Project Number 780251

D3.3 TyphonML to TyphonDL Model Transformation Tools

Version 1.0
23 December 2019
Final

Public Distribution

University of L'Aquila

Project Partners: **Alpha Bank**, **ATB**, **Centrum Wiskunde & Informatica**, **CLMS**, **Edge Hill University**, **GMV**, **OTE**, **SWAT.Engineering**, **The Open Group**, **University of L′Aquila**, **University of Namur**, **University of York**, **Volkswagen**

Every effort has been made to ensure that all statements and information contained herein are accurate, however the TYPHON Project Partners accept no liability for any error or omission in the same.

© 2019 Copyright in this document remains vested in the TYPHON Project Partners.

Project Partner Contact Information

Alpha Bank Vasilis Kapordelis 40 Stadiou Street 102 52 Athens Greece Tel: +30 210 517 5974 E-mail: vasileios.kapordelis@alpha.gr	**ATB** Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany Tel: +49 421 22092 0 E-mail: scholze@atb-bremen.de
Centrum Wiskunde & Informatica Tijs van der Storm Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 9333 E-mail: storm@cwi.nl	**CLMS** Antonis Mygiakis Mavrommataion 39 104 34 Athens Greece Tel: +30 210 619 9058 E-mail: a.mygiakis@clmsuk.com
Edge Hill University Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk	**GMV Aerospace and Defence** Almudena Sánchez González Calle Isaac Newton 11 28760 Tres Cantos Spain Tel: +34 91 807 2100 E-mail: asanchez@gmv.com
OTE Theodoros E. Mavroeidakos 99 Kifissias Avenue 151 24 Athens Greece Tel: +30 697 814 7618 E-mail: tmavroeid@ote.gr	**SWAT.Engineering** Davy Landman Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 633754110 E-mail: davy.landman@swat.engineering
The Open Group Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org	**University of L′Aquila** Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it
University of Namur Anthony Cleve Rue de Bruxelles 61 5000 Namur Belgium Tel: +32 8 172 4963 E-mail: anthony.cleve@unamur.be	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Volkswagen Behrang Monajemi Berliner Ring 2 38440 Wolfsburg Germany Tel: +49 5361 9-994313 E-mail: behrang.monajemi@volkswagen.de	

Document Control

Version	Status	Date
0.1	Document outline	28 November 2019
0.2	First draft	7 December 2019
0.7	First full draft	16 December 2019
0.8	Further editing draft	18 December 2019
1.0	Final updates after partner reviews	23 December 2019

Table of Contents

1 Introduction **1**

1.1 Structure of the deliverable 1

2 Explicit management of functional and non-functional requirements in TyphonML models **2**

2.1 Overview of functional and non-functional data requirements 5

2.1.1 Functional Requirements 5

2.1.2 Non-functional Requirements 6

2.2 Specification of data requirements and their feasibility checks 7

2.3 Generation of TyphonML data mappings 12

3 The TyphonML to TyphonDL transformation **15**

4 Conclusions **18**

Executive Summary

This document presents the techniques and tools that have been developed in collaboration with WP2 to support the specification of TyphonML models in a consistent way with the requirements that the developer wants to achieve with the system being modeled. An enhancement of the TyphonML language and supporting tools has been needed to enable the generation of TyphonDL-based deployment configurations, which are able to satisfy functional and non-functional requirements defined at the TyphonML level.

1 Introduction

Designing and deploying hybrid data persistence architectures that involve combination of relational and NoSQL databases is a complex, and error-prone task. TyphonML (being defined and developed in WP2) provides developers with a domain specific language enabling the specification of the conceptual entities that need to be managed by the application being developed. Moreover, such entities are subsequently mapped to the different kinds of available DB systems.

In the context of TYPHON, WP3 is focusing on polystore deployment aspects by designing and developing the TyphonDL language, which provides concepts that lie at an abstraction level between that of TyphonML, and that of specific data stores and virtual machine configuration technologies. Model-to-text transformations that consume TyphonDL models and produce installation and configuration scripts targeting the selected virtual machine image assembly technologies (e.g. Chef, Docker) are also in the scope of WP3. In this document, we focus on the developed approach and supporting tools for managing the guided specification of TyphonML models and the corresponding automated generation of TyphonDL specifications. In particular, in this document we present results of WP2 and WP3 related to the following task (from the TYPHON DoW):

> *Task 3.3: Transformation of TyphonML Design Models to Deployment (TyphonDL) Models.* This task will produce automated transformations for mapping polystore design (TyphonML) models to deployment (TyphonDL) models, which will specify the way to assemble virtual machines (or instances) that contain all the software components needed to support a polystore (e.g. operating system, system services, configured persistence back-ends). The transformations between TyphonML and the TyphonDL models will be developed using contemporary model transformation languages like ATL [4] and ETL [6].

Thus, this document presents the tools that have been developed in collaboration with WP2 to support the specification of TyphonML models in a consistent way with the requirements that the developer wants to achieve. Such TyphonML models are subsequently consumed to generate TyphonDL specifications so to eventually obtain deployment configurations, which are able to satisfy functional and non-functional requirements defined at the TyphonML level.

1.1 Structure of the deliverable

The structure of the deliverable is as follows: the functional and non-functional requirements that modelers can specify at the level of conceptual entities are overviewed in Section 2 together with the conceived tools that are defined in the context of WP2 to enhance TyphonML accordingly. Section 3 describes the model transformation tools, which are able to generate TyphonDL specifications from source TyphonML ones. Section 4 concludes the document and provides an overview of the next steps.

