|  |  |
| --- | --- |
|  | |
| **CONFIDENTIAL DOCUMENT**  The reproduction and distribution of this document is forbidden without written authorization from Medial-Research LTD | |
|  | |
| **Title** | **AlgoMarker Library SW Version 1.1 - Software Design Document** |
| **Doc No’** | **RDG-04-11-33 Rev. D** |
| **Based on Template:** | **RDF-04-11-12 Rev. A** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Prepared by:** | **Name** | **Signature** | **Date** |
| Head of Data Science | Alon Lanyado |  | 13-Jun-2023 |
|  |  |  |  |
| **Approved by:** | **Name** | **Signature** | **Date** |
| Senior Director of Engineering | Eldan Halberthal |  |  |
| VP Product | Rinat Gutman |  |  |
| Chief of Clinical Research and Compliance Officer | Eran Choman |  |  |
| Software Quality Manager | Michal Mahlab |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision**  **Status** | **Change**  **Summary** | **DCR No.** | **Date** |
| A | Baseline | 33-18 | 10-Jul-2018 |
| B | Remove specific AM name from the title and introduction | 6-19 | 21-Jan-2019 |
| C | Add SO to design | 46-19 | 28-Jul-2019 |
| D | Moved to json API, add explainability functionality, added option to check eligibility by model |  | 13-Jun-202324-Jul-23 |

**Table of Contents**

1. Introduction 3

1.1. Purpose of This Document 3

1.2. Scope 3

1.3. Definitions, Acronyms and Abbreviations 3

1.4. System Overview 3

1.5. Applicable Documents 5

1.5.1. Applicable Standards and Regulations 5

1.5.2. Applicable Procedures and Guidelines 5

1.5.3. Applicable Project Documents 5

1.6. Overview 5

2. System Architecture 6

3. Decomposition Description 7

3.1. Overview of Software Classes 7

3.2. Software Classes Decomposition 7

3.2.1. AlgoMarker 7

3.2.2. MedialInfraAlgoMarker 7

3.2.3. Input json for AddDataByType 8

3.2.4. Input json request to CalculateByType 8

3.2.5. Output json response 9

3.2.6. Explainability output json 9

3.2.7. CalculateByType API flow 10

3.2.8. Error codes: 11

3.3. Runtime Decomposition 11

4. Dependencies Description 11

4.1. Component Dependencies 11

4.2. Inter-module Dependencies 11

4.2.1. Dependency map 11

4.3. Threading Dependencies 12

4.4. Data Dependencies 12

5. Interface Description 12

5.1. AlgoMarker base class API 13

5.2. MedialInfraAlgoMarker API 13

5.2.1. Class API 13

5.2.2. The configuration file format 13

5.2.2.1. General configuration file parameters 13

5.2.2.2. Eligibility tests configuration 15

5.2.3. Signals and channels 16

5.2.4. Timestamps formats 16

5.3. AlgoMarker API – the AM API 16

5.4. Communications and Hardware Interface 17

5.5. External Interface 17

6. Appendices 17

6.1. Requirements Traceability from the SRS 17

6.2. Additional Information 18

# Introduction

## Purpose of This Document

This purpose of this document is to present the Software Design for the AlgoMarker library. This document forms the baseline for the implementation of the software. The AlgoMarker library is provided as a part of AlgoMarker products.

## Scope

This Software Design Document (SDD) specifies the software design for the software of the AlgoMarker library. The document is written according to IEEE standards (see para.‎1.5.1 ref. ‎1) with appropriate tailoring.

## Definitions, Acronyms and Abbreviations

The IEEE standard (see para. ‎1.5.1 ref. ‎2), and the list of terms, acronyms and abbreviations with their associated meanings shown below, are used in this document.

| **#** | **Term/ Acronym** | **Definition** |
| --- | --- | --- |
|  | API | Application Program Interface |
|  | DLL/SO | Dynamic Link Library (Windows)/Shared Object Library (Linux) |
|  | HTTP | Hypertext Transfer Protocol |
|  | HTTPS | Secured Hypertext Transfer Protocol |
|  | ID | Identification |
|  | SDD | Software Design Document |
|  | SRS | Software Requirements Specifications |
|  | URL | [Uniform Resource Locator](https://en.wikipedia.org/wiki/Uniform_resource_locator) |

## System Overview

The AlgoMarker library enables to create wrappers for medical prediction models and calculators in a way that:

