2. EXISTING BENCHMARKS

In this section we briefly discuss some benchmarks that are close to the benchmark that we have developed.

2.1 TPC-C

This benchmark is described in [8]. Some of the important characteristics of the environments that it aims to simulate are:

(1) Various transactions are executed concurrently.

(2) Transactions must adhere to the ACID properties.

(3) The system consists of many tables that differ greatly in size, attributes and relationships.

To this end, the specification describes a fictional data model of a wholesale supplier that has tables like warehouse, district and customer. The model consists of 9 tables. It describes in detail which attributes each table should have and what initial master data should be present in the system.

On this data model, a number of transaction profiles are described: the new-order transaction, the payment transaction, the order status transaction, the delivery transaction and the stock-level transaction. A complete business cycle uses a combination of these transactions to simulate the flow of a business process.

TPC-C produces four primary metrics:

(1) **tpmC:** the business throughput per minute, measured as the number of processed orders.

(2) **price per tpmC:** the total cost of running the system for 3 years, divided by the tpmC.

(3) **Availability date**: the earliest date at which all components of the system will be available.

(4) **watts per KtpmC:** the cost in energy per 1,000 tpmC (optional).

2.2 TPC-E

TPC-E is described in [9]. Both in goal and specification it is very similar to TPC-C. The main difference is that it is much more complex than TPC-C and claims to be more realistic and have a broader coverage. Its data model is that of a fictional brokerage firm. According to Chen et al. [3] the main differences are that TPC-E specifies more tables, more attributes, has check constraints and also tests for referential integrity.

2.3 Multi-Tenant TPC-W

TPC-W is a benchmark for e-Commerce web applications. It emulates a fictional online bookstore. It features 14 web pages that allow users to browse, search, order and pay for products. The test is run by using emulated browsers that simulate real users by employing random wait times between 7 and 70 seconds.

TPC-W produces two primary metrics:

(1) **WIPS**: the number of web interactions per second that can be sustained by the system.

(2) **Cost per WIPS**: the cost of the system divided by the WIPS rate.

Kreb et al. [5] enhanced TPC-W to make it into a Multi-Tenant benchmark. To this end, they added a column "tenantId" to every table and added a central administration mechanism for assigning primary keys. Unfortunately, their extension does not provide clear comparable performance metrics. The results are graphs that show how increasing tenants impacts the original TPC-W metrics. Another problem with this benchmark is that it does not account for customizability.

3. MTC-DB BENCHMARK (MTCB)

In this section we describe the MTC-DB Benchmark (MTCB) that we developed. First we discuss the requirements we formulated, then the specification that we designed and finally the concrete steps a developer should take to start using the benchmark to implement and benchmark their own MTC-DB implementation.

3.1 Requirements

The design of MTCB has the following requirements:

R1 Realistic. It should be a realistic reflection of an MTC environment in terms of performance aspects.

R2 Unambiguous. It should be as unambiguous as possible. An MTC-DB developer should be able to implement it without support from an expert such as one of the authors of this paper.

R3 Comparable. It needs to provide objective and easily comparable performance metrics.

R4 Correct. It needs to test for correctness. This mainly concerns the ACID properties of database transactions: they should be atomic, consistent, isolated and durable.

R5 Scalable. It needs to be easy to benchmark very small scenarios, very large scenarios and a number of inbetween scenarios.

R6 Simple. It should be as simple as possible. Every added complexity should aid one of the other requirements in some way.

R7 Independent. The specification should stand on its own. It needs to be independent from any MTC-DB implementations. For example, there should be no dependency on the paradigm of SQL and/or RDBMS.

3.2 Conceptual Model

MTCB not only defines the metrics to measure the performance of MTC-DB implementations, it also defines what an MTC-DB implementation is. Our definition of MTC-DB is pure: there must be as little core functionality as possible. For example, the Force.com platform [7] is an MTC application that has a large amount of core functionality. For this core functionality they use traditional tables and only for the modifications by third party platform developers generic extension tables are used. In this model, the platform developers are second class to the native Force.com developers.

We defined the absolute minimum of core functionality to be four complex types[2] and four primitive types[3]. The complex types are tenant, user, type and attribute. The primitive types are string, number, timestamp and boolean. Some of these concepts are explained in detail below.

