Extensible Data Model with Applications for Trading Systems

Iosif Ziman

***Abstract*—An extensible main-memory data model is presented with applications in writing client-server components both on the server side as well as in the client applications domain. Such a model is required in applications where end user base clients have specific needs and it offers a framework within which various services are implemented based on a common extensible core. The end result of the proposed implementation is a core set of services that uses an XML based API which can be used to define data structures, events and actions in an easily usable package that allows iterative updates and evolution of the environment. Use cases are presented for the implementation of trading systems used in the capital markets.**

***Index Terms*—main-memory database, client-server, framework, API**

I. Introduction

THE work on EDM came about as the personal struggle of the author while building the Fusion Order eXecution platform [4] to find a way of building complex trading systems that also allow a great degree of customization. After several very successful attempts at building bespoke order-execution trading systems, the complications arise from the need to develop and maintain a platform for many clients who tend to have different, sometimes mutually exclusive requirements. Some trading systems vendors, such as Fidessa [6] have been successful at introducing such platforms. One of the main ingredients in building this type of system is an easily configurable in-memory database, using a script based language. There are a limited number of practically usable in-memory-database systems and most are proprietary implementations owned by large corporations, for ex. Oracle owned TimesTen [8]. As such an enterprise wishing to use a high performance, flexible platform allowing customizable implementations while maintaining costs low may go through the route of using a proprietary developed framework. EDM is such an environment. The subject of main memory database systems is not new and has been analyzed of some time, as early as the ‘80s [5], yet due to their still somewhat esoteric nature, when compared with traditional relational database systems, main-memory database systems remain worthy of more intense research. The paper presents the detailed approach to implementing such a main-memory database system, with application directly in building systems used in the capital markets. Aspects considered are the building of relevant language and parser constructs, database primitives and data hierarchy, and offers an in-depth view of the mechanisms and level of detail required to consider when building the framework.

Manuscript received July 22, 2012; revised Aug 10, 2012.

I. Ziman is with BFAM Partners (Hong Kong) Ltd. (e-mail: iziman@ bfam-partners.com).

II. The Purpose of EDM

The main reason for using a model like EDM is to formalize the approach to building customizable, service and main-memory based trading systems in the capital markets space. While such systems have many common needs with other systems that require stability, high volume throughput, low latency (such as in the case of telecommunications), one of the relevant aspects for trading systems is their perpetual evolution and need to customize to the level of each, possibly, individual client. Such a degree of development and maintenance is not possible to be executed by software providers, and so the need to allow clients themselves to offer “in-house” customization. When building such a framework we have consulted environments already existing [4] as well as studied theoretical aspects related to ways of approaching building such systems as well [6-7].

The key areas that EDM is looking to propose a solution for are thus: use cases where different clients have different service needs; different clients need different data views where the data is computed based on different rules, etc.. The situation, when not using a customizable system is that: writing a server that provides all of the services is impossible; writing applications that provide all of the services that the server does not support is impossible; writing applications that support all user interface views needed by the client is impossible. The good news are that we can identify a set of core primitives and services needed by all the clients (API calls to AddOrder, AddRelease, etc), and we can also identify a set of core data types used by all the clients (like Order, Release, etc.). This way, the different client needs in terms of the views applied on the data can be considered a customization of the core data types with additional (and different) attributes.

III. Extensible Data Model

A possible solution to the problem that specific clients need specific data views can be to create a data model that enables the customization of its data types. This paper proposes such a dynamic model. In this model new types can be created and existing types can be changed by adding new fields to them. The values of these additional fields are computed based on rules specified by the clients.

ISBN: 978-988-19252-4-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

A. Step 1 – Customizable Data types

Our main requirement is to give the client the ability to customize existing data types. Let's look into the problem in more detail. Consider a class that has a set of attributes *expressed in terms of instance variables*. How can we add a new attribute to such a class? In the traditional approach, this is possible only by sub-classing our class with another one that contains the desired attributes. This method is both inflexible (the client app must be rebuilt each time a data type is customized) and impractical (if different users using the same client application need different customized data types, we end up with an ever growing class hierarchy).