* Enables to get predictions or insights like explainability (calculator results are also predictions in this sense)
  + For single prediction points or a batch.
* Exposes a unified, simple, yet rich enough API in order to do so. The API is intentionally written in a manner that enables easy wrapping for usage from either C/C++ code, or C# code.
* Enables a smooth usage of models built using Medial EarlySign libraries and tools.
* Ensures the exact same models used in development and performance analysis are the ones used in the field.
* Contains mechanisms to configure and test eligibility of input data.
* Allows an architecture in which a single DLL/SO can be used for many different AlgoMarkers.
* Allows an easy creation of new types of AlgoMarkers using an inheritance mechanism, this makes sure the AM API works automatically as long as the needed inherited virtual functions are implemented in the child AlgoMarker class.

The AlgoMarker library (and DLL/SO) relies on several large libraries developed internally in Medial, the major ones are:

* AlgoMarker wrapper library and API (will be described in this document)
* InfraMed library: handles creation and usage of medical data repositories.
* MedAlgo library: a collection of many different prediction algorithms, classifiers and regressors (linear regressions, logistic regressions, Random Forest variants, Gradient Boosting trees variants, Neural Nets, Clustering algorithms, and more) with a unified API and an easy ability to switch one with another.
* MedProcessTools library: the major library to create models, taking care of a full process for training, serialization, and predictions, in particular handling:
  + Cleaners
  + Imputers
  + Signal and Feature generation
  + Preparing matrices for classifiers/regressors
  + Normalization
  + Training classifiers/regressors
  + Post processors – calibration, fairness, explainability

The scope of this document is the AlgoMarker library only, and not the detailed description of the large algorithmic libraries it relies on.

The AlgoMarker library enables an easy creation of new AlgoMarker types (as explained above). To do so the new AlgoMarker inherits from the base AlgoMarker (or an existing one) and needs to implement several basic functions: Load, Unload, ClearData, AddDataByType and CalculateByType.

In addition, several basic classes are defined and used in the AlgoMarker library that define the "language" of defining the inputs and outputs of an AlgoMarker using json which is versatile enough to present complicated objects.

## Applicable Documents

The following specifications, standards and documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered a superseding requirement.

### Applicable Standards and Regulations



| **#** | **Title** | **Revision/Date** |
| --- | --- | --- |
|  | IEEE Std 1016-2009, IEEE Standard for Information Technology – Systems Design - Software Design Descriptions | 20 July 2009 |
|  | ISO/IEC/IEEE 24765:2010 - Systems and Software Engineering – Vocabulary | 2010-12-15 |

### Applicable Procedures and Guidelines

| **#** | **Title** | **Revision/Date** |
| --- | --- | --- |
|  | RDP-04-11 Software Development Life Cycle Procedure | Rev. M |
|  | RDP-04-12 Design Control | Rev. H |

### Applicable Project Documents

| **#** | **Title** | **Revision/Date** |
| --- | --- | --- |
|  | RDS-04-11-18 AlgoAnalyzer 1.1 Software Requirements Specifications | Rev. D |

## Overview

This Software Design Document contains the following sections:

* Section 1 - Introduction
* Section 2 - System Architecture
* Section 3 - Decomposition Description
* Section 4 - Dependencies Description
* Section 5 - Interfaces Description
* Appendix A -Traceability to the Software Requirements

# System Architecture

The following schematics describes the general relations between the libraries described in the previous section. The AlgoMarker DLL/SO/Lib in the middle enables an external user such as the AlgoAnalyzer (on left in the drawing), to get results for a specific AlgoMarker using the AM API (for example a Pre2D AlgoMarker or a ColonFlag AlgoMarker) using the specific AlgoMarker binary model and configuration files. In order to do so, the AlgoMarker libs rely on several large libraries, the major ones of which are given in the right side of the drawing.

InfraMed

lib

AM API

External User

(i.e. AA)

AlgoMarker

DLL/SO/Lib

MedAlgo

lib

Specific

AlgoMarker

Configuration

MedProcessTools

lib

# Decomposition Description

In this section we will only describe the AlgoMarker library and not the algorithmic libraries it relies on.

This section describes the software architecture and its decomposition to modules. This section contains the following subsections:

* Overview of the objects
* Description of each object, its major members and major API's.
* Runtime description of what happens when an AlgoMarker is being loaded and used.