2 Explicit management of functional and non-functional requirements in TyphonML models

As described in deliverable D2.4, the TyphonML language and supporting tools permit modelers to specify both the conceptual entities of the application being developed and how they have to be mapped on the available DB infrastructures. Thus, depending on the wanted functional and non-functional properties, modelers have the responsibility of properly considering the appropriate technologies. For instance, Listing 1 shows an explanatory TyphonML model specifying a simple e-commerce system. Different database systems have been considered for storing the modeled conceptual entities. For instance, products, orders, and credit cards, are modeled so to be mapped on a relational database since it permits to store data by protecting them by strong ACID transactions. Such a strong requirement is not required e.g., to manage reviews and comments, which are consequently persisted as nested documents in a document database. Data about products that customers tend to buy together are stored in a graph database.

As previously mentioned, modelers have to pay particular attention when mapping the modeled conceptual entities to concrete databases. By still referring to the explanatory e-commerce example, if modelers wrongly map the *Order* entity e.g., to a document database, the ACID transaction requirement that was expected to be satisfied for that conceptual entity would not be met. In other words, the numerous existing database technologies can impede the well-informed selection of the data store, which is the most appropriate with respect to the wanted functional and non-functional requirements that modelers would like to address for the system being modeled.

```
1  -- Conceptual entities
2  entity Review{
3          product -> Product[1]
4  }
5
6  entity Product{
7          name : String
8          description : String
9          review :-> Review."Review.product"[0..*]
10         orders -> Order[0..*]
11         photo : Jpeg;
12 }
13
14 entity Order{
15         date : Date
16         totalAmount : Int
17         products -> Product."Product.orders"[0..*]
18         users -> User."User.orders"[1]
19         paidWith -> CreditCard[1]
20 }
21
22 entity User{
23         name : String
24         surname : String
25         comments :-> Comment[0..*]
26         paymentsDetails :-> CreditCard[0..*]
27         orders -> Order[0..*]
28 }
29
```

```
30 entity Comment{
31         freetext content [SentenceSegmentation,TextClassification]
32
33         responses :-> Comment[0..*]
34 }
35
36 entity CreditCard{
37         number : String
38         expiryDate : Date
39 }
40
41 -- Specification of data mappings
42 relationaldb RelationalDatabase{
43         tables{
44                 table {
45                         OrderDB : Order
46                                 index orderIndex {
47                                         attributes ("Order.date")
48                         }
49                 }
50                 table {
51                         UserDB : User
52                         index  userIndex{
53                         attributes ("User.name")
54                         }
55                 }
56                 table {
57                         ProductDB : Product
58                         index productIndex{
59                                 attributes ("Product.name")
60                         }
61                 }
62                 table {
63                         CreditCardDB : CreditCard
64                         index creditCardIndex{
65                                         attributes ("CreditCard.number")
66                                 }
67                         }
68                 }
69         }
70 }
71
72 documentdb ReviewCommentDB{
73         collections{
74                 ReviewsCol : Review
75                 CommentsCol : Comment
76         }
77 }
78
79 graphdb ConcordanceDB  {
80         nodes {
81                 node ProductNode!Product {
82                         name = "Product.name"
```

```
83                  }
84          }
85          edges {
86                  edge concordance {
87                          from ProductNode
88                          to ProductNode
89                          labels {
90                                  weight:int
91                          }
92                  }
93          }
94  }
```

Listing 1: Sample TyphonML specification

To mitigate the difficulties of properly mapping conceptual data entities to the most appropriate database systems, a novel tool supported approach has been conceived as shown in Fig. 1. In particular, the TyphonML language has been enhanced in order to permit modelers to annotate conceptual entities with functional and non-functional requirements. Thus, the tagged conceptual entities (referred with TyphonML$_{CE}$ hereafter) are given as input to the *Feasibility Checks* tool, which performs statical analysis on the given input in order to check if the given requirements can be met with respect to the available database technologies. In case multiple solutions are possible, modelers is asked to select one of those. It can happen that the given requirements are too strict and cannot be met as they are. Thus, in such cases, modelers are asked to relax some of the given requirements (if possible). After such a phase, the *Mapping Generator* tool is used to generate a complete TyphonML specification, which includes the mappings that best fit the given requirements (see TyphonML$_{DBMap}$ block in Fig. 1). The generated complete TyphonML specification is in turn given as input to the *TyphonDL Generator* tool, which generates the deployment specification w.r.t. the given functional and non-functional requirements.

The remaining part of the section is organized as follows: Sec. 2.1 makes an overview of functional and non-functional requirements that make sense in the context of data modeling and data management. Section 2.2 introduces the enhancements that have been operated to the TyphonML language and tools for enabling the specification of data requirements and their feasibility checks. In case of multiple or empty solutions, human interventions are needed to subsequently generate the TyphonML data mappings as described in Sec. 2.3. The generation of TyphonDL specifications out of input TyphonML models is described in the next section.