The source of the problem is the fact that attributes are represented as instance variables. A better approach would be to *represent the attributes of an object as a collection*, rather than as its instance variables. In this approach, every object will have a dictionary called "attributes" that maps the name of the attribute to its value. An Attribute class will hold not only the name and value of the attribute but also its type. We are now able to create different instances of this class that have different attributes.

```
Type orderObject;
orderObject.AddAttribute(Attribute("ID", "string"));
orderObject.AddAttribute(Attribute("clientID", "string"));
Type releaseObject;
ReleaseObject.AddAttribute(Attribute("ID", "string"));
ReleaseObject.AddAttribute(Attribute("orderID", "string"));
```

In our application domain we have also collection objects that contain collections of orders, releases, etc. Using the above approach, every object in our collections would be represented as a set of three-tuples of (name-type-value). One can easily see that the name and type information is duplicated throughout the collection. The solution to this problem is to put the type information in the collection, because every object is of the same type. The Value attribute of the Attribute class is also removed because from now the collection is the one who manages the values.

However, this approach doesn't capture the fact that our application domain has different data types (Orders are not Releases) and those types can be related to each other (using foreign keys). We could subclass the "CollectionType" class to obtain different types. Relationships could be represented but using a special type of attributes ForeignKeyAttribute that contains both the name of the related type and the name of the related attribute.

The problem with this design arises from the fact that there are also simple objects that have the same type. We have not only order collections, but also orders and events that have orders as their content. To solve this problem, we could create a similar model for simple objects, too. However, in this way we are duplicating the type information and the relations between the types. For example, in the simple object diagram, we can have a ForeignKeyAttribute named "orderID" of a ReleaseType object that points to the "ID" attribute of an OrderType object. Likely, in the collection diagram, we will have a ForeignKeyAttribute named "orderID" of a ReleaseCollectionType object that points to the "ID" attribute of an OrderCollectionType object.

A more difficult problem is that each instance of a simple object contains the set of attributes it has. In case of the collections we know that they are unique, that is we have only one order table, one release table, and so on. But how can we guarantee that the simple object instances are also singletons? In this way, every instance of a release object would contain the list of attributes that data type has. The problem is *that we keep the type information in the objects*. This problem can be solved by *separating the type information from the actual instances*. We can create types as objects and let each simple object or collection object to have an attribute called "type" that points to an actual type object. This way we can specify types once and reuse them for multiple objects. This approach also eliminates the need of sub-classing the object and the collection hierarchy.

Using this model, we can create different type objects (one object for every type) and customize them by adding different attributes. For example, the construction of an Order type will look something like this:

```
// Create a new type called Order
Type Order;
Order.AddAttribute(Attribute("ID", "string"));
Order.AddAttribute(Attribute("clientID", "string"));
. . .
```

Now we are able to create different objects (both simple and collection objects) by specifying their type in the constructor:

```
// Create a simple order object of type Order
Object order(Order);
// Create a collection of orders
Collection orders(Order);
```

B. Step 2 – Rules

As presented, the object and collection classes have methods that get and set the values of different attributes. In our application domain, the core set of attributes in each type are set using well defined business rules. What happens with the values of the attributes added by the client? A generic client application cannot know the semantic meaning of the attributes added to the types. This meaning must be specified by the client by means of rules that compute the values of those attributes.

At this point, we have to apply a specialization in the design. There are attributes that are part of the core data model - we will call them native attributes. There are also attributes added by the clients – we will call them added attributes. The difference between them is that added attributes get their value by evaluating user specified rules while native attributes are computed based on hard coded business rules. Added attributes and their associated rules will be specified by the client. In order to do this, the client can use a configuration file or (a better approach) an utility application possibly with a graphical interface for customizing the data types and for specifying the rules.

The problem to resolve next is how to represent rules and functions in the system. The most obvious way to *represent rules and functions* is *with a language*. This requires therefore implementing the language. There are lots of ways to do this: use a compiler tied to the underlying machine, define a simple virtual machine and compile it the virtual machine or develop an interpreter. These implementation techniques trade off ease of implementation with speed of the final program.

Rules can be represented by a rule hierarchy that corresponds to the grammar that is being interpreted. There are subclasses that represent constants, attributes, table lookups, and perform arithmetic or aggregation operations.