3.2.1 Tenant and User

Bezemer et al. [2] give the following definition of tenant:

[2] Complex types are types that have attributes

[3] Primitive types are types that have no attributes

A tenant is the organizational entity which rents a multi-tenant SaaS solution. Typically, a tenant groups a number of users, which are the stakeholders in the organization.

This is a good basic definition, but in our model, we use an enhanced version of this definition. A tenant as described in the definition above is what we call a data tenant. This kind of tenant mostly contains transactional data and master data, and little to no metadata. Its data is also isolated: no other tenants can access it.

A second type of tenant is a module tenant. This kind of tenant contains little transactional data and no master data, but mostly consists of metadata instead. Furthermore, its data is not isolated: it is accessible by all tenants that have declared a dependency on it. It can also be dependent on other module tenants itself. This extension of the definition is necessary, because metadata needs to be shareable. If metadata could not be shared, then each data tenant would have to build its own application from scratch.

3.2.2 Type and Attribute.

Types are the metadata building blocks of the system. A type has a name and a display name, is defined within a tenant and has some attributes. Types can be also be enhanced with additional attributes by tenants other than the tenant that owns them.

Attributes refer to two types: their master type and their data type. The master type is the type that they are an attribute of, and the data type is the type of the data that they store. This can be one of the primitive types, but it can also be another complex type. An attribute must also indicate if it is searchable. This determines whether the attribute can be used in the predicate of a search query, to allow the MTC-DB implementation to optimize for this.

3.2.3 Search Design.

One of the things that should be benchmarked is how fast search queries run in the system. We distinguish two types of queries: load by ID and attribute search.

Load by ID. Load by ID queries simulate the major workload of OLTP systems. In typical OLTP systems that use an RDBMS for storage, a window in the user interface displays one row of a particular database table. To load this data, the application must retrieve this row. If this row contains foreign key references to other tables, then these must also be resolved.

In regular SQL, this can be done with a query of the following form:

```
SELECT t.ID, t.Name, t1.Name, t2.Name
FROM t,
JOIN t1 ON t1.ID = t.t1_ID
JOIN t2 ON t2.ID = t.t2_ID
WHERE t.ID = 1000;
```

To benchmark these type of queries, we specify Transaction Data Types (TDT) and Master Data Types (MDT). Both are synthetic data types that are also used for benchmarking customizability and the creation of new type instances. An MDT is a type that contains only simple attributes. A TDT also contains complex types: references to MDTs.

Attribute search. To benchmark queries with search predicates, we need to design search data and search queries. The search queries need to be representative for worst-case scenarios in the system and the search data needs to be generated automatically and needs to be scalable.

To this end we defined a designated search type and a designated search tenant, which are created in the Setup script (see section 3.3.2). This script also creates a number of instances of the search type in the search tenant, dependent on the Profile (see section 3.3.2).

We defined two distinct attribute searches: a conjunctive search and a disjunctive search. In regular single-tenant SQL, these searches have the following form.

Conjunctive search:

```
SELECT * FROM t
WHERE a1=? AND a2=? AND a3=? AND a4=? AND a5=?
```

Disjunctive search:

```
SELECT * FROM t
WHERE a6=? OR a7=? OR a8=? OR a9=? OR a10=?
```

We chose these two queries because they are two extremes and if an MTC-DB can implement these queries, they can implement all single type queries, because any propositional formula can be translated into conjunctive normal form (CNF) and the disjunctive normal form (DNF).

The search type has 10 attributes, 5 that are used by the conjunctive search, and 5 that are used by the disjunctive search. When instantiating these types, the attributes for the conjunctive search are populated with random integers in the range $[1, \sqrt[5]{n}]$, and the attributes for the disjunctive search are populated with random integers in the range $[1, 5n]$ where n is the total number of search type instances. Similarly, when creating the search queries, the search terms are picked from the same range at random.

These ranges were picked specifically to make sure that no matter how large n gets, the searches will have approximately the same probability of returning no results, namely $\frac{1}{e}$, about 37%. This is true, because the probability for returning no results for the respective queries can be expressed with the following formulas, which both approximate $\frac{1}{e}$ for $\lim n \to \infty$.