## Overview of Software Classes

This section provides an overview of the software classes and their capabilities as listed in the following table:

|  | **Class** | **High level Description** |
| --- | --- | --- |
| 1 | AlgoMarker | General base class for AlgoMarkers |
| 2 | MedialInfraAlgoMarker | An inherited AlgoMarker that enables the usage of Medial models. |

## Software Classes Decomposition

This section provides the description of each software class.

### AlgoMarker

AlgoMarker is the general base class for any type of AlgoMarker. The AM API is implemented in a way that relies ONLY on this base class and the definition of a few major virtual functions. This means that any class inheriting from AlgoMarker will immediately work well with the AM API, and thus this creates a fast easy path for creating new AlgoMarker types. The major API's of this class are described in the section for API's.

### MedialInfraAlgoMarker

Is a class inheriting from AlgoMarker that implements an AlgoMarker that:

* Loads from a config file containing:
  + The actual model used
  + Definitions for the internal data repository model
  + Definitions for eligibility tests to be done on input data
  + Definitions of explainability processings if applicable
* Enables loading data into its in-memory repository to be retrieved later when calculating the score
* Upon getting a request:
  + Runs eligibility tests on each prediction point in the request
  + Returns scores for each point, plus errors and warnings if there are any.

The API of the the MedialInfra AlgoMarker is the same as the one for an AlgoMarker and will be described in the API's section. We will describe there also the config file and the different ways of using it.

### Input json for AddDataByType

The signature is (const char \*data, char \*\*messages)

The data request also supports receiving data in batch. The AlgoMarker will search for opening “{“ and closing brackets “}” for each request in the data string and will handle all requests if there is more than 1. The data input is concatenated jsons or single json request. Here we will describe a single json.

If the json is invalid “AM\_ERROR\_DATA\_JSON\_PARSE” error will be returned (coded as 1202).

The input json can start with “**body**” element or not and then those elements.

* “**patient\_id**”: integer numeric identifier for the patient we are loading the data for. Should be provided as numeric and not string.
* “**signals**”: an array of json blocks with those elements that each describes a signal:
  + **“code”:** the name of the signal. Should match list of signals
  + **“data” :** An array of json blocks with those elements that each describes a single data point for the current signal type:
    - **“timestamp”**: an array of timestamps. Matches the signal’s time channels size. Can be also empty. The type is numeric long long
    - **“value”**: an array of values. Matches the signal’s value channels size. The type is string. If the signal is categorical AM will use dictionaries to convert this string to the numeric code. If it’s suppose to be numeric, it will try to convert it to float.

### Input json request to CalculateByType

The input json defines which scores are requested from the AlgoMarker. It contains the following:

* “**type**”: which should be “**request**”. Other types are not supported.
* “**request\_id**”: a token that will be copied to the matching json response answer in order to allow for the user of the API to match requests and responses.
* “**requests**”: an array of requests for calculations
  + “**patient\_id**” or “**pid**” - the patient identifier
  + “**scoreOnDate**” or “**time**” - the requested time for prediction. Applying the model with respect to this time.
  + “**load**” - optional boolean flag to indicate patient data is within this request. Default 0. If data is 1, we should have “data” element under it:
    - “**data**” (only when load is provided and load=1) - json of AddDataByType was described before in AddDataByType section
  + “**export**” - array of elements to extract from model output.
    - Based on key and value, the key will be used in the response json to hold the result. The value is the element to extract from the model.
      * The value can be ”pred\_X” or “pred X” (with underscore or space) to take prediction from model. Usually X=0, but there might be cases where the model returns multiple predictions
      * A token of 2 fields, with space between them. Starts with “attr” and the name of the attribute to extract from the samples (might be useful for example in Diabetic coder)
      * A token of 2 fields with space between them. Starts with “json\_attr” and the name of attribute to extract that is json format (for example explainability)
      * A token of 2 fields with space between them. Starts with “json” to fetch information from “json” in the samples object

Those fields of “export” can appear under the json and not under “reqeusts” if there is a single request or if we want to build “default”. Useful if we want to define the “export’ once for all listed requests in this json.