ISBN: 978-988-19252-4-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Every component in the hierarchy must support an operation called "evaluate" that evaluates an instance of a given component. Using these components, we can build up program trees that can be later interpreted.

There are different types of rules. The primary distinction between them is that some of them generate a single value, while some of them generate multiple values.

Single value generator rules:

- A Constant rule always evaluates to the same value. This value is specified during construction.
- A CollectionElement rule evaluates the value of a given collection element's attribute (for example, the value of the "orderID" attribute of the 5th row in a releases collection).
- An EventElement rule evaluates an attribute of a given event.
- A TableLookup rule evaluates the value of an attribute from a given row in a given table.
- A BinaryOperation rule always has two operands. It evaluates by applying a binary operator to the evaluated values of its operands. One can see that the operands can theirselves be rules that generate single values.
- An AggregationRule applies an aggregation operator on its operands. The operands can be any types of rules. First the operands are evaluated, resulting a list of values. Then, the aggregation operator is used to obtain the final value.

Multiple value generator rules:

- MultipleTableLookup evaluates to a list of values. Each element in the list is the evaluation of the same attribute from a table but for different rows. The values of the rows for which the attribute is evaluated (the selector) are computed using another rule. This mechanism enables the implementation of the join operations.
- CollectionElements rule evaluates the same attribute for a collection object but for different rows. The values of the rows for which the attribute is evaluated (the selector) are computed using another rule. This mechanism enables the implementation of the join operations.

There can be many other rules not included in the class diagram. To make the language complete, we must identify all the representative rules by means of which every program can be implemented.

An important thing to consider is the fact that rules are always evaluated relative to a given context. This context contains additional "run-time" information needed for the evaluation of a rule. This context is passed to the root node of the program and is propagated down the three during the evaluation process. Every node can add additional information to the context, but a given node should not alter the original context. One of the most common kinds of contexts is a name space, which is usually represented as a dictionary that maps names to objects. If a rule needs to know the value of an attribute, it uses the context to read or to find that value.

For example, in the case of a CollectionElement rule, the element itself identifies the collection and the attribute that is evaluated. The only missing information is the row for which this evaluation should be made. One of the possibilities of specifying the identifier of the row is by inserting it into the context. During the evaluation, the CollectionElement rule will inspect the context for such a value, and if finds it, it will use it for the subsequent computation.

Let's consider a simple expression and build for it the program that represents it. The expression is:

```
2+3*5
```

The following section of code builds the program that describes this expression:

```
Constant c1(2), c2(3), c3(5);
BinaryOp op1(MULTIPLY, c2, c3);
BinaryOp op2(ADD, c1, op1);
```

To evaluate the value of the expression, one can execute the following instruction:

```
op2.Evaluate();
```

C. Step 3 – Relations

One can easily observe that there are actually two data models in the client application. One of them is the model we try to build, and the second is the built-in data model, which is hard coded in the implementation. In fact, *our data model is built on top of the built-in data model*. This is why our data model must be at least as powerful as the underlying one. The hard coded data model expresses relationships between different types (in a hardcoded manner). The consistency of the model is preserved by the operations. Our data model must also be able to express relations.

To model as close as possible the relational data model encoded in the existing built-in model, we *define relations in terms of foreign keys*. Foreign keys are attributes of data types that refer to other attributes in other data types.

IV. Implementation Issues

A. The Configuration File

There must be a place where the customized data types as rules are saved for later reuse. This can be a configuration file - a file consisting of several sections, each section holding a related group of declarations/statements/rules. By comparison CSQL [4] is not persisting rules.

One section would describe the additional fields of each data type. This section may look like this:

```
<Extensions>
 <Type Name="Order">
  <NewField Name="O1" Type="Integer">
   <Rule> // rule to compute O1
   </Rule>
  </NewField>
 <NewField Name="O2" Type="Float" Format="xxx.xx">
   <Rule> // rule to compute O2
   </Rule>
 </NewField>
</Type>
<Type Name="Release">
 <NewField Name="R1" Type="String" Length="25">
  <Rule> // rule to compute R1
  </Rule>
 </NewField>
</Type>
</Extensions>
```

ISBN: 978-988-19252-4-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

In the previous example, the Order data type is extended by two additional fields. The first field has the name "O1" and it is an integer. Second field is named "O2" and it is a float. For each new field there is a rule that computes its value. Rules are described in the next section.