Conjunctive Search:

$$P(n) = (1 - (\frac{1}{\sqrt[5]{n}})^5)^n$$

Disjunctive Search:

$$P(n) = (\frac{5n-1}{5n})^{5n}$$

3.3 Specification

MTCB consists of two main parts: the model and the scripts. The model consists of the contract of six interfaces. Every MTC-DB implementation must provide implementations for these interfaces. The scripts use these interfaces to run the benchmark and report the performance metrics.

3.3.1 The model

The model consists of six interfaces: `MTCDB`, `PO`, `Type`, `Attribute`, `Tenant` and `User`. For `MTCDB` the most important elements of its contract are shown in Table 1. For the

complete specification of all interfaces we refer to the example implementation[4].

MTCDB is the main entry point. An instance of this class must be fed to the scripts. The most important operations that are defined on MTCDB are create `Tenant()`, `createType()`, `createAttribute()` and `createPO()`.

`PO` stands for Persistent Object. This is the base contract for all entities that must be stored persistently. Its most important operations are `persist()` and several `GetValueAs...()` operations to retrieve a value for an attribute by attribute name.

`Type`, `Attribute`, `Tenant` and `User` are interfaces that extend `PO`. So they contain all the operations that `PO` contains, including some extra operations. For example, `Type` has the operation `getAttributes()` to get all attributes for that type, and `Attribute` has the operations `getMasterType()` to get the type it is an attribute of, and `getDataType()` to get the type of the value that it can store.

Table 1: The contract for the interface MTCDB

MTCDB Interface	
addModule()	Add a metadata module to the tenant in the context.
createAttribute()	Add a new attribute to an existing type.
createPO()	Create a new instance of an existing type.
createTenant()	Create a new empty tenant.
createType()	Create a new type without attributes.
createUser()	Create a new user.
getByConjunction()	Retrieve PO instances by conjunction (AND) of attributes.
getByDisjunction()	Retrieve PO instances by disjunction (OR) of attributes.
getPOByID()	Retrieve a PO instance by ID.
getSizeOnDisk()	Retrieves the total size on disk in MB of the MTC-DB.
getTenant()	Get tenant by name.
getTenants()	Get all tenants that exist in the system.
getType()	Retrieve type by tenant and name.
getTypes()	Get all types for a tenant.
initializeSystem()	Set up the most elemental metadata necessary for the system.

3.3.2 The scripts

There are three scripts: the Aulbach script, the setup script and the main script. These scripts are explained below. As input, each of these scripts needs an implementation instance of `MTCDB`. The latter two also need a profile that contains a set of parameters. How `MTCDB` is implemented and instantiated is up to the specific MTC-DB implementation.

Profile. The profile needs to be one of the options shown in Table 2. The profile *Tiny* is mostly meant for development purposes, to have a benchmark profile available that runs with minimal resources and allows for a quick test. The profile *Small* is for scenarios in which the system is expected to only accommodate a small number of tenants. The profile *Medium* should be a realistic scenario for many applications that are in need of an MTC-DB layer. We have purposely omitted terms such as *Large* and *Very Large* to allow these terms to be added in the future, because it is to be expected that computer systems will keep scaling up as the available computing power and storage space will keep growing exponentially.

The parameters mentioned in Table 2 have the following meanings:

(1) **DT**: Data Tenants. The number of data tenants that are created in the setup script.

(2) **CF**: Concurrency Factor. The number of threads each benchmark operation of the main script will use. Since there are 7 operations, the total number of concurrent threads the main script uses is 7 times *CF*.

(3) **MDT**: Master Data Types. The number of master data types that are defined in the metadata module. We define master data as data that does not refer to other data, but is referred to by transaction data.

(4) **TDT**: Transaction Data Types. The number of transaction data types that are defined in the metadata module. We define transaction data as data that refers to *MINRA* to *MAXRA* (see below) master data records.

(5) **MDI**: Master Data Type Instances. The number of master data type instances per tenant per type that will be created in the setup step of the test script.

(6) **STI**: Search Type Instances. The number of search type instances. Used for benchmarking the search (see section 3.2.3).

(7) **TI**: Test Interval. The duration of the test in seconds.

(8) **MINRA**. Minimum Reference Attributes. The minimum number of reference attributes on transaction data types.

(9) **MAXRA**. Maximum Reference Attributes. The maximum number of reference attributes on transaction data types.

