interpretation

One very interesting aspect shared across all the resulting chronicles is that each chronicle $\mathcal{C}^+$ discriminant for $\mathcal{S}^+$ with respect to $\mathcal{S}^-$ has its support value $\text{supp}(\mathcal{C}^+, \mathcal{S}^-) = 0$ and each chronicle $\mathcal{C}^-$ discriminant for $\mathcal{S}^-$ with respect to $\mathcal{S}^+$ has $\text{supp}(\mathcal{C}^-, \mathcal{S}^+) = 0$, signifying that the events in the sequence sets $\mathcal{S}^+$ and $\mathcal{S}^-$ are different to a large extent.

6.9 Conclusion

The last chapter of this thesis described the input dataset and the important criteria on the resulting set of discriminant chronicles. Next, a set of metrics used for mining parameters tuning along with a new metric called *specificity* were proposed along with a tool for evaluating the chronicle set specificity and extracting specific chronicles. Afterwards, multi-step parameter tuning was performed in order to obtain a result satisfying the criteria. Lastly, the results of the mining were listed in two tables and discussed in the context of the input dataset.

E($\mathcal{C}$)	T($\mathcal{C}$)	supp($\mathcal{C}, \mathcal{S}^+$)	supp($\mathcal{C}, \mathcal{S}^-$)
$\{\!\{e_1 = I, e_2 = Q\}\!\}$	$\{e_1 \llbracket (68.6, 68.8, 68.9, 69.1, 69.2), (69.0, 69.1, 69.3, 69.4, 69.5) \rrbracket e_2\}$	7	0
$\{\!\{e_1 = G, e_2 = G\}\!\}$	$\{e_1 \llbracket (-124, -125, -125, -126, -127), (-122, -123, -123, -124, -124) \rrbracket e_2\}$	8	0
$\{\!\{e_1 = G, e_2 = A, e_3 = Q\}\!\}$	$\{e_1 \llbracket (139, 140, 141, 142, 142), (\infty, \infty, \infty, \infty, \infty) \rrbracket e_2,$ $e_1 \llbracket (-\infty, -\infty, -\infty, -\infty, -\infty), (249, 268, 287, 306, 325) \rrbracket e_3,$ $e_2 \llbracket (43.5, 61.3, 73.7, 73.8, 74.0), (\infty, \infty, \infty, \infty, \infty) \rrbracket e_3\}$	9	0
$\{\!\{e_1 = A, e_2 = I, e_3 = Q\}\!\}$	$\{e_1 \llbracket (-\infty, -\infty, -\infty, -\infty, -\infty), (15.5, 17.9, 18.0, 18.1, 18.2) \rrbracket e_2,$ $e_1 \llbracket (43.1, 60.6, 73.7, 73.8, 74.0), (\infty, \infty, \infty, \infty, \infty) \rrbracket e_3,$ $e_2 \llbracket (-\infty, -\infty, -\infty, -\infty, -\infty), (144, 145, 146, 146, 163) \rrbracket e_3\}$	10	0
$\{\!\{e_1 = A, e_2 = I\}\!\}$	$\{e_1 \llbracket (431, 451, 470, 490, 510), (\infty, \infty, \infty, \infty, \infty) \rrbracket e_2\}$	12	0
$\{\!\{e_1 = I, e_2 = K\}\!\}$	$\{e_1 \llbracket (72.5, 72.7, 72.8, 73.0, 73.1), (137, 144, 162, 180, 198) \rrbracket e_2\}$	12	0
$\{\!\{e_1 = G, e_2 = A, e_3 = I\}\!\}$	$\{e_1 \llbracket (-\infty, -\infty, -\infty, -\infty, -\infty), (-128, -128, -129, -130, -130) \rrbracket e_2,$ $e_1 \llbracket (55.9, 73.8, 91.6, 109, 127), (\infty, \infty, \infty, \infty, \infty) \rrbracket e_3,$ $e_2 \llbracket (398, 412, 414, 416, 418), (\infty, \infty, \infty, \infty, \infty) \rrbracket e_3\}$	13	0