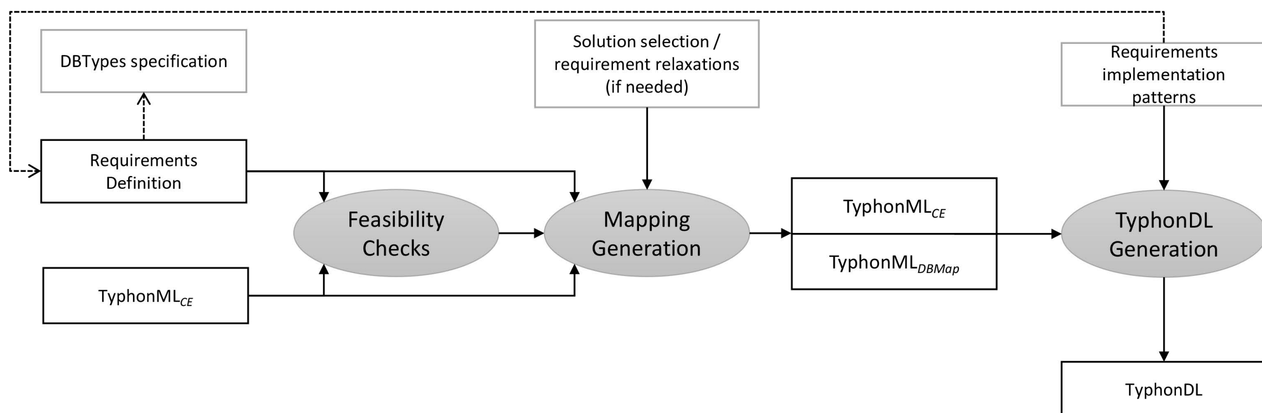


Figure 1: Overview of the conceived data mapping generation

2.1 Overview of functional and non-functional data requirements

As described in [3] relational database systems have reached an unmatched level of reliability, scalability, and support through decades of development. However, in some application areas (e.g., social networks and IoT systems) data have reached a vast amount so that they cannot be stored in traditional database solutions. Horizontal scalability and high availability are just examples of requirements that are satisfied by NoSQL databases at the price of sacrificing query capability and consistency guarantees of relational databases. Thus, when developing data-intensive applications, it is necessary to bare clearly in mind how data will be stored, consumed, and consequently the functional and non-functional requirements that need to be met.

According to [3] it is possible to classify database systems with respect to at least the functional and non-functional requirements described below.

2.1.1 Functional Requirements

The main functional requirements that can be used to describe a DBMS are overviewed below.

Scan queries: The DBMS implements plans to execute queries efficiently. To this end, the query to be executed is processed so to identify the steps that the DBMS has to perform for executing the query.

ACID transactions: The operations that are managed by the considered DBMS will have the following properties

- *Atomic*: the operation will either work or not work (not half-work);
- *Consistent*: by applying the same operation over and over again given the same starting state, the result would be always the same;
- *Isolated*: the operation will not be impacted by events that are not related to it;
- *Durable*: the result of the performed opearation will remain unless the resulting data is modified by another ACID transaction.

Conditional Writes: This is a relevant requirement that needs to be satisfied for instance in case of banking applications where over-counting or under-counting are not admitted. Conditional writes ensure that no other requests interfere with the operation being operated. In other words, conditional writes are a form of lightweight transactions.

Joins: The DBMS is able to fetch data from two or more entities and combine them to appear as a single set of data. Values that are common to the considered entities are used to combine stored data accordingly.

Sorting: The DBMS implements techniques able to sort items in ascending or descending order.

Full-text search: The DBMS implements searching and indexing mechanisms able to retrieve stored texts matching the words in the given search query.

Aggregation and analytics: The DBMS permits to aggregate data coming from structured or semi-structured sources into additional columns for making them easier to query and to analyse.

	Functional Requirements						
	Scan Queries	ACID	Conditional Writes	Joins	Sorting	Full-text Search	Analytics
MongoDB	✓		✓		✓	✓	✓
Redis	✓	✓	✓				
HBase	✓		✓		✓		✓
Riak							✓
Cassandra	✓		✓		✓		✓
MySQL	✓	✓	✓	✓	✓	✓	✓

Table 1: Comparison of MongoDB, Redis, HBase, Riak, Cassandra and MySQL with respect to functional requirements [3]

In [3] authors analysed different DBMSs (i.e., MongoDB[1], Redis[2], HBase[3], Riak[4], Cassandra[5], and MySQL[6]) and classified them with respect to the functional requirements previously overviewed. Table 1 shows the result of such an analysis by showing the functional requirements that are supported by the considered technologies.

2.1.2 Non-functional Requirements

Concerning non-functional requirements, according to [3] DBMSs can be classified with respect to the ones described below.

Data Scalability: It refers to the cabability of the considered DBMS to scale up or down depending on the amount of the data it has to manage without sacrificing performance.

Write Scalability: It refers to the capability of the considered DBMS to manage write-intensive workloads without sacrificing performance. An example of technique that is employed to support write scalability is sharding (partitioning) data across different nodes (shards) in the system. A simpler approach, even though not always enough, is adding more CPU and RAM to some levels of the considered system.

Read Scalability: Similarly to write scalability, it refers to the capability of DBMSs to manage read-intensive workloads without sacrificing performance. The adoption and management of data replicas is a possible way to achieve read scalability.

Elasticity: It is related to scalability and according to [1] it is *"the ability to deal with load variations by adding more resources during high load or consolidating the tenants to fewer nodes when the load decreases, all in a live system without service disruption"*.

Consistency: The considered DBMS implements techniques that permit to maintain data in a predictable, and consistently safe, state.

Write Latency: The system provides the means to reduce the overall write latency, thereby improving runtime performance. The adoption of replicas is a way to reduce write latency.

[1] `https://www.mongodb.com/`
[2] `https://redis.io/`
[3] `https://hbase.apache.org/`
[4] `https://riak.com/`
[5] `http://cassandra.apache.org/`
[6] `https://www.mysql.com/`

	Non-Functional Requirements										
	Data Scalability	Write Scalability	Read Scalability	Elasticity	Consistency	Write Latency	Read Latency	Write Throughput	Read Availability	Write Availability	Durability
MongoDB	✓	✓	✓		✓	✓	✓	✓	✓		✓
Redis			✓		✓	✓	✓	✓	✓		✓
HBase	✓	✓		✓	✓	✓		✓			✓
Riak	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
Cassandra	✓	✓	✓	✓		✓		✓	✓	✓	✓
MySQL			✓		✓						✓

Table 2: Comparison of MongoDB, Redis, HBase, Riak, Cassandra and MySQL with respect to non-functional requirements [3]

Read Latency: Similarly to write latency, the system provides the means to reduce the overall latency concerning read operations, thereby improving runtime performance. The adoption of in-memory storage is an example of technique that can be employed to reduce read latency.