Aulbach script. The Aulbach script checks if the implementation is a correct MTC-DB implementation by testing if it is capable of representing the example MTC data structure used in the paper by Aulbach et al.[1]. This example consists of an Account table that is used by three tenants. One tenant uses the table with an extension for the health care industry, one with an extension for the automotive industry and one uses it without an extension.

Setup script. The setup script sets up the MTC-DB implementation for running the main script. To this end, it creates several synthetic tenants, users, types and attributes. The amount of data it generates is heavily dependent on the chosen profile (see Table 2).

Main script. The main script consists of seven operations that concurrently run for *TI* seconds. Each operation runs in *CF* concurrent threads, so the total number of threads is 7 times *CF*. The operations are:

(1) **Create Tenants.** Every 5 seconds, create a new tenant with a random name and a dependency on the module "Main-

[4] https://bitbucket.org/actfact/mtcdb-benchmark

Module". Report the total number of tenants created and if it managed to achieve maximum performance.

(2) **Create Types**. Every 500 milliseconds, create a new type with a random name for a random tenant. Report the total number of types created and if it managed to achieve maximum performance.

(3) **Create Attributes**. Every 100 milliseconds, create a new searchable string attribute with a random name for a random transaction data type for a random tenant. Reports the number of attributes created and whether the maximum performance was achieved.

(4) **Create Transaction Data Type Instances.** Constantly create transaction data type instances. Because these have references to *MINRA* to *MAXRA* master data types, this workload also includes retrieving master data types by name. Report the total number of TDIs created.

(5) **Load by ID.** Constantly load previously created transaction data type instances by ID. This workload also includes loading all the references of the TDT to MDTs by ID. Reports the number of TDTs loaded.

(6) **Conjunctive Search.** Constantly perform 5-way conjunctive (AND) searches. Only the first result is retrieved. It is designed in such a way that each search has about a 63% chance of returning a result. Reports the number of conjunctive searches performed per minute. See section 3.2.3 for more information.

(7) **Disjunctive Search.** Constantly perform 5-way disjunctive (OR) searches. Only the first result is retrieved. It is designed in such a way that each search has about a 63% chance of returning a result. Reports the number of disjunctive searches performed per minute. See section 3.2.3 for more information.

Table 2: Parameter Profiles

		TINY	SMALL	MEDIUM
DT	Data tenants	10	100	1000
CF	Concurrency Factor	1	5	10
MDT	Master Data Types/Tenant	20	100	100
TDT	Transaction Data Types/Tenant	80	400	400
MDT	Master Data Types Instances/MDT/Tenant	2	2	2
STI	Search Type Instances	10,000	100,000	1,000,000
TI	Time Interval	60	300	300
MINRA	Minimum Reference Attributes on TDT	2	2	2
MAXRA	Maximum Reference Attributes on TDT	15	15	15

3.3.3 Metrics.

The benchmark produces the following metrics:

(1) **Aulbach compliance:** a boolean indicating whether the implementation is able to represent the example MTC scenario described by Aulbach et al. [1]. Every implementation needs to score true on this metric.

(2) **Size on disk**: the total size on disk in MB of the MTC-DB after running the setup script. There is no maximum indication. The lower the better.

(3) **Tenants created:** the percentage of tenants created in relation to the maximum possible amount. This should be 100%.

(4) **Types created:** the percentage of types created in relation to the maximum possible amount. This should be 100%.

(5) **Attributes created:** the percentage of attributes created in relation to the maximum possible amount. This should be 100%.

(6) **TDI created per minute:** the number of transaction data type instances created per minute while running the main script. There is no minimum indication. The higher the better.

(7) **TDI loaded by ID per minute:** the number of transaction data type instances loaded by ID per minute. There is no minimum indication. The higher the better.

(8) **Conjunctive searches per minute**: the number of conjunctive searches performed per minute. There is no minimum indication. The higher the better.

(9) **Disjunctive searches per minute**: the number of conjunctive searches performed per minute. There is no minimum indication. The higher the better.

