MASTER THESIS

Radek Mácha

Data Processing in a Generic Benchmarking Environment

Department of Distributed and Dependable Systems

Supervisor of the master thesis:  RNDr. Andrej Podzimek
Study programme:  Software systems
Specialization:  Software engineering

Prague 2015
First of all, I would like to thank my colleagues from the **EverBEEN** team for over a year of effective collaboration in an unforgettable atmosphere.

This great cooperation wouldn’t have been possible without the supervisor of this thesis and the entire **EverBEEN** project – RNDr. Andrej Podzimek – to whom I would like to express my gratitude for his patient and helpful approach.

Many thanks also belong to Doc. Ing. Petr Tůma for his helpful insights both in and outside of the scope of the **EverBEEN** realization effort.

Last but not least, I would like to express my sincerest thanks to Sarah, Therese and Shaniele for their unyielding moral support.
I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ........ date ............
Název práce: Zpracování dat v generickém prostředí pro benchmarkování

Autor: Radek Mácha

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: RNDr. Andrej Podzimek, Katedra distribuovaných a spolehlivých systémů

Abstrakt:

V září roku 2013, byl na Matematicko-fyzikální fakultě Univerzity Karlovy v Praze obhájen softwarový projekt EverBEEN implementující třetí inkarnaci generického prostředí pro benchmarkování síťových aplikací.

I přes značná zlepšení v oblasti spolehlivosti a snadnosti použití, která tato verze systému přinesla oproti jeho předchozí etapě, měl EverBEEN svoje nedostatky z hlediska komerční nasaditelnosti. Jedným z těch hlavních byla absence standardizovaného způsobu extrakce a zpracování dat.

Tyto nedostatky vedly ke stěžejní otázce této práce: Jak se dají v generickém prostředí, jako je EverBEEN, zpracovávat data bez předchozí znalosti jejich struktury?

Ačkoliv se tato práce soustředí na vytvoření reusabilních nástrojů pro extrakci a shlukování dat ve frameworku EverBEEN, zabývá se mimo jiné i analýzou prostředku pro automatizaci toku řízení v tomto frameworku a jeho zapojení do procesu kontinuální integrace.

Klíčová slova: vyhodnocování výkonnosti, zpracování dat, prostředí pro benchmarkování
Title: Data Processing in a Generic Benchmarking Environment

Author: Radek Mácha

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Andrej Podzimek, Department of Distributed and Dependable Systems

Abstract:

In September 2013, at the Charles University in Prague, Faculty of Mathematics and Physics, a software project implementing the third incarnation of a generic benchmarking environment aimed at performance evaluation of networked applications was presented: **EverBEEN**.

Despite significant advancements achieved by this incarnation in both reliability and ease of use, **EverBEEN** still came out somewhat wanting in terms of commercial usability. One of its major shortcomings was the absence of a standardized way of data extraction and processing.

The want of such means in **EverBEEN** laid foundation to the central question of this thesis: How to extract and process data from a framework like **EverBEEN**, with no prior knowledge of the structure of said data?

Albeit centered on the creation of a common, reusable data extraction and aggregation codebase for said framework, this thesis also strives to analyze means of automating **EverBEEN** control-flow and incorporating the framework, and its data processing, into continuous integration.

Keywords: performance evaluation, data processing, benchmarking environment
8 Extracurricular features
8.1 Software Repository binding configuration ........................................... 30
8.2 Task Descriptor XSD cleanup ................................................................. 30
8.3 Improved logging .................................................................................. 30
8.4 Bugfixes ................................................................................................. 30

9 Notes on digital content
9.1 Source ..................................................................................................... 31
  9.1.1 EverBEEN source code additions ....................................................... 31
  9.1.2 Data processors .................................................................................. 32
9.2 Binaries .................................................................................................... 33
9.3 Documentation .......................................................................................... 33
9.4 Deployment and use guidelines ................................................................. 33
9.5 Task development guidelines ................................................................... 34

Conclusion
9.6 Retrospective ............................................................................................ 36
9.7 Roadmap ................................................................................................... 36
  9.7.1 Short term goals .................................................................................. 37
  9.7.2 Long term goals .................................................................................. 37

Bibliography .................................................................................................... 38

Appendices
A Glossary ........................................................................................................ 41
B Data processor properties ............................................................................ 42
  B.1 Selectors .................................................................................................. 42
  B.2 Processors ............................................................................................... 43
  B.3 Generators .............................................................................................. 43
C Primitive types .............................................................................................. 44
Introduction

In September 2013, along with my colleagues Martin Sixta, Tadeáš Palusga and Jakub Břečka, we presented, at the Charles University in Prague, Faculty of Mathematics and Physics, the third incarnation of a generic benchmarking environment aimed at performance evaluation of networked applications: EverBEEN. Although the former goal of this project was mainly refactoring, renovation, introduction of new technologies and general maintenance aimed at improving the former incarnation’s sub-optimal reliability, the project came out as a complete overhaul, creating a modular codebase with use of modern technologies and up-to-date development practices.