The format of the file is selected randomly. The end user can manipulate the data model through an interface program that will hide the details of the configuration file.

Another section of the configuration file would describe the events that can occur and the fields that are evaluated every time a specific event occurs.

```
<Evaluations>
  <OnEvent Name="NameOfEvent1">
    <Evaluate Field="O1" Of="Order"/>
    <Evaluate Field="O2" Of="Order"/>
    <Evaluate Field="R1" Of="Release"/>
  </OnEvent>
  <OnEvent Name="NameOfEvent2">
    . . .
  </OnEvent>
</Evaluations>
```

B. The Language for Rules

The language for rules must be as simple as possible, but in the same time it must be powerful enough to enable the client to express a large category of operations. Another important issue is that rules must be intuitive. This means that some kind of implicit information must exist every time a rule is evaluated. The writer of the rule must not specify this information in order to enable the rule to be evaluated.

Simple rules enable the computation of a given attribute's value by using the different mathematical operators, constants, table lookups, collection elements and event elements.

```
Order.O1 = 100
Order.O2 = SUM(Release.R5)
Order.O3 = Order.Sent – Order.Done
Order.O4 = AVG(TAB1.Field1)
Order.O5 = RTP.ASK
```

Complex rules enable the specification of the join operation. An example of a complex rule is:

```
Order.O6 = SUM [Order.O1 = Order.O5] Order.O3
```

This rule computes the value of the O6 field of the Order data type as the sum the O3 fields of the same data type. Between brackets a self-join operation is expressed. The equivalent SQL statement for this rule would be:

```
SELECT SUM Order1.O3
FROM Order1, Order2
WHERE Order1.O1 = Order2.O5
```

V. The Evaluation Model

This paper presents a model that can serve as the *underlying data model* for both client and server applications. The main characteristics of the model are:

- It captures the existing hard coded relations between the data types
- It enables the dynamic customization of existing data types
- It enables the computation of different attributes' values based on user-defined rules

Another important aspect of the model is that, while it is probably *not complete*, it is *stable*. This means that additions to the model can be made in order to achieve completeness without the need to change the main design issues. Just to remember, these main decisions are:

- Keeping type information apart from object instances
- Representing rules as a language for which an implementation grammar draft exist

A. Case Studies

We can present case studies related to the ways these evaluation methods can be used to map the required operations on the following type of containers:

- retrieval of data from DBs together with operations on this data
- operations on data acquired from RTP (Real Time Pricing) services.

As can be seen, the following cases are only two specific ways to use the model we have described.

DBs Representing the DB data as collections the following operations are available:

- retrieval of individual fields
- operations on individual and column type data
- operations on composite type of data (JOIN)

RTP Representing the RTP data as collections we have access to the same set of operations as described for DBs.

VI. Workflow

This section describes different run-time scenarios explaining the behavior of the system from the implementation point of view. We will consider client applications. In an extensible application scenario, since additional columns should be displayed on the GUI, the display function will be extended to display the additional columns as well.

A. Initialization

Every app will generate at start-up the Type instances for every data type. For each Type object, the core attributes will be added. Finally, the relations between these data types are established.

B. Parsing

We need a parser to read, validate and interpret the configuration file and data definition file at the start-up of the app. We may optionally implement some logic in the parser, like verification of field names used for additional fields, validity of the evaluation order, validity of the rules, type matching, etc.

While the parser reads the configuration file, it customizes the existing Type objects by adding new attributes to them. For every added attribute, a program tree is generated for evaluating its value.

C. Reacting on Events

Every time an event occurs, the app will make some processing and after that it will pass the control to the rule interpreter. The interpreter evaluates each rule and after that returns the control to the main app.

Every event carries some data that has a given type. We will denote this type with the identifier EventType. The attribute for which a rule is evaluated will be called TargetAttribute and the type of the collection for which this

ISBN: 978-988-19252-4-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

attribute is evaluated will be referred as TargetCollection. They will be referred generically as the Target of the operation. The operand of a rule will be generically called Operand. The attribute of an Operand that participates in the computing process will be referred as OperandAttribute and the type of the collection from which this attribute comes will have the name OperandType.