3.4 Developer Guide

To help developers utilizing this bench mark with minimal effort, we have implemented an example implement ationin Java8. This code is available under the Mozilla Public Licenceat: https://bitbucket.org/actfact/mtcdb-benchmark

Developers can clone this Git repository and follow the instructions in the *readme* file. To implement their own MTC-DB implementation, they will need to write implementations for all the Java interfaces in the MTCB codebase. They are encouraged to refer to the example MTC-DB implementation or even use it as a starting point if they are unsure how to proceed. Of course it is also possible to write a non-Java implementation, but in this case the developer will first have to implement the API himself.

4. EVALUATION

The evaluation consists of two parts. First we perform a conceptual evaluation, in which we evaluate if this benchmark fulfills the requirements we formulated in Section 3.1. Second, we perform a practical evaluation. In this part, we discuss an MTC-DB example implementation we developed and how we used it to evaluate the usability of the benchmark.

4.1 Conceptual Evaluation

4.1.1 Realistic

We defined a main module that contains the metadata for types that all data tenants use. In a real world situation it will also be the case that a large majority of metadata is the same for each tenant.

In the main script, concurrent users create new data, while other threads concurrently perform metadata operations. Metadata changes are a small part of the total work load of such applications, but it is important that regular data creation is not blocked while these operations are being performed. It was shown by Wevers that this is a significant problem for many Relational Databases [10].

4.1.2 Unambiguous

We provide an implementation neutral specification and accompany this with an example implementation in Java 8. So wherever the specification leaves room for multiple interpretations, the example implementation can be referred to.

4.1.3 Comparable

The benchmark specifies a small set of parameter profiles and produces a small number of simple quantitative metrics. This enables easy and objective comparison of different implementations that use the same profile.

4.1.4 Correct

The *Aulbach script* is a minimal test that checks if the implementation is a real MTC-DB implementation. Currently no automated check is implemented for ACID compliance. On a more general note, it is not possible to automatically guarantee complete correctness. We can only check if the implementation is consistent in itself. To guarantee correctness an audit by a human expert will always remain necessary.

Table 3: The benchmark result over 10 runs for a schema based implementation, showing the average (μ) and the coefficient of variation (σ/μ).

	Tiny		Small	
Aulbach compliance	True		True	
Size on disk	138 MB		5,471 MB	
	μ	$(\frac{\sigma}{\mu})$	μ	$(\frac{\sigma}{\mu})$
Tenants created	40%	(0.09)	5%	(0.00)
Types created	100%	(0.00)	100%	(0.00)
Attributes created	23%	(0.51)	67%	(0.29)
TDI created per minute	1,504	(0.47)	2,442	(0.17)
TDI loaded by ID per minute	144,585	(0.29)	168,722	(0.21)
Conjunctive searches per minute	85,461	(0.02)	9,074	(0.06)
Disjunctive searches per minute	665,173	(0.04)	580,297	(0.06)

4.1.5 Scalable

The parameter profiles allow for benchmarking a number of scenarios of different sizes.

4.1.6 Simple

Instead of specifying a real world data model for the main module, we chose to use synthetic types and attributes. The same goes for the scripts that generate data and metadata: it is randomized and without meaning. Using a real world scenario would make MTCB extremely complex and would decrease its scalability and flexibility.

4.1.7 Independent

We specify an API that contains the operations that should be supported by the MTC-DB implementation. This API places no restrictions on the MTC-DB implementation in terms of underlying platform. For example, even though many implementations will use an RDBMS as underlying platform, this is not implied in the API. It should be equally possible to implement the MTC-DB in a document-oriented database, a functional database or any other kind of persistent storage structure.

4.2 Practical Evaluation

For the practical evaluation, we have developed a naive MTC-DB implementation. This implementation has been developed in Java 8 and PostgreSQL 9.6 and is schema based: every tenant is defined in a separate schema. It is loosely based on what Aulbach et al. call the Private Table Layout [1].It is available on the same repository as the benchmark itself, in the project mtcb-schemabased: https://bitbucket.org/actfact/mtcdb-benchmark.

We ran MTCB for this implementation on a Centos 7 server with an Intel Xeon E3-2200 Quad Core CPU and 32GB RAM. For each profile we ran the main script 10 times and report the average (μ) as well as the coefficient of variation (σ/μ) in Table 3. The reason to run it 10 times was that we noticed considerable differences between separate runs. This can be seen from the high variation for some metrics. We did not benchmark this implementation for the *Medium* profile, because it does not seem to be feasible for such a large scenario. We estimate that running the setup script would not even finish in 48 hours.