Motivation

Aside from aiming for the aforementioned criteria of robustness, redundancy and reliability, we also heeded standing experience with the framework’s practical deployment, as described in J. Täuber’s master thesis [1], and strove to improve our version’s commercial usability. Much improvement was achieved in terms of simplicity of deployment, stability, and ease of use in terms of gathering benchmarking results.

However, our incarnation of the project suffered from a want of reusable means of retrieving or analyzing collected data. The primary goal of this thesis is to remedy this situation by creating a universal and reusable data extraction/processing mechanism which will enable EverBEEN users to perform data exports and assemble simple aggregation metrics without having to develop additional code.

Focus

During the development of EverBEEN, we tried to consider as many use-cases for the framework as we could find, but the two following really stood out above the rest.

Offline back-to-back benchmarking A fairly simple scenario, where a benchmark runs on a predefined set of revisions and, potentially, implementations of evaluated software. It involves a lot of repository checkouts and/or compilation. I call it ”offline”, since it is initiated manually, by the user.
Continuous regression benchmarking  A use case on par with the current trend of continuous integration, where **EverBEEN** is used side-by-side with a Continuous Integration Tool (**CIT**), such as **Jenkins CI Server** or, recently, **IntelliJ TeamCity**. The execution of benchmarks is not initiated by the user, but driven by payload incoming from the **CIT**.

While offline back-to-back benchmarks of the **Corba** middleware were the original use-case that led to the creation of the first incarnation of the **BEEN** framework, continuous regression benchmarking is the use case I consider more enticing. To my mind, it makes better use of the unique features **EverBEEN** is capable of providing, and is more interesting for commercial use. Therefore, the features I develop in this thesis will not only attempt to implement a data processing engine, but also lean towards improving the useability of **EverBEEN** for continuous regression benchmarking.
1. Benchmarking data in EverBEEN (Motivation)

The EverBEEN framework supports custom, user-defined data types (as long as they extend the Result class, which is a part of the API for writing tasks), and thus allows the user to store and recollect task data in a seamless way. This behavior, however, is achieved in a particular way, and the core framework itself does not "understand" user types. Seen as understanding data is an important part of being able to process them, I would like to describe data handling in the original EverBEEN implementation.

Before I start, though, a word of caution for the readers unfamiliar with the EverBEEN project: I will be referring heavily to project-specific naming conventions – both in this section and in the rest of my thesis. Although I tried to alleviate the situation by explaining the crucial terms in the glossary in appendix A it might be necessary for the reader to consult the EverBEEN project documentation [3] for more insight on the topic being discussed.

1.1 User types and definitions

Each task that EverBEEN launches on one of its Host Runtime nodes is run in a separate JVM. This architectural decision was originally made to increase the framework’s robustness and separate memory spaces of user code and framework code, but it also has interesting implications on type awareness.

The most important ramification of this design is that the task runs with a completely different classpath than the framework uses. This raises opportunity for user code separation, which is once again beneficial for the stability of the framework itself, but also means that EverBEEN will not be able to understand user types.

Generally speaking, loading user bytecode into a framework’s main JVM is not advisable without the support of another framework to handle class unloading. Seen as frameworks with custom class loaders usually have a high memory footprint and sometimes generate hardly predictable load, we felt like this was not the way to go when we were implementing EverBEEN.

1.2 Transit, storage and querying

Let us examine the lifecycle of a user-defined type (Result subclass). The user’s task has presumably finished benchmarking and is now looking to store his measurements persistently. To that end, the user has created class A:
class A extends Result {
    private String myVal1;
    private Integer myVal2;
    // ... more fields ...

    public String getMyVal1() {
        return myVal1;
    }

    public void setMyVal1(String myVal1) {
        this.myVal1 = myVal1;
    }

    public Integer getMyVal2() {
        return myVal2;
    }

    public void setMyVal2(Integer myVal2) {
        this.myVal2 = myVal2;
    }

    // ... more getters / setters ...
}

1.2.1 Persistence

In his/her task code, the user fills this POJO with values he benchmarked and then presumably invokes either

@Override
public void run(String[] args)
throws TaskException, MessagingException, DAOException {
    // ... do some calculations
    final A result = results.createResult(A.class);
    // ... set values from benchmarks to result
    results.persistResult(A, "myResultGroup");
}

or
```java
@override
public void run(String[] args)
throws TaskException, MessagingException, DAOException {
    // ... do some calculations
    final A[] results = new A[resultCount]
    // ... set values from benchmarking to results
    final Persister rp =
        results.createResultPersister("myResultGroup");
    for(A result: results)
        rp.persist(result);
}
```

or in some other way makes use of the persisting functionality.