This is used in AM\_API\_CalculateByType

### Output json response

The output json response holds the result of a CalculateByType() operation of an AlgoMarker. It contains the following:

* **“type”**: “response”. It is always “response” type
* “**request\_id**” - copy of the request id
* “**errors**” (optional) - if there are errors. Array of error messages.
* **"responses" -** an array of results
  + “**patient\_id**” - the patient id the result relates to
  + “**time**” - the requested prediction time
  + “**DEPEND\_ON\_EXPORT\_KEY\_FIELD**S”: the result of this extraction from the model

The AM API uses json response in the following functions:

* AM\_API\_CalculateByType

### Explainability output json

Part of the response output “**DEPEND\_ON\_EXPORT\_KEY\_FIELD**S” can be exaplainability. I will describe the output json schema here.

* “**static\_info**”: an element that contains array of other elements that contains general information about the patient. This information will always be presented and it’s part of the explanability configuration. Each sub element will contain the following fields
  + “**signal**”: name of the signal that we are revealing, can be also “Age” and not a real signal
  + “**value**”: A string value of the signal. If no information is available, we will return “Missing”. If it’s numeric, a string representation of the value will be presented.
* “**explainer\_output**”: an element that contains array of other elements. Each element is a contributing factor that changed the score. Each element will contain those fields:
  + “**contributor\_name**” – name of the contributor
  + “**contributer\_value**”: numeric value with the raw influence level of this contributor (should not be presented to client). If the value is positive, it means this contributor caused the model to increase the score, if it’s negative the contributor decreased the score.
  + “**contributer\_percentage**”: numeric value that normalize this contribution value in percentage out of all other contributors that exists (not all of the contributors might be exposed in the result, so it won’t sum up to 100)
  + “**contributer\_elements**” – An array of other elements that contain the specific features of the model that relates to this contributor group. Each contributor might be a set of 1 or more features. Each element will contain those fields:
    - “**feature\_name**” : name of the raw model feature name
    - “**feature\_value**”: numeric value of this feature. Missing values will be presented as -65336.
  + “**contributor\_description**”: A string that present different aliases to the category if applicable. For example, when the contributor is diagnosis code, a string with all other aliases separated by “|” will be presented. Otherwise it will be empty string.
  + “**contributor\_level**”: a scaled contributor influence level that should be presented to client. An integer, higher means higher influence, ranges from 0 to “contributor\_level\_max”
  + “**contributor\_level\_max**”: an integer with the maximal level of influence. The scaling range of this influence level. Can be controlled by “max\_threshold” and “num\_groups” in the amconfig explainability settings.
  + “**contributor\_records**”: an array of references in the EMR that relates to this contributor concept/group. Each element will contain those fields:
    - “**signal**”: name of the signal or “Age”, “Gender”, name of the feature if it’s based on single feature (mainly age, gender are contributor groups with single feature).
    - “**timestamp**”: an array of numeric timestamps that relate to this signal if applicable. Can be empty array.
    - “**value**”: an array of string values of the signal. If the signal is numeric, 6 digits will be presented.
  + “**contributor\_records\_info**”- an element that contains meta data information on the contributor records. It will contain those fields.
    - “**contributer\_max\_time**”: Will present a numeric integer with the time window limit that was used to fetch the contributor records
    - “**contributer\_max\_time\_unit**”: the unit that relates to the contributor\_max\_time. Mainly “Days”
    - “**contributor\_max\_count**”: The limit on how many records we will fetch for this signal, from the most recent timestamps.

### CalculateByType API flow

The CalculateByType goes through those steps:

1. Not valid json - "ERROR: Could not parse request as a valid json" and exits

2. Check the request type - "ERROR: missing type request". "type" should be "request"

3. Check for request\_id - "ERROR: no request\_id provided"

4. Checks it has "requests" - "ERROR: missing actual requests in request XXX"

5. If "load" is turned on. Will call AddDataByType. If there is error in data load - "ERROR: error when loading data for patient id XXX"

6. Checks patient\_id and requested time. patient is integer >0. time is valid time in years 1950-3000. No testing of valid date. - "ERROR: BAD request patient id or time : failed in inserting pid: "

7. Checks eligibility before model - no errors

8. Apply model - if something is wrong (bug). "Failed getting scores in AlgoMarker"

9. Checks eligibility after model - no errors

10. If one of the eligibility testes crashed - "ERROR: sanity tests crashed"

11. If one of the tests didn't pass - "Custom/configurable message by the AlgoMarker"

12. If export field is not in current request respose: "ERROR: did not get result for field"

13. If export requested prediction channel that doesn't exists - "ERROR: prediction channel"

### Error codes:

List of error codes and description of error codes can be found in here: [error\_codes.xlsx](https://medial.sharepoint.com/:x:/r/sites/algoteam/Shared%20Documents/General/AlgoMarker/error_codes.xlsx?d=w04b4e7905ddd45bc85a0c2fadc267433&csf=1&web=1&e=wXaT2n)