For example, in the rule below

```
Order.O1 = AVG (Release.R5)
```

that is evaluated when a ReleaseAdded event comes, we will have:

```
EventType: Release
Target: Order.O1
TargetAttribute: O1
TargetType: Order
Operand: Release.R5
OperandAttribute: R5
OperandType: Release
```

D. Determining the Target

Let's take a closer look to this process. As it was stated before, the rules are specified in an intuitive manner. In the rule:

```
Order.O1 = AVG (Release.R5)
```

there is no information regarding the row in the Order collection for which the evaluation is made. That is, the rule only states the name of the collection and the name of the attribute. The extra information must come from the event itself. Consider for example a ReleaseAdded event. When the app receives such an event, the event also contains some additional data, in our case a release. But every release contains the ID of the order for which it was generated. We can use this information to identify the specific row in the Order collection for which the O1 field is computed. But how can the program know what are the fields in the event that uniquely identify the target? There are three cases described below. In any case, the same restriction holds: *there must be a relation between the type of the object associated with an event and the type of the collection for which an attribute is evaluated*. The two types can be identical or the EventType must directly or indirectly refer the TargetType.

1. EventType is the same as TargetType. In this case we can use the primary key fields of the common type in order to determine the row for which TargetAttribute is evaluated.
2. EventType directly refers the TargetType. In this case we can use those foreign key attributes from the EventType that point to the TargetType, so the identification is done.
3. EventType indirectly refers the TargetType. In this case we can use a recursive algorithm to determine the target row. At each step we have a current collection type for which we determine those foreign key attributes that directly or indirectly point to the TargetType. Based on the values of these attributes, we can determine the row in the directly referred collection. Then, the type of this collection becomes the current type, and the process is repeated until we reach the TargetType.

After the computation of TargetAttribute's row, this value is inserted into the context that will be passed to the interpreter to evaluate the rules.

E. Evaluating the Rules

Evaluating a rule means evaluating the program tree associated with that rule. In order for this to be done, the rules expressed in the configuration files must first be transformed in a convenient format that fits into our rule class diagram.

Consider the following rule:

```
Order.O1 = AVG (Release.R5)
```

This rule can be rewritten as:

```
Order.O1 = SELECT AVG (R5)
FROM Release
WHERE OrderId = Order.Id
```

One can observe that at the moment of the evaluation of this rule, Order.Id is already computed, i.e. it is a constant.

Now let's take a look at our internal rule grammar. This grammar can be followed in the rule class diagram. One component of this diagram is the CollectionElements class. This class contains the following important instance members:

- A reference to a collection type on which the rule operates. In the above example, this reference can be the Type object associated with the Release type. This type name is the operand of the FROM part of the statement.
- A reference to the attribute of the collection whose values are extracted. In the above example, this will be the string "R5" that is the name of the attribute. This attribute is the operand of the SELECT part of the statement.
- A list of attributes that must satisfy some conditions in order to make that row to participate in the evaluation process. In the above example, this will be the string "OrderId". This is attribute is the first operand of the WHERE part of the statement.
- A reference to a set of rules that compute the permitted values of the attributes in the WHERE part of the statement. In the above case, this will be a reference to a single Constant rule object.

Evaluating an instance of the CollectionElements class means executing the following statement:

```
SELECT selectedAttributeName
FROM collectionType
WHERE selectorAttributeNames = selectorValues
```

For each attribute name in the selectorAttributeNames list there must be a corresponding rule in the selectorValues.

One should note that the selectorValues list is a list of Rule objects! Considering the fact that a CollectionElements is a Rule, this means that we can use as a selector value the evaluation of another CollectionElements object. This facility enables the specification of the join operation between collections.

Consider the example:

```
Order.O6 = SUM [Order.O1 = Order.O5] Order.O3
```

ISBN: 978-988-19252-4-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

As we could see, the equivalent SQL statement is:

```
SELECT SUM Order1.O3
FROM Order1, Order2
WHERE Order1.O1 = Order2.O5
```

This rule can be further rewritten as:

```
SELECT SUM Order.O3
FROM Order
WHERE Order.O1 =
( SELECT Order.O5 FROM Order )
```

In this case, the CollectionElements object that can be constructed will have the following fields:

- The collectionType field will contain a reference to an order type object
- The selectedAttributeName field will contain the attribute name "O3"
- The selectorAttributeNames list will contain only the "O1" field name
- The selectorValues list will contain a single object. This object will be another instance of the CollectionElements class.