At this point, the instance of A passed to one of the persisting serialized into JSON

```json
{
    "myValue1": "Some string value.",
    "myValue2": 15
}
```

and passed as a String into an EntityCarrier envelope, which identifies the targeted persistence layer collection the result should land in.

The EntityCarrier is then pumped through a 0MQ socket to the Host Runtime overseeing the node on which the user task is being executed. The Host Runtime, or any other EverBEEN component, only unpacks the EntityCarrier, but not its data segment, containing the serialized JSON representation of the A instance.

### 1.2.2 Retrieval

Retrieving and evaluating the results in the original EverBEEN implementation requires the implementation of an additional task. When evaluating results of previous benchmarks (in an evaluator implementation), the user again invokes the ResultFacade interface, like in the following example.
@Override
protected EvaluatorResult evaluate()
throws
    TaskException,
    MessagingException,
    DAOException
{
    // get the contextId of results we want to evaluate
    final String contextId = ...;

    final Query query = new QueryBuilder()
        .on(new EntityId()
            .withKind("result")
            .withGroup("myResultGroup")
        )
        .with("contextId", getContextId())
    final Collection<A> retrievedResults =
        results.query( query, A.class);

    // ... do some calculation with retrieved results ...
    // ... produce a result file (int byte array form)

    final EvaluatorResult result = new EvaluatorResult();
    // ... fill the result ...
    return result;
}

This results in serialization of the user’s query (for detailed description of the process, see the EverBEEN Project Documentation [3]). The query is then transferred through the cluster to the Object Repository, which unpacks it. The Query objects are known to both the task JVM and the cluster, so this causes no issues. However, the Object Repository implementation must be capable of querying user data types. On top of that, the EverBEEN framework strictly separates the persistence layer interface from its default MongoDB implementation. Therefore, the Query and QueryAnswer interfaces are designed for total abstraction, and all actual querying is done through field name mappings to generic conditions modeled with Java classes in the EverBEEN framework itself. For more information on available queries, see [3].

The persistence layer once queried, the Object Repository only retrieves serialized JSON representation of fetched objects, which are funneled all the way back to the requesting task using distributed maps. Only once delivered to the task’s JVM are they deserialized, using the user’s class A for the mapping.

1.3 Implications

The above described model grants users relative comfort when grouping the data accumulated during their benchmarks. However, to successfully retrieve and process
the results produced by a previously run task, the user must first create an *evaluator*

```java
@override
protected EvaluatorResult evaluate() throws TaskException, MessagingException, DAOException {

    // get the contextId of results we want to evaluate
    final String contextId = ...

    final Query query = new QueryBuilder()
        .on(new EntityId()
            .withKind("result")
            .withGroup("myResultGroup")
        .with("contextId")
    final Collection<A> retrievedResults = results.query(query, A.class);

    // ... do some calculation with retrieved results ...  
    // ... produce a result file (int byte array form)
    final byte[] resultFile = ...

    final EvaluatorResult result = new EvaluatorResult();
    result.setData(resultFile);
    result.setMimeType(EvaluatorResult.MIME_TYPE_CSV);
    result.setFilename("MyResultFile.csv");
    result.setTimestamp(System.currentTimeMillis());

    return result;
}
```

Among other, this requires the user to:

- Create a new project.
- Write an *evaluator* implementation, similar to the above.
- Parameterize the implementation to his needs, and therefore handle task property parsing etc.
- Ensure that the *evaluator* shares the same user type that was previously used by the *task* for storing its results. This may sometimes become a hassle, if fields are added / removed from class A, as that may result in unmarshaling errors.
• Implement conversion to file (for storage in EvaluatorResult).

Clearly, a lot of work here is left to the user him/herself. Seen as he/she was already put through designing the benchmarking logic, implementing it in task code and prototyping the result POJO, it would be commendable that he receive an instrument to at least verify his data and perform some basic statistics on them.
2. Data extraction

The major drawback of EverBEEN as it was presented to the committee at the Faculty of Mathematics and Physics, Charles University in Prague was the lack of a simple way of at least extracting the results produced by previously run tasks. Letting the user choose between writing an evaluator and viewing the results in a database console is indeed not very user friendly.

Therefore, the first step I took in this thesis towards remedying the drawback explained in the previous chapter, was to implement reusable export tasks. The idea behind these is simple: Let the user specify which results are relevant to him using a set of well-defined parameters, and the export task will retrieve these results and serialize them into a file, which will be available for download from the framework.

2.1 Selector enhancements, aliasing

Seen as the user may not need his entire record, which he specified in his Result extension (class A in the previous chapter), I decided to ensure the field selection features of EverBEEN enabled the user to recursively select any field from his own object, even if it is a composite.