This implementation scores very well on some metrics. Most notably the disjunctive search: more than half a million per minute for both profiles. Loading TDIs is also fast.

On some of the other metrics the implementation scores very poorly. The largest problem is tenant creation. The implementation fails to comply with the requirement to create a new tenant every 5 seconds. For the small scenario, it only creates 5% of the maximum. This means it takes about 100 seconds to create a tenant. The reason for this is that in this implementation, for each tenant creation the database must run DDL[5] to create all the tables that are defined in the metadata module. Aside from taking a lot of time, this also causes the implementation to score poorly on the metric *Size on disk*. On top of this, the DDL statements have a disruptive nature, irregularly causing operations such as TDI Creation to be stalled for considerable times. This causes a high variation for those operations.

Another interesting note is that the performance does not degrade much when going from the *Tiny* to the *Small* profile. For some metrics, the performance even increases. The most likely reason for this is that the *Medium* profile has a higher degree of concurrency, running in 35 threads, whereas the *Tiny* profile only runs in 7 threads. This allows the *Medium* profile to maximize its use of the hardware resources. However, the conjunctive search still shows a severe degradation. This is probably due to the stark increase in search data: 10 times as much as in the *Small* profile.

5. CONCLUSION

We present a benchmark specification *MTCB* and a naive example implementation that proves that it is implementable. Test results show that this naive schema per tenant RDBMS implementation is not sufficient, because it cannot handle metadata modifications efficiently and causes a huge overhead in redundant metadata storage. Future work should use this example implementation as a baseline system. An interesting next step would be to create implementations based on the schema-mapping techniques discussed by Aulbach et al. [1].

We believe that this benchmark is an important contribution to the community of MTC-DB developers. Not only does it allow for objective comparison, it also makes an attempt at a very precise definition of the concept of MTC-DB, backed by a concrete implementation.

6. REFERENCES

[1] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. 2008. Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques. In *SIGMOD' 08.*

[5] Data Definition Language. SQL statements that alter the data dictionary: mostly `CREATE TABLE` and `ALTER TABLE` statements

Proceedings of the 2008 ACM SIGMOD international conference on Management of data. ACM, 1195 – 1206.

[2] Cor-Paul Bezemer and Andy Zaidman, 2010. Challenges of Reengineering into Multi-Tenant SaaS Applications. *Delft University of Technology Software Engineering Research Group*. Technical Report Series (2010).

[3] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica. 2010. TPC-E vs. TPC-C: Characterizing the new TPC-E benchmark via an I/O comparison study. *SIGMOD Record 39*, 3 (2010), 5 – 10. www.scopus.com.

[4] W. R. Friedrich and J. A. Van Der Poll. 2007. Towards a methodology to elicit tacit domain knowledge from users. *Interdisciplinary Journal of Information, Knowledge, and Management* 2 (2007), 179 – 193. www.scopus.com.

[5] R. Krebs, A. Wert, and S. Kounev. 2013. *Multi-tenancy performance benchmark for web application platforms*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 7977 LNCS. 424-438 pages. www.scopus.com.

[6] R. M. Locke. 2002. The Promise and Perils of Globalization: The Case of Nike. *MIT Working Paper* (2002). Downloaded 16 December 2016 from https://ipc.mit.edu/sites/default/files/documents/02-007.pdf.

[7] Salesforce.com. 2008. The Force.com Multitenant Architecture: Understanding the Design of Salesforce.com's Internet Application Development Platform. (2008). Accessed 4 January 2017 on http://www.developerforce.com/media/ForcedotcomBookLibrary/Force.com_Multitenancy_WP_101508.pdf.

[8] Transaction Processing Performance Council. 2010. TPC BENCHMARK C Standard Specification Revision 5.11. (2010). http://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf

[9] Transaction Processing Performance Council. 2015. TPC BENCHMARK E Standard Specification Version 1.14.0. (2015).

[10] L. Wevers. 2012. *A Persistent Functional Language for Concurrent Transaction Processing*. Master's thesis. University of Twente.

[11] Roel J. Wieringa. 2014. *Design science methodology for information systems and software engineering*. Springer.