## Runtime Decomposition

When using an AlgoMarker things happen in the following manner:

1. Create and allocate a new AlgoMarker (AM\_API\_Create)
2. Load the AlgoMarker, one can pass a string representing a config file name for it (AM\_API\_Load)
3. Add data to the AlgoMarker using the AM\_API\_AddDataByType API. The loaded data can be for a single patient, or for several ones (a batch). Several AddDataByType calls can be made one after the other in order to load more and more data into the AlgoMarker. This data will later be used to request scores for one or several Calculate requests.
4. Make requests on loaded data:
   1. Create a request json string
   2. Call the CalculateByType API to actually calculate the scores (AM\_API\_CalculateByType)
   3. Get json response output
   4. Steps a-c above can be repeated as many times as needed.
5. Clear data – release all the data loaded before, and make AlgoMarker ready to get a new batch of data. (AM\_API\_ClearData), one can go back to step 3 here if needed.
6. Dispose of the AlgoMarker (AM\_API\_DisposeAlgoMarker)

# Dependencies Description

This section presents the relationships among design entities and system resources. This section contains the following subsections:

* Component dependencies
* Inter-module dependencies
* Inter-process dependencies
* Data dependencies

## Component Dependencies

NA.

## Inter-module Dependencies

The dependencies between the different libraries and classes are described in the following map.

### Dependency map

AlgoMarker DLL/SO/Lib

MedInfraAlgoMarker

class

AlgoMarker class

InfraMed

lib

MedAlgo

lib

MedProcessTools

lib

## Threading Dependencies

The AlgoMarker API is single threaded, and is a library, so will run in the context of the user of this library, who's going to use it via the AM API.

However – the implementation of a specific AlgoMarker can use threads for its basic operations.

MedialInfraAlgoMarker uses threads in many places in a hidden way, as the basic algorithmic libraries: InfraMed, MedAlgo and MedProcessTools use threads for better performance. These threads are opened only while a Load() or CalculateByType() method is used and are closed when the method returns.

In MedialInfraAlgoMarker:

* AddDataByType() is not thread safe, and hence add data calls should be sequential.
* CalculateByType() is not thread safe, but handles a batch request for a large number of prediction points at a time. This is threaded inside and hence is more efficient than calling these one by one.

## Data Dependencies

The only data dependency is for an instance of a MedialInfraAlgoMarker. This AlgoMarker type loads from a config file of a certain format that will be described in the API section.

# Interface Description

This section describes and references the API's for:

* AlgoMarker base class: needed to be understood in order to implement new AlgoMaker types,
* MedialInfraAlgoMarker : additional information regarding the specific MedialInfra AlgoMarker implementation such as the format of its config file,
* The AlgoMarker API (the AM API) which is the official way to work with the AlgoMarker library.

## AlgoMarker base class API

There are 5 major API's each AlgoMarker is required to implement:

* Load(const char \*config\_f) : Loading a new algomarker from a config\_file (or without one if the algomarker loads without it, in that case the implementation will ignore the config\_f parameter).
* Unload() : Releasing all memory allocations done by algomarker, and makes it free ready.
* ClearData() : Clearing all loaded data.
* AddDataByType(const char \*data, char \*\*messages) : Loading data into the algomarker, for json input object and returns list of warning/erros in the data loading.
* CalculateByType(int CalculateType, char \*request, char \*\*response) : actually calculating the scores for all the prediction points given in request, using the data that was loaded before or given in the request. Response will be later needed to be free from memory.

## MedialInfraAlgoMarker API

### Class API

This is similar to the API of a general AlgoMarker as defined in 5.1 above.

### The configuration file format

A MedialInfraAlgoMarker (MI-AM) is loaded using a configuration file containing all the information needed in order to load it, define it and get it to a working state.

The typical usage is having the config file reside in a directory together with all the other needed files. Upon load the MI-AM detects the directory from the given configuration file name and will allow giving other file names inside in a relative path. This way the location of the directory is not rigid and it can be moved freely while still maintaining a smooth load.

Lines starting with '#' are comment lines.

Other lines have the format of a <field name> and <value> that are tab separated.