Write Throughput: It refers to the overall capability of the considered DBMS to write data. It is possible to increase write throughput e.g. by employing memory caches and append-only storage that tries to maximize throughput by writing sequentially [3].

Read Availability: It is related to the availability of the system for read operations. Thus, it is the probability that a read operation is successful in the given configuration. The adoption of replicas is a way to increase such a probability.

Write Availability: Similarly to the read availability, it is the probability that a write operation is successful in the given configuration.

Durability: It is referred to how strongly persistent the stored data is in the event of some kind of catastrophic failure within the store. Examples of a catastrophic failure are power outages, disk crashes, physical memory corruption, or even fatal application programming errors.[7]

Similarly to the functional requirement comparison shown in Table 1, in [3] authors analysed different DBMSs and classified them with respect to the non functional requirements previously overviewed. The result of the performed analysis is shown in Table 2.

2.2 Specification of data requirements and their feasibility checks

To enable the specification of data requirements, TyphonML has been enhanced with the support of dedicated tags as shown in the e-commerce example in Listing 2. In the shown explanatory example, the developer has specified that for the conceptual entity `Review` the functional requirement `fulltextsearch` should be supported by the wanted mapping (see the tag `@fr[fulltextsearch]`). Non functional requirements have been also specified for the same entity by means of the `@nfr` tag. In particular, the `Review` entity shall be managed by a DBMS supporting `readscalability` and `readavailability`.

```
1 @fr[fulltextsearch]
2 @nfr[readscalability,readavailability]
3 entity Review {
4         id: String
5         content: String
```

[7] https://docs.oracle.com/en/database/other-databases/nosql-database/19.3/java-driver-kv/setting-synchronization-based-durability-policies.html

```
 6          product -> Product[1]
 7  }
 8
 9  entity Product {
10          id: String
11          name: String
12          description: String
13          orders -> OrderProduct[0..*]
14          review:-> Review."Review.product"[0..*]
15  }
16
17  @nfr[consistency]
18  entity OrderProduct {
19          id: String
20          date: Date
21          totalAmount: int
22          products -> Product.products[0..*]
23          users -> User."User.orders"[1]
24          paidWith -> CreditCard[1]
25  }
26
27  @fr[sorting]
28  @nfr[consistency]
29  entity User {
30          id: String
31          name: String
32          comments -> Comment[0..*]
33          products -> Product.products[0..*]
34          paymentsDetails :-> CreditCard[0..*]
35          orders -> OrderProduct[0..*]
36  }
37
38  @fr[fulltextsearch]
39  @nfr[readscalability, readavailability]
40  entity Comment {
41          id: String
42          content: String
43          responses :-> Comment[0..*]
44  }
45
46  @nfr[consistency]
47  entity CreditCard {
48          id: String
49          number: String
50          expiryDate: Date
51  }
```

Listing 2: Specification of tagged conceptual entities

The enhancements that have been operated to the TyphonML language can be exploited also by means of the graphical editor as shown in Fig. 2

The functional and non-functional requirements that can be specified in a TyphonML model are defined in a dedicated *Requirement Definition* model conforming to the metamodel shown in Fig. 3. Such a model is

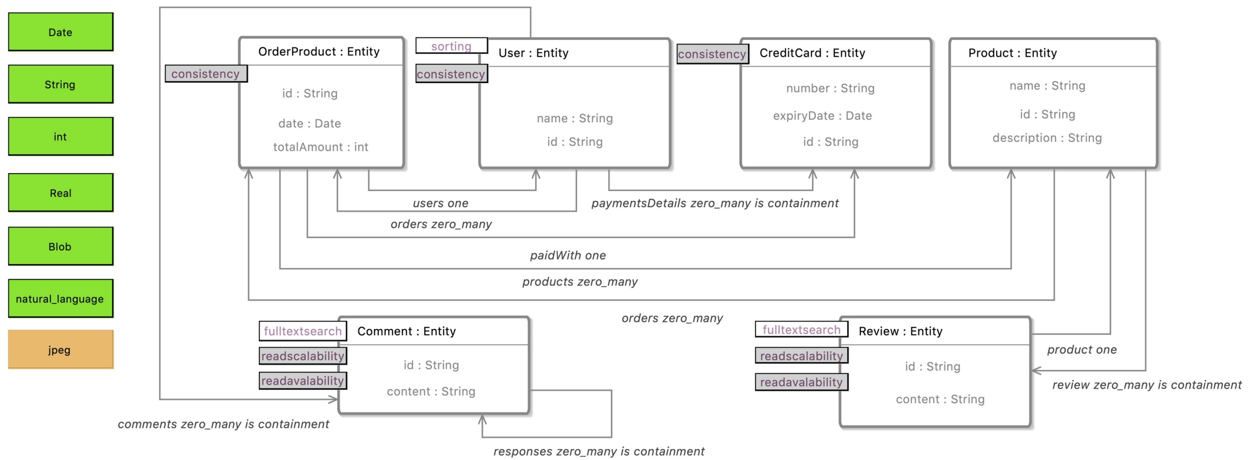


Figure 2: TyphonML conceptual model with functional and non-functional tags

defined once for all and can be extended when the support for new database types is added. According to Fig. 3, a requirement definition model consists of the specification of functional and non-functional requirements. Additionally, each supported database type is also specified by linking it to the supported requirements. For instance, the requirement definition model shown in Fig. 4 encodes the analysis shown in Table 1 and Table 2. For example, consistently with what has been presented in [3], *MongoDB* is able to support the *sorting*