This is the way a join operation is represented.

VII. Developments

Developments may be considered using ideas proposed or available in implementations such as DBCache[3], FastDB[4] and Sprint [10].

A. Extensible Data Model on both the Server and Client

The extensible data model may be used both on the server and client side. At start-up the server can read the data model from a configuration file or, more generally, from a "data model repository". When a client logs in to the system, it will also receive the same data model from the repository.

B. Extensible Service Model

The server can be considered as an environment in which state machines corresponding to each data types are instantiated and executed. For example, in the case of the Order Execution Server, every data type has a corresponding state diagram. When a CancelOrder request arrives from a client, the server makes the state transition of the corresponding order object from the "ADDED" state into the "CANCELLED" state.

Every service offered by the server can be described as a transition between two states for a given object (order, release, etc.). New services can be simply added by creating new states and new transitions between states. Existing services can be refined by modifying the guards in the transitions and by modifying the actions performed during transitions. Every operation request can be then mapped into a state transition generating event in the corresponding state machine.

C. The Application Programming Interface

The API must support both the extensible data model as well as the extensible service model. In order to achieve this, there can't be a fixed set of service request functions, like AddOrder, AddRelease, etc. Instead, the API needs to support a service request primitive named `Request` that will request a specific operation form a service. The general format of this primitive can be:

```
Request("OperationName", parameterObject)
```

In the case of the Order Execution Server, the requested operations could be AddOrder, AddRelease, etc.

D. Extensible Validation Model

Every time a client requests an operation from a service, there are two validation aspects to consider:

- Whether the client is authorized to execute the operation requested
- Whether the preconditions of the requested operation are satisfied

The second aspect directly relates to the business rules of the application. This type of validation can be easily modeled in the server side state machine by the means of guards. Every transition has an associated guard that must evaluate to true in order to let the transition to proceed. Business related validations could be described as guards in a rule-like language.

The first aspect needs a separate validation model. This model could enable the specification of roles. Every role could have a set of permitted events that can be generated by entities that have that role. Also, every role could have a set of accessible data types and a set of restrictions.

VIII. Conclusion

EDM is an original high performance framework that allows a flexible customization of solutions that may be used in the capital markets electronic exchange interactions domain. Specific consideration has been given to the requirements of building an extensible service model as well as an extensible validation model, these components needs to be built in a way that enables co-operation between them. The components presented are based on frameworks that enable the building of highly extensible and configurable new generation applications.

References

[1] Jerry D. Baulier et all, The DataBlitz "Main-Memory Storage Manager: Architecture, Performance, and Experience", 1998, The VLDB Journal
[2] P. A. Boncz et all . MonetDB/XQuery: "A Fast XQuery Processor Powered by a Relational Engine". Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, IL, USA, June 2006.
[3] CSQL – DBCache http://csql.sourceforge.net/
[4] FastDB: a main-memory database object-relational database system, Available: http://www.garret.ru/~knizhnik/fastdb/FastDB.htm
[5] FOX, Fusion Order eXecution platform, internal documentation, 1999
[6] I. Lee, H. Y.Yeon, T. Park, "A New Approach for Distributed Main Memory Database Systems: A Causal Commit Protocol", IEICE Trans. Inf. & Syst., Vol.ES7, No.1 January 2004
[7] S. Manegold, P. A. Boncz, and M. L. Kersten. "Optimizing Main-Memory Join on Modern Hardware", IEEE Transactions on Knowledge and Data Engineering (TKDE), Vol.14, No.4, pp.709–730, July 2002
[8] H. Garcia-Molina, K.Salem, "Main Memory Database Systems: An Overview", IEEE Transactions On Knowledge And Data Engineering, Vol. 4, No. 6, Dec 1992
[9] RoyalBlue Fidessa http://www.fidessa.com/
[10] Sprint, Adaptive data management for in-memory database clusters, http://www.inf.unisi.ch/projects/sprint/
[11] TimesTen, http://www.timesten.com

ISBN: 978-988-19252-4-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)