### General configuration file parameters

* **TYPE** : should always be MEDIAL\_INFRA for a MI-AM.
* **NAME** : the name of the MI-AM, can be get later via the AM\_API
* **TIME\_UNIT** : The time unit to be used for the signals. In Ver 1.0 should be Date, although others (such as Minutes) would probably work as well, but are not for this version.
* **REPOSITORY**: a repository file name. This is based on repositories as defined and used by the InfraMed library. For a MI-AM we only need the definitions parts of the repository, and only those that are needed for the specific AM. Mainly it means it needs to have:
  + Signals definition file: containing the names of all the signals as the model uses them, their numerical codes, and their type, which in itself defines how many time channels and how many value channels they have.
  + Dictionaries: dictionaries contain the transformation from strings to codes, as the model uses. It is extremely useful when handling data such as drug information or read codes, ICD9 codes, or any large categorical data. Along these dictionaries allow to define sets of categories, which are often used when building a model.
  + The formal format of the repository and its configuration files is beyond the scope of this document.
  + If the repository config file is given without a leading '/' it will be relative to the directory of the MI-AM configuration file. Otherwise – it will be a full path to it.
* **MODEL**: the model file name, as created by medial tools when training this model.
* **MODEL\_END\_STAGE:** (optional) to control where to stop the model calculation, default is till the end**.**
* **~~EXTENDED\_RESULT\_FIELDS:~~** ~~list of tokens separated by “;”. (deprecated, not in use in current API)~~
* **DEBUG\_MATRIX**: option to store debug matrix if given**.**
* **AM\_UDI\_DI, AM\_MANUFACTOR\_DATE, AM\_VERSION –** version string
* **EXPLAINABILITY\_PARAMS:** parameters for explainability if applicable. Given as a string with “;” delimeter between each parameter. The parameters are given as key=value
  + **max\_threshold –** the threshold for scaling the shapley scores. What’s the maximal threshold that above this we will “trim” the contribution. Will be used also for binning the range**.**
  + **num\_groups:** number of groups to divide the contribution level**.**
  + **use\_perc:** calculate the threshold on the percentage. Default is false**.**
  + **total\_max\_reasons –** limit on the maximal number of positive and negative contribution list
  + **total\_max\_pos\_reasons, total\_max\_neg\_reasons**: limit on the maximal number of positive/negative contribution list. Separate constrain for positive and negative.
  + **ignore\_groups\_list –** name of groups to ignore and omit in Explainability**.**
  + **threshold\_abs –** threshold that below this (in absolute value) the contributor is omitted
  + **threshold\_percentage** – threshold that below this (in percentage value) the contributor is omitted**.**
  + **static\_features\_info** – a string with comma “,” to fetch static list of signals and always present them. If missing, “missing” will appear in the result. Should be list of signals or “Age”**.**
  + **cfg** – a file path to configure settings for presenting and fetching info on groups (\*)

(\*) Explainability config file format – tab delimited. Each row corresponds to what to present for a specific “group” of contributors. If not given and the group has more than one feature, nothing will be presented. If there is only one feature it will return the feature value. Those are the columns:

* **column 1: contributer\_group\_name** : the name of the contributer group that we want to refer to in this row
* **column 2: signal\_name** : name of signal to fetch for presenting something on this group
* **(optional) column 3: max\_count**: how many items of most recent events to fetch for this signal (default is 1)
* **(optional) column 4: max\_time\_window**:maximal time window to fetch information from prediction request. Omit too old events. Default 0 which means only from current date
* **(optional) column 5: time\_channel:** which time channel to use (default 0)
* **(optional) column 6: time\_unit:** The time unit of the time window (default is Days)
* **(optional) column 7: val\_channel**: a value channel to use for filtering events if next column is given (default is 0)
* **(optional) column 7: sets:** comma separated list of codes to filter val\_channel event of signals. Supports only categorical. Default is empty. if empty, will not filter.