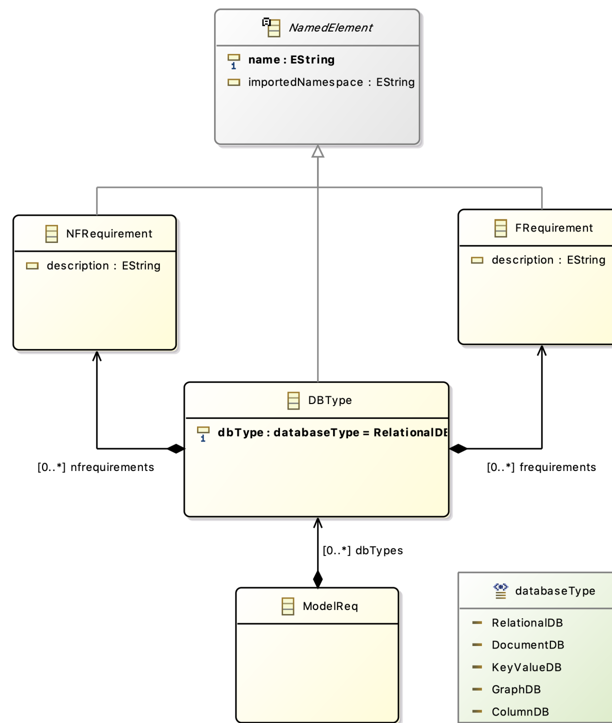


Figure 3: The *Requirement Definition* metamodel

Figure 4: An example of requirement definition model conforming to the metamodel in Fig. 3.

functional requirement, further than *datascalability*, *readlatency*, and *durability* among others. It is important to remark that the requirement definition model is not fixed and can extended when the set of supported database technologies change.

As previously mentioned, once the conceptual entities have been specified as given in the example shown in Listing 2, a *feasibility check* is performed. In particular, for each modeled entity, TyphonML tools identify the database types supporting all the specified functional and non-functional requirements. Such a feasibility check is implemented by means of Epsilon Object Language (EOL)[8] queries as shown in Listing 3. EOL is an imperative programming language for creating, querying, and modifying EMF models. The primary aim of that language is to provide a reusable set of common model management facilities, atop which task-specific languages can be implemented. However, EOL can also be used as a general-purpose stand-alone model management language for automating tasks that do not fall into the patterns targeted by task-specific languages.

[8]`https://www.eclipse.org/epsilon/doc/eol/`

The `feasibilityCheck` operation as defined in Listing 3 gets executed during the mapping generation phase, which is detailed in the next section.

```
1  operation TyphonML!Entity feasabilityCheck(): List<Map>{
2  var mapRank = new List<Map>;
3
4  //For each DBType
5  for(dbType in TyphonMLReq!DBType.allInstances()){
6          var mapDBWithFNF = new Map<TyphonMLReq!DBType, mapFNFRequirements>;
7          var mapFNFRequirements = new Map<functionalList, nFunctionalList>;
8          var functionalList = new Map<Integer, List<String>>;
9          var nfunctionalList = new Map<Integer, List<String>>;
10         //Check Entity Functional Tags
11         for(fr in self.functionalTags){
12                 var fCount = 0;
13                 var functionalStringList = new List<String>;
14                 for(fRname in dbType.frequirements){
15                         if(fRname.name = fr.name){
16                                 functionalStringList.add(fr.name);
17                                 fCount++;
18                         }
19                 }
20                 functionalList.put(fCount, functionalStringList);
21         }
22         //Check Entity Not Functional Tags
23         for(nfr in self.nfunctionalTags){
24                 var nfCount = 0;
25                 var nfunctionalStringList = new List<String>;
26                 for(nfRname in dbType.nfrequirements){
27                         if(nfRname.name = nfr.name){
28                                 nfunctionalStringList.add(nfr.name);
29                                 nfCount++;
30                         }
31                 }
32                 nfunctionalList.put(nfCount, nfunctionalStringList);
33         }
34         //Suggest DBType only if there are some functional or non functional matches
35         if(not functionalList.isEmpty and not nfunctionalList.isEmpty){
36                 mapFNFRequirements.put(functionalList, nfunctionalList);
37                 mapDBWithFNF.put(dbType,mapFNFRequirements);
38                 mapRank.add(mapDBWithFNF);
39         }
40 }
41
42 return mapRank;
43 }
```

Listing 3: EOL fragment of the feasibility checker

In case of many possible solutions, the modeler is provided with all of them so that the final selection can be done. In case of no possible solutions, the modeler is still provided with the list of database types that are ordered with respect to the number of covered requirements. By looking at such a list, modeler can decide if some requirement can be relaxed and modify TyphonML$_{CE}$ accordingly. For instance, concerning the conceptual entity `Review` as specified in Listing 2 the feasibility check tool gives the following result as output:

- **MongoDB**: 1 fr [fulltextsearch], 2 nfr [readscalability, readavailability]
- **Riak**: 1 fr [fulltextsearch], 2 nfr [readscalability, readavailability]
- **Cassandra**: 1 fr [fulltextsearch], 2 nfr [readscalability, readavailability]
- **MySQL**: 1 fr [fulltextsearch], 1 nfr [readscalability]
- **Redis**: 0 fr [], 2 nfr [readscalability, readavailability]
- **HBase**: 0 fr [], 0 nfr []

This means that it is possible to satisfy all the requirements given for the `Review` entity, by managing it with MongoDB, Riak, or Cassandra. The usage of MySQL, Redis, or HBase to store `Review` data would not satisfy all the functional and non-functional requirements that the modeler has specified for it.