$\{\!\{e_1 = G, e_2 = A, e_3 = I\}\!\}$	$\{e_1 \llbracket (-\infty, -\infty, -\infty, -\infty, -\infty), (-374, -376, -378, -380, -382) \rrbracket e_2,$ $e_1 \llbracket (-\infty, -\infty, -\infty, -\infty, -\infty), (88.6, 107, 125, 144, 162) \rrbracket e_3,$ $e_2 \llbracket (398, 412, 414, 416, 418), (\infty, \infty, \infty, \infty, \infty) \rrbracket e_3\}$	13	0
$\{\!\{e_1 = A, e_2 = I, e_3 = Q\}\!\}$	$\{e_1 \llbracket (-\infty, -\infty, -\infty, -\infty, -\infty), (36.9, 37.0, 37.1, 37.1, 37.2) \rrbracket e_2,$ $e_1 \llbracket (43.1, 60.6, 73.7, 73.8, 74.0), (\infty, \infty, \infty, \infty, \infty) \rrbracket e_3,$ $e_2 \llbracket (-\infty, -\infty, -\infty, -\infty, -\infty), (144, 145, 146, 146, 163) \rrbracket e_3\}$	13	0
$\{\!\{e_1 = G, e_2 = Q, e_3 = Q\}\!\}$	$\{e_1 \llbracket (-\infty, -\infty, -\infty, -\infty, -\infty), (249, 267, 286, 305, 324) \rrbracket e_2,$ $e_1 \llbracket (60.4, 78.3, 96.3, 114, 132), (\infty, \infty, \infty, \infty, \infty) \rrbracket e_3,$ $e_2 \llbracket (-32.5, -32.5, -32.6, -32.7, -32.7), (-31.8, -31.8, -31.9, -32.0, -32.0) \rrbracket e_3\}$	14	0
$\{\!\{e_1 = G, e_2 = A\}\!\}$	$\{e_1 \llbracket (-\infty, -\infty, -\infty, -\infty, -\infty), (-375, -377, -379, -381, -383) \rrbracket e_2\}$	16	0
$\{\!\{e_1 = G, e_2 = K\}\!\}$	$\{e_1 \llbracket (26.5, 43.7, 44.6, 44.8, 45.1), (161, 180, 198, 217, 236) \rrbracket e_2\}$	17	0
$\{\!\{e_1 = G, e_2 = I, e_3 = Q\}\!\}$	$\{e_1 \llbracket (104, 117, 134, 151, 163), (\infty, \infty, \infty, \infty, \infty) \rrbracket e_2,$ $e_1 \llbracket (-\infty, -\infty, -\infty, -\infty, -\infty), (189, 205, 222, 239, 256) \rrbracket e_3,$ $e_2 \llbracket (-60.4, -60.5, -60.6, -60.7, -60.8), (\infty, \infty, \infty, \infty, \infty) \rrbracket e_3\}$	18	0
$\{\!\{e_1 = A, e_2 = Q\}\!\}$	$\{e_1 \llbracket (372, 391, 411, 430, 449), (\infty, \infty, \infty, \infty, \infty) \rrbracket e_2\}$	21	0
$\{\!\{e_1 = A, e_2 = A, e_3 = Q\}\!\}$	$\{e_1 \llbracket (-\infty, -\infty, -\infty, -\infty, -\infty), (-33.7, -51.3, -68.9, -86.7, -105) \rrbracket e_2,$ $e_1 \llbracket (-\infty, -\infty, -\infty, -\infty, -\infty), (73.4, 73.5, 73.7, 73.8, 74.0) \rrbracket e_3,$ $e_2 \llbracket (-\infty, -\infty, -\infty, -\infty, -\infty), (159, 160, 161, 162, 180) \rrbracket e_3\}$	23	0
$\{\!\{e_1 = A, e_2 = K\}\!\}$	$\{e_1 \llbracket (286, 305, 324, 343, 362), (\infty, \infty, \infty, \infty, \infty) \rrbracket e_2\}$	30	0
$\{\!\{e_1 = A, e_2 = A\}\!\}$	$\{e_1 \llbracket (-135, -135, -136, -137, -137), (-127, -128, -128, -129, -130) \rrbracket e_2\}$	38	0
$\{\!\{e_1 = Q, e_2 = K\}\!\}$	$\{e_1 \llbracket (-27.3, -27.4, -27.4, -27.5, -27.5), (\infty, \infty, \infty, \infty, \infty) \rrbracket e_2\}$	72	0

Table 6.6: Resulting set of chronicles discriminant for $\mathcal{S}^+$ with respect to $\mathcal{S}^-$, sorted by supp($\mathcal{C}, \mathcal{S}^+$), rouded to 3 significant figures