### Eligibility tests configuration

* **INPUT\_TESTER\_CONFIG**: if '.' means that the definitions for the tests configuration are in the configuration file, otherwise – in the file given.
* **FILTER** : followed by a '|' separated string with the following fields:
  + <filter type>
    - 'simple' : a simple filter allowing tests of ranges, existence, correct codes, in different time windows. See filtering params below. These are typically tested before the actual prediction is done.
  + 'attr' : a model can be configured in preparation and training to output several attributes for each prediction point, these attributes can be the results of the cleaners running on the patient data, and can then be queried to create an eligibility test. See the matching parameters below.<filter params>
    - Many options, explained below
  + <warning\_or\_error>
    - Values are WARNING or ERROR , and define how to treat this test (as a warning only or as an eligibility error)
  + <use\_for\_max\_outliers\_flag>
    - ACC=1 or ACC=0 , signs which filters are used when summing the outliers for the MAX\_OVERALL\_OUTLIERS test
  + <external\_rc>
    - Return code to be passed by the message
  + <internal\_rc>
    - Return code used internally for debugging and logging
  + <err\_msg>
    - String – free text of the error that will be returned in the case the defined test does not pass. Can use “$VALUE” token to refer to unknown code in simple filter when “values\_in\_dictionary” is 1.
  + <cant\_evel\_msg>
    - String – free text of error message when we can’t calculate the eligibility.
  + <stop\_processing\_more\_errors >
    - If 1 will stop processing more eligibility rules if this fails
* Filter params for 'simple' filter explained:
  + All given in a <name>=<val> format and are ';' separated. Besides sig, all parameters are optional , and their combination by the user is what creates a specific test.
  + sig : the signal we test (i.e. Glucose, GENDER, AGE, ….)
  + win\_from , win\_to : defining a windows (in version 1.0 – in days units) before the current prediction point. For example win\_from=0;win\_to=365 is a window of up to 1Y before the prediction point.
  + min\_Nvals : tests that at least the given number of values from the signal appear in the time window in the given range.
  + max\_Nvals : tests that at most the given number of values from the signal appear in the time window.
  + min\_val, max\_val : defines the non-outlier range for the signal.
  + min\_left : tests that at least this number of non outliers remained in the given window.
  + time\_ch, val\_ch : channels to work on in the test (default 0,0)
  + values\_in\_dictionary : if 1 , tests that the values appear and are defined in the repository dictionary.
  + allowed\_values : a list of values separated by ',' can be given and the test will test that all values are in this list.
* Filter params for 'attr' filter explained:
  + Name: the name of the attribute to test (needs to be an actual attribute the model is designed to output).
  + Max: the maximal value allowed for the attribute to be considered OK.
* Filter params for “feature\_json” filter explained – will be used to run a different model and use one of the model features to determine the eligibility:
  + feature\_name – name of feature to extract from model/json model to determine if passed or didn’t pass the test
  + feat\_min\_val, feat\_max\_val – threshold to define the good range to pass the eligibility test for the extracted feature. (default is missing value and will no check the lower/upper bound)
  + json\_model\_path – path to model or json model. Can also be relative to amconfig file
  + is\_binary\_model – a binary b it 1/0 to determine if json\_model\_path is binary model or json
  + verbose\_learn, verbose\_apply – binary flags 1/0 to indicate verbose output in applying those steps. “learn” will happen once only if the model is not binary and json. The “learn” should be something that doesn’t need to infer something from the data (empty samples are given to this learn)
  + ~~allow\_missing\_signals – comma separated list of signals that we allow to be missings. Deprecated. Not needed anymore~~.

### Signals and channels

The repository model defined a constant number of time channels and constant number of value channels for each signal. These types are defined in the InfraMed library, and each signal used by the algomarker has its type given in the repository signals file.

These should be understood and known by the user, and are important when adding data to a MI-AM.

When doing so, using the AddDataByType() API, one has to provide an array of times and an array of values, as well as the number of elements to load. The packing is all the channels from first to last, and then the next element, for both timestamps and values.

### Timestamps formats

MI-AM Ver1.0 supports time formats of :

* YYYYMMDD
* YYYYMMDDhh
* YYYYMMDDhhmm
* YYYYMMDDhhmmss

And will convert internally and automatically the given format to the needed timestamp used by the algomarker.

## AlgoMarker API – the AM API

The AlgoMarker library implements the following API to be used by an outside user (such as the AlgoAnalyzer for example):