2.3 Generation of TyphonML data mappings

Once a set of database systems has been identified to satisfy the requirements of all the modeled conceptual entities, the corresponding mappings are generated. The TyphonML data mapping generation has been developed by relying on the Epsilon Object Language and on the Epsilon Validation Language (EVL)[9] provided by the Epsilon platform [5]. In particular, models are analysed by a set of checks each devoted to the discovery of a potential mapping between a certain entity and a possible database.

EVL is a validation language built on top of EOL and provides a number of features such as support for detailed user feedback, constraint dependency management, semi-automatic transactional inconsistency resolution and (as it is based on EOL) access to multiple models of diverse metamodels and technologies. The aim of EVL is to contribute model validation capabilities to Epsilon. More specifically, EVL can be used to specify and evaluate constraints on models of arbitrary metamodels and modelling technologies. EVL also supports dependencies between constraints (e.g., if constraint A fails, the constraint B cannot be evaluated), customizable error messages to be displayed to the user and specification of fixes (in EOL) which users can invoke to repair inconsistencies. Also, as EVL builds on EOL, it can evaluate inter-model constraints (unlike OCL). Finally, the language permits to handle the severity of validation result:

- **Constraints**: they are used to capture critical errors that invalidate the model;
- **Critiques**: they are used to capture non-critical situations that do not invalidate the model, but should nevertheless be addressed by the user to enhance the quality of the model.

To implement the *Mapping Generation* phase shown in Fig. 1, a set of EVL rules have been defined. They take two models as input: the first one is a *TyphonML*$_{CE}$ specification (i.e., conceptual entities enriched with functional and non-functional tags); the second one is a *requirement definition model* containing all the databases that can be managed by TyphonDL with also all the functional and non-functional requirements that each of them is able to meet (see Fig. 4).

We exploit the potential of EVL to search all (and only) metaclasses of type `Entity` that have functional and/or non-functional requirements specificed (see line 1-2 of Listing 4). For each retrieved entity the `generateDatabaseEntity` critique is executed aiming at generating a corresponding data mapping. To this end, the `feasibilityCheck` operation is executed (see line 11) to identify in the input requirement definition model the database types that can be used.

```
1 context TyphonML!Entity {
2         guard: self.hasTags() //It gets only Entities with Requirements
```

[9] https://www.eclipse.org/epsilon/doc/evl/

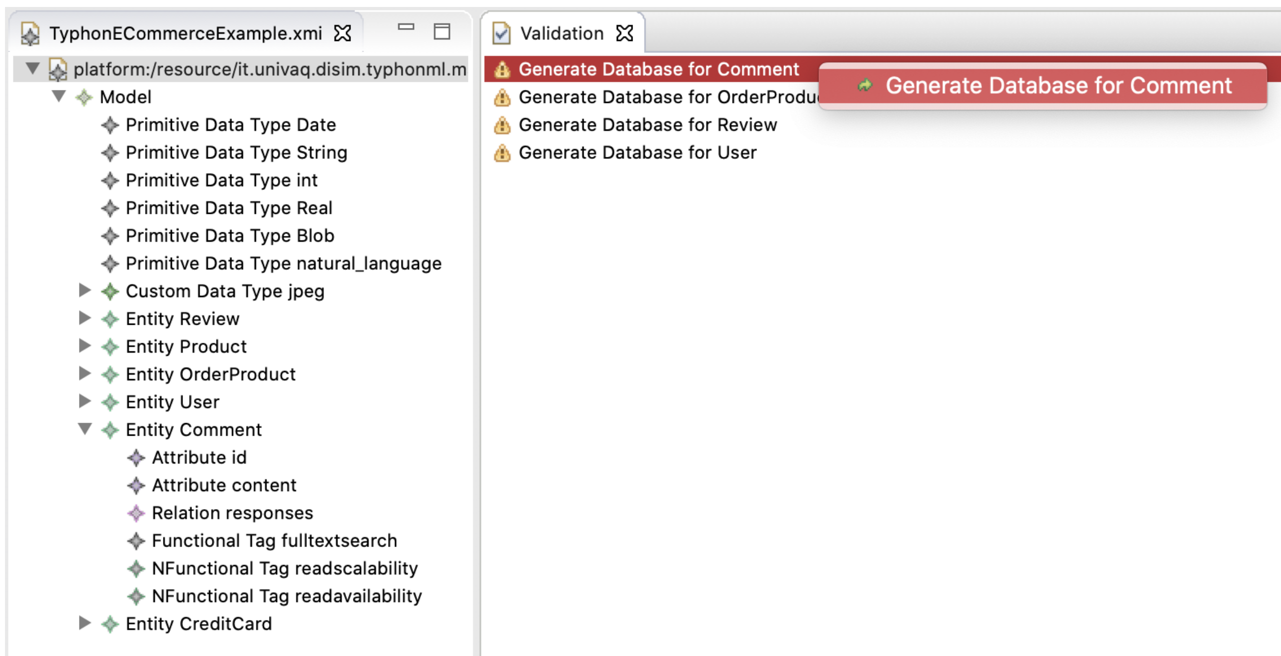


Figure 5: EVL selection entity generator based on entity tags

```
3          critique generateDatabaseForEntity{
4          check{
5                  return self.feasabilityCheck().isEmpty();
6          }
7          message: "Generate Database for "+self.name
8          fix {
9                  title: "Generate Database for "+self.name
10                 do {
11                         var selectedDatabase = System.user.choose("Select Best Match Database Type:
                               ↪ ", self.feasabilityCheck(), new List<String>);
12                         self.generateDatabaseForEntity(selectedDatabase);
13                         }
14                 }
15         }
16 }
```

Listing 4: Fragment of the EOL-based TyphonML data mapping generator

As implemented in lines `34-39` of Listing 3, the `feasibilityCheck` operation returns as output the list of databases that permits to satisfy all and only the requirements specified by the user. The returned list is ranked with respect to the total number of requirements, which are satisfied by each single solution. At this point, the user can select the best solution by means of a window dialog as shown in Figure 6.