$\mathrm{E}(\mathcal{C})$	$\mathrm{T}(\mathcal{C})$	$\mathrm{supp}(\mathcal{C}, \mathcal{S}^+)$	$\mathrm{supp}(\mathcal{C}, \mathcal{S}^-)$
$\{\!\!\{e_1 = A, e_2 = Q\}\!\!\}$	$\{e_1\llbracket(12.1, 12.1, 12.1, 12.1, 12.1), (13.6, 13.6, 13.6, 13.6, 13.7)\rrbracket e_2\}$	0	13
$\{\!\!\{e_1 = Q, e_2 = Q\}\!\!\}$	$\{e_1\llbracket(-30.8, -30.8, -30.9, -31.0, -31.0), (-30.1, -30.2, -30.2, -30.3, -30.3)\rrbracket e_2\}$	0	14
$\{\!\!\{e_1 = G, e_2 = A\}\!\!\}$	$\{e_1\llbracket(316, 333, 351, 368, 385), (\infty, \infty, \infty, \infty, \infty)\rrbracket e_2\}$	0	16
$\{\!\!\{e_1 = G, e_2 = Q\}\!\!\}$	$\{e_1\llbracket(393, 411, 429, 447, 465), (\infty, \infty, \infty, \infty, \infty)\rrbracket e_2\}$	0	18
$\{\!\!\{e_1 = G, e_2 = I, e_3 = Q\}\!\!\}$	$\{e_1\llbracket(-\infty, -\infty, -\infty, -\infty, -\infty), (80.3, 80.5, 80.6, 80.7, 80.9)\rrbracket e_2,$ $e_1\llbracket(-\infty, -\infty, -\infty, -\infty, -\infty), (50.2, 50.2, 50.3, 50.4, 50.5)\rrbracket e_3,$ $e_2\llbracket(-\infty, -\infty, -\infty, -\infty, -\infty), (67.4, 67.5, 67.6, 77.0, 93.3)\rrbracket e_3\}$	0	19
$\{\!\!\{e_1 = G, e_2 = I\}\!\!\}$	$\{e_1\llbracket(318, 334, 351, 367, 384), (\infty, \infty, \infty, \infty, \infty)\rrbracket e_2\}$	0	31
$\{\!\!\{e_1 = I, e_2 = I\}\!\!\}$	$\{e_1\llbracket(-30.9, -31.0, -31.0, -31.1, -31.1), (\infty, \infty, \infty, \infty, \infty)\rrbracket e_2\}$	0	37
$\{\!\!\{e_1 = G, e_2 = G\}\!\!\}$	$\{e_1\llbracket(-121, -122, -122, -123, -123), (\infty, \infty, \infty, \infty, \infty)\rrbracket e_2\}$	0	39

Table 6.7: Resulting set of chronicles discriminant for $\mathcal{S}^-$ with respect to $\mathcal{S}^+$, sorted by $\mathrm{supp}(\mathcal{C}, \mathcal{S}^-)$, rouded to 3 significant figures

Conclusion

All the goals of this thesis were fulfiled:

Chapter 1 introduced the reader to the theoretical basis for discriminant chronicles mining and presented an implementation of an algorithm used for discriminant chronicles mining.

Chapter 2 proposed changes to the original theoretical basis along with new concepts needed for generalizing discriminant chronicles mining for multi-dimensional data.

Chapter 3 elaborated the original implementation – its technological aspects, modular structure and technical issues.

Chapter 4 proposed changes to be made to the original implementation in order to generalize the system for multi-dimensional data.

Chapter 5 proposed a multi-modal testing methodology which was later executed and helped to fix the bugs in the system after implementing the changes proposed in the previous chapter.

Chapter 6 applied the modified system to multi-dimensional input data concerning crystal growth and interpreted the discriminant chronicles from the resulting set.

Future Development

The modified version of the system is, thanks to its refined modular structure and automatic tests, easy to quickly and reliably extend or modify further for any other type of input data – possibly supporting a more diverse set of use-cases where discriminant chronicles mining might be a useful data mining technique.

Bibliography

[1] Dauxais, Y.; Guyet, T.; et al. Discriminant chronicles mining. In *Artificial Intelligence in Medicine*, edited by A. Teije; C. Popow; J. Holmes; L. Sacchi, Cham: Springer, 2017, ISBN 978-3-319-59758-4, doi:10.1007/978-3-319-59758-4_26. Available from: `https://www.springer.com/us/book/9783319065045`

[2] Wright, A. P.; Wright, A. T.; et al. The use of sequential pattern mining to predict next prescribed medications. In *Journal of Biomedical Informatics Volume 53*, San Diego, CA, United States: Elsevier, 2015, doi:10.5555/2953211.