AlgoMarker API

|  |  |
| --- | --- |
| **1** | **AM\_API\_Create**(int am\_type, AlgoMarker \*\*new\_am) |
| Create a new AlgoMarker of type am\_type.  Each new added AlgoMarker inheriting class needs to have a name and define it in the make\_algomarker method in the base AlgoMarker class. |
|  |  |
| **2** | **AM\_API\_Load**(AlgoMarker\* pAlgoMarker, const char \*config\_fname) |
| Loading AlgoMarker and making it ready to get Requests. For an AlgoMarker using a configuration file config\_fname is the configuration file, for those that load without it this parameter will be ignored. |
| **3** | **AM\_API\_ClearData**(AlgoMarker\* pAlgoMarker) |
| Clearing all data loaded into the AlgoMarker (recommended at the start and/or end of each query session |
| **4** | **AM\_API\_AddDataByType**(AlgoMarker \* pAlgoMarker, const char \*data, char \*\*messages) |
| Loads data into the AlgoMarker from data json. Returns error code RC\_AM\_FAILD (-1) and error messages in messages if has errors. Can also return AM\_ERROR\_DATA\_JSON\_PARSE (1202) if not valid json. Please use AM\_API\_Dispose to dispose the message object |
| **5** | **AM\_API\_CalculateByType**(AlgoMarker \*pAlgoMarker, int CalcType, char \*request, char \*\*responses) |
| Get scores for a input json request and put them into the response json. |
| The CalcType should be 3001, “JSON\_REQ\_JSON\_RESP”. Please use AM\_API\_Dispose to dispose the responses object |
| **6** | **AM\_API\_Dispose**(char \*data) |
|  | To clear response data or AddDataByType message string |
| **7** | **AM\_API\_DisposeAlgoMarker**(AlgoMarker \*pAlgoMarker) |
| Dispose of AlgoMarker - free all memory |
| **8** | **AM\_API\_Discovery**(AlgoMarker \*pAlgoMarker, char \*\*resp |
|  | Returns json string that describes the AlgoMarker inputs and potential outputs for example explainability contributors. Info on model and versions info. Please use AM\_API\_Dispose to dispose the resp object |

## Deprecated API methods

***The following API methods are not documented in this document, but can be found in RDG-04-11-33 AM Library SW Version 1.0 Software Design Document – Rev C. Those methods are deprecated and won’t be used in future AlgoAnalyzer versions. The new AlgoMarker still contains them, but no further improvements will be released for those API calls:***

* **AM\_API\_AddData**
* **AM\_API\_CreateRequest**
* **AM\_API\_CreateResponses**
* **AM\_API\_Calculate**
* **AM\_API\_GetResponsesNum**
* **AM\_API\_GetSharedMessages**
* **AM\_API\_GetResponseIndex**
* **AM\_API\_GetResponsesRequestId**
* **AM\_API\_GetResponseScoreByType**
* **AM\_API\_GetResponseAtIndex**
* **AM\_API\_GetResponseScoresNum**
* **AM\_API\_GetResponseScoreByIndex**
* **AM\_API\_GetResponseMessages**
* **AM\_API\_GetScoreMessages**
* **AM\_API\_GetResponsePoint**
* **AM\_API\_GetName**
* **AM\_API\_DisposeRequest**
* **AM\_API\_DisposeResponses**

## Communications and Hardware Interface

No communications or hardware interfaces exist within the context of the AlgoMarker Library/DLL/SO.

## External Interface

The MedialInfraAlgoMarker uses several large external algorithmic libraries, main ones are MedInfra, MedAlgo and MedProcessTools. The scope of working with these libraries and interfaces to them is (way) beyond the scope of this document.

# Appendices

The following appendices are defined:

* Requirements traceability from the SRS
* Additional information

## Requirements Traceability from the SRS

The following traceability matrix, shown in the following table, traces the software requirements, as specified in the AlgoAnalyzer SRS document (see para. ‎1.5.3 ref. ‎1), to the units in this SDD.

| **SRS Req. #** | **Title / Short description** | **SDD Para #** | **Title / Module / Unit** |
| --- | --- | --- | --- |
| SRS-AMR-001 | Verify valid file paths, and load all files | 3.2.2  5.3 | MedialInfraAlgoMarker class  Load API |
| SRS-AMR-002 | Confirm AlgoMarker name | 3.2.2  5.3 | MedialInfraAlgoMarker class  GetName API |
| SRS-ACL-001 | Load patient data, verify the score type | 3.2.2  5.3 | MedialInfraAlgoMarker class  AddData API |
| SRS-ACL-002 | Return the same requestId, a calculation valid time, score | 3.2.2  5.3 | MedialInfraAlgoMarker class  GetResponsesRequestId API |

## Additional Information

A practical guide to using the AlgoMarker and MedialInfraAlgoMarker library can be found in Medial internal wiki under:

http://confluence:8090/display/WIK/AlgoMarkers