Once the choice is made, another EOL function (line 12 in Listing 4) will take care of generating the corresponding database mapping in the TyphonM$_{DBMap}$ model being produced and making the necessary associations with the reference entity.

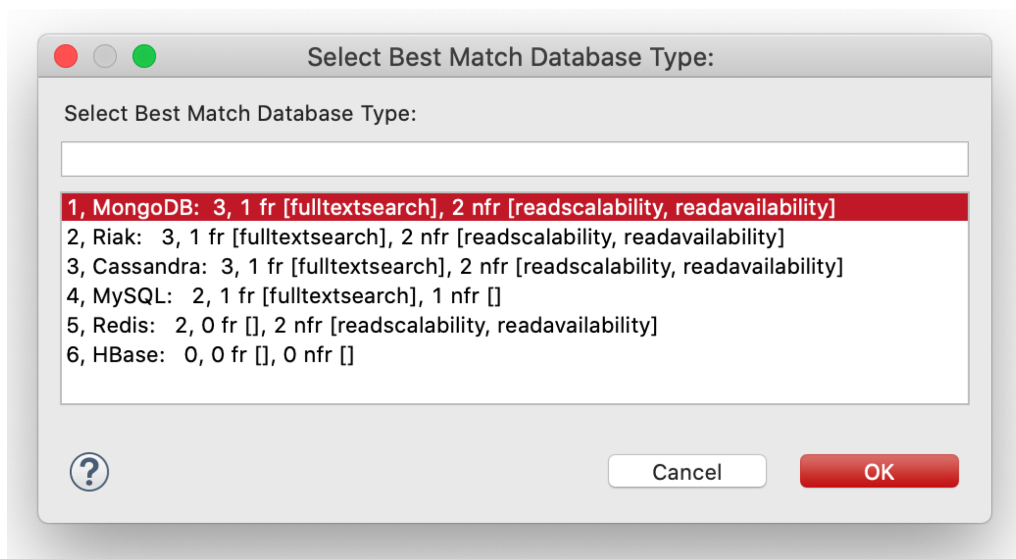


Figure 6: Database selection for the entity *Comment* with ranked solution based on satisfied functional and non-functional requirements.

It can happen that there are entities that are not tagged or that there is no database that meets the requirements defined by the user. In such cases, the user is warned that the entities to be mapped are missing, and consequently, all the databases managed by TyphonDL are shown so that the user can manually specify the mapping with a specific database.

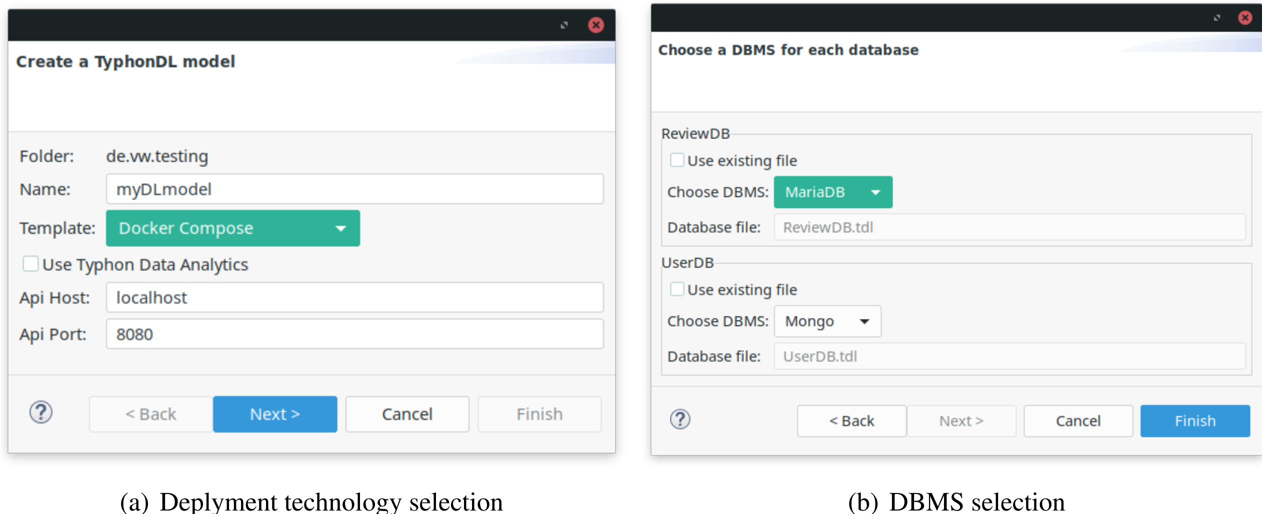
3 The TyphonML to TyphonDL transformation

As presented in the deliverable D3.2 [2], the creation of TyphonDL models is supported by means of a wizard, which takes as input a complete TyphonML model. On the first page of the wizard (see figure 7(a)) the name for the TyphonDL model has to be entered and a deployment technology such as Docker[10] or Kubernetes[11] has to be chosen from a drop-down menu. The selected technology will be included in the model in the form of `containertype` which is used when defining a container:

```
1       containertype Docker
2       container ContainerName : Docker ...
```

Listing 5: Defining a containertype

The input TyphonML model is parsed by the TyphonDL Wizard to identify the databases that needs to be deployed by TyphonDL. For each database the second page of the wizard (see Fig. 7(b)) provides the possibility to either use a pre-existing database configuration file or create a new database configuration from a template.