[3] Fradkin, D.; Mörchen, F. Mining sequential patterns for classification. In *Knowledge and Information Systems Volume 45*, Springer, 2017, doi:10.1007/s10115-014-0817-0.

[4] Cohen, W. Fast Effective Rule Induction. In *ICML'95: Proceedings of the Twelfth International Conference*, 1995.

[5] *DAUXAIS Yann / DCM at Inria Gitlab [online]*. 2020, [cit. 2020-02-08]. Available from: `https://gitlab.inria.fr/ydauxais/DCM`

[6] *DAUXAIS Yann / GDC-PBC at Inria Gitlab [online]*. 2020, [cit. 2020-02-08]. Available from: `https://gitlab.inria.fr/ydauxais/GDC-PBC`

[7] Dropka, N.; Holena, M.; et al. Fast forecasting of VGF crystal growth process by dynamic neural networks. In *Journal of Crystal Growth*, San Diego, CA, United States: Elsevier, 2019, doi:10.1016/j.jcrysgro.2019.05.022.

[8] Lutz, M. *Learning Python, Fourth Edition.* 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., fourth edition, 2009, ISBN 978-0-596-15806-4.

[9] Stroustrup, B. *The C++ programming language*. One Lake Street, Upper Saddle River, New Jersey 07458: Pearson Education, Inc., fourth edition, 2013, ISBN 978-0-321-56384-2.

[10] *scikit-learn: machine learning in Python [online]*. 2020, [cit. 2020-02-08]. Available from: `https://scikit-learn.org/`

[11] Frank, E.; Hall, M. A.; et al. *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. Morgan Kaufmann, fourth edition, 2016, [cit. 2020-02-08]. Available from: `https://www.cs.waikato.ac.nz/ml/weka/Witten_et_al_2016_appendix.pdf`

[12] Kitware, Inc. *CMake [online]*. 2020, [cit. 2020-02-08]. Available from: `https://cmake.org/`

[13] *Doxygen: Main Page [online]*. 2020, [cit. 2020-02-08]. Available from: `http://www.doxygen.nl/`

[14] Suryanarayana, G.; Ganesh, S.; et al. *Refactoring for Software Design Smells: Managing Technical Debt*. Elsevier Science & Technology, 2014, ISBN 978-0-12-801397-7.

[15] Mili, A.; Tchier, F. *Software Testing : Concepts and Operations*. John Wiley & Sons, Inc., first edition, 2015, ISBN 978-1-118-66287-8.

[16] *catchorg/Catch2: A modern, C++-native, header-only, test framework for unit-tests, TDD and BDD - using C++11, C++14, C++17 and later (or C++03 on the Catch1.x branch) [online]*. 2020, [cit. 2020-03-25]. Available from: `https://github.com/catchorg/Catch2`

Appendix A

Acronyms

BDD	Behavior-Driven Development
CLI	Command Line Interface
CSV	Comma-Separated Values
DC-PBC	Discriminant Chronicles Pattern-Based Classification
DCM	Discriminant Chronicles Mining
GUI	Graphical User Interface
K&R	Kernighan and Ritchie
LoC	Line of Code
ML	Machine Learning
STL	Standard Template Library
SVC	Support Vector Classifier
TDD	Test-Driven Development
WEKA	Waikato Environment for Knowledge Analysis

Appendix B

Contents of enclosed CD

```
README.md ............................ file with CD contents description
|__ results . directory with resulting chronicles and parameter tuning results
|__ src .................. directory with source codes of the implementation
|   |__ DCM .......................... source codes of the DCM component
|   |__ DC-PBC ....................... source codes of the DC-PBC component
|   |__ datasets_old .......... datasets provided by DCM/DC-PBC authors
|   |__ datasets_new ...................... datasets used for demonstration
|   |__ evaluation_tools ... chronicle_statgen tool used for demonstration
|   |__ testing_patches ........... patches used for generating the test data
|   |__ testing_tools ............. dcm_differ tool used as a testing helper
|   |__ component_testing ............. source codes of the component tests
|   |__ integration_testing ........... source codes of the integration tests
|__ thesis_src ................ directory of LaTeX source codes of the thesis
|__ DP_Busa_Radek_Bc.pdf ....................... thesis text in PDF format
```