(a) Deplyment technology selection (b) DBMS selection

Figure 7: TyphonDL Creation Wizard

To create a new database configuration the user is asked to choose one of the supported DBMS for the specific database types, e.g. MariaDB[12] or MySQL for relational databases or MongoDB for document databases. In Listing 6, a basic MongoDB configuration is generated from the database name taken from the TyphonML model and a database configuration template included in the TyphonDL plugin (see also [2]). Per default the latest image from Docker Hub[13] with only the necessary variables is used.

```
1 dbtype mongo {
2       default image = mongo:latest ;
3 }
4 database ReviewDB : mongo {
```

[10] https://www.docker.com/
[11] https://kubernetes.io/
[12] https://mariadb.org/
[13] https://hub.docker.com/

```
5         environment {
6                 MONGO_INITDB_ROOT_USERNAME = admin ;
7                 MONGO_INITDB_ROOT_PASSWORD = admin ;
8         }
9 }
10            ...
11                    container reviews : Docker {
12                            deploys ReviewDB
13                            ports {
14                                    target = 27017 ;
15                                    published = 27018 ;
16                            }
17                    }
18            ...
```

Listing 6: Template for MongoDB

The creation of a TyphonDL model has been improved since the deliverable D3.2 [2] to consider also the functional and non-functional requirements that are given as described in the previous sections. In particular, two different improvements have been operated:

1. The list of possible target databases shown in the wizard is narrowed down by considering the output of the *feasibility check* tool presented in the previous section. In particular, with the automated generation of TyphonML data mappings presented in the previous section, a list of possible DBMSs for each database is also generated and can be parsed by the TyphonDL creation wizard. Thus, the user can only choose from the recommended DBMSs fitting the wanted requirements.

2. Deployment configurations are generated so to satisfy the selected requirements. To help the user implement a deployment that satisfies all the needs previously defined in the TyphonML model, some TyphonDL fragments can be generated automatically by exploiting available deployment patterns (see the right hand side of Fig. 1). For instance, if the user wants to use a MongoDB and needs *durability*, the journal[14] can be activated in the deployment model by adding a command that is passed to the image when the container is started. Since the field `command`[15] in Docker Compose accepts a string array or a string, in Listing 7 the command `--journal` is added as a `Key_Value` pair. In the Kubernetes specification `command`[16] needs a string array, so Listing 8 is adjusted accordingly. Both additions to the model result in enhanced deployment scripts.

```
1                    container reviews : Docker {
2                            deploys ReviewDB
3                            ports {
4                                    target = 27017 ;
5                                    published = 27018 ;
6                            }
7                            command = --journal ;
8                    }
```

Listing 7: Template for MongoDB with requirement *durability* using Docker Compose

[14] https://docs.mongodb.com/manual/reference/glossary/#term-journal
[15] https://docs.docker.com/compose/compose-file/#command
[16] https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.16/#container-v1-core

```
1                container reviews : Docker {
2                    deploys ReviewDB
3                    ports {
4                        target = 27017 ;
5                        published = 31015 ;
6                    }
7                    command [--journal]
8                }
```

Listing 8: Template for MongoDB with requirement *durability* using Kubernetes with Docker containers

The mapping of functional and nonfunctional requirements to the actual deployment specifications is defined in separate configuration files for each DBMS and will be delivered with the forthcoming version of the TyphonDL plugin.

4 Conclusions

In this document we presented enhancements that have been operated to the TyphonML and TyphonDL tools to enable the automated generation of deployment configurations satisfying requirements defined by the modeler. In particular, the TyphonML language has been extended in order to enable the specification of functional and non-functional requirements for each conceptual data entity specified in the TyphonML model being defined.

Feasibility checks have been also developed to check if the given requirements can be met by the available DBMSs. TyphonML data mappings are now generated in a requirement-consistent way. Also the TyphonDL tools have been enhanced so that deployment configurations can be generated by exploiting the availability of reusable requirement implementation patterns.

The conceived techniques and tools will be improved in the forthcoming months in order to address unforeseen requirements that can arise by the use case providers and by the other WPs.

References

[1] Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J. Elmore. Database scalability, elasticity, and autonomy in the cloud. In Jeffrey Xu Yu, Myoung Ho Kim, and Rainer Unland, editors, *Database Systems for Advanced Applications*, pages 2–15, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] Institut für angewandte Systemtechnik Bremen. D3.2 – TyphonDL Tools, 2019.

[3] Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. NoSQL database systems: a survey and decision guidance. *Computer Science - Research and Development*, 32(3-4):353–365, July 2017.

[4] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *Proceedings of the 2005 International Conference on Satellite Events at the MoDELS*, MoDELS'05, pages 128–138, Berlin, Heidelberg, 2006. Springer-Verlag.

[5] Dimitrios Kolovos, Louis Rose, Richard Paige, and Antonio Garcıa-Domınguez. The epsilon book. *Structure*, 178:1–10, 2010.

[6] DimitriosS. Kolovos, RichardF. Paige, and FionaA.C. Polack. The epsilon transformation language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 2008.