Since the obvious question here is "How does the user specify which fields to load?", I will ask the reader to indulge a slight anachronism from my part: The fields are selected based on type mappings provided by the user. But since the entire chapter 3 is dedicated to the problematics of type mapping, I will leave that matter for later, focusing purely on data extraction in this chapter.

Given that the variable names are bound to get quite long in recursive notation, e.g. composite1.composite2.field1, I also decided to implement an aliasing feature for data extractors. This enables the user to specify alias bindings, for example:

a -> field1 | b -> composite1.field2

All aliased fields will then be exported with the alias as their name, rather than the full dotted path.

2.2 Output formats

There are obviously multiple formats I wanted EverBEEN to be able to export to. After some consideration, these three made their case in the implementation:
As the no. 1 choice in machine-readable formats, XML was the first export format that I implemented support for. As the attribute names vary from run to run, I opted out of using XML binding frameworks and implemented XML serialization by hand. Arguably, I could have used a streaming API, such as StAX, which is a decision that may yet come to realization, if there is demand for namespace-aware or validating exports.

A less likely format to use than XML, but still quite probable to be required by some. JSON seems to have the most support in NoSQL databases, so there’s definitely a potential use case in exporting data from EverBEEN and uploading them to some document storage system.

As a traditional format, CSV still has the best support for imports to array-oriented software, such as R Project. This alone is a reason to implement support for it, not to mention that it is probably the only of the three implemented formats that enables Microsoft Office users to import EverBEEN data into Microsoft Excel.

Seen as the different formats use different software packages, I decided to separate them into independent modules. That way, the dependency on serialization software packages are kept apart. However, the querying logic, along with all associated classes and metadata, remains the same for all export modules, and is reused in data processing as well. Therefore, this codebase is extracted to the data-processing-commons module.

2.3 Extractor/processor skeleton

To establish the above said common codebase for data extractors/processors, I started implementing the following features for sharing:

Reflection property loading  So far, the property loading in tasks has been slightly too verbose to my taste. I implemented the ProcessingConfigurationParser, which can use the task properties to automatically load typed fields into subclasses of ProcessingConfiguration using Reflection. The ProcessingConfiguration already contains the attributes necessary for loading results, so by extending this class later for data processors, I will only need to add processing-specific fields. The ProcessingConfigurationParser will take care of loading the values from the task descriptor for me. The only thing I will have to worry about later on is that the task descriptor properties match my ProcessingConfiguration override.
Logic service loading  As I already explained in section 2.2, the data selection logic remains exactly the same, no matter what happens with selected data afterward. This counts both for both data extraction and data processing. Therefore, I implemented a generic data selection task called DataProcessor, which performs the data selection and then loads a processing logic implementation using a ServiceLoader over the DataProcessorLogic interface. The DataProcessorLogic interface contains a general contract for all processors:

getName()
Every data processor should be named, to allow its identification in logging messages, exceptions etc.

createConfig()
Seen as every DataProcessorLogic may use a different subclass of the DataProcessorConfiguration and implementing ServiceLoader instantiation seems unnecessarily complicated, it seems suitable that every DataProcessorLogic create its own configuration object. This object’s allocation is invoked by the generic DataProcessor implementation, filled from task properties using the ProcessingConfigurationParser and then reinjected back to the logic instance when DataProcessorLogic#process() is invoked.

process()
The main logic execution code. Receives the unpacked dataset, performs the processing and produces an EvaluatorResult, which is essentially a wrapper over a file (in byte [] form), containing metadata such as identifiers, creation timestamp and MIME type.

It is left at the discretion of the developer extending the data-processing-commons package to ensure that only the correct DataProcessorLogic implementation is available on the classpath when the ServiceLoader scans for it. This is somewhat limiting in case of need for multiple logic variants, but even then, it is hardly impossible to switch some logic branches based on the DataProcessorConfiguration if need be, seen as the configuration object is passed to the process() method with all the logic-specific configuration properties.

Task descriptor inheritance  While ensuring inheritance for configuration objects and logic implementations is fairly simple, I quickly realized this will be more of an issue for task descriptor extensions. When we were implementing EverBEEN, we worried about easy BPK creation and the understandability of the Task API, but we didn’t give much thought to task inheritance at all.

Given the common task property base, I considered it best for these properties to be defined in the data-processing-commons module, and for extending extractors/processors to simply add their logic-specific property declarations to the common base. Sadly, the only way to do this properly was to include an XSL transformation
step to each module build descriptor (pom.xml) and declare the additional task properties, as well as the changes in naming convention, in the XSL stylesheet.

Such a stylesheet is present in src/main/xsl/taskDescriptorCustomization.xsl in each of the following modules:

- export-xml
- export-json
- export-csv
- aggregator

For the export tasks, its role is mainly in changing the naming convention. In the aggregator module, I also add some additional task properties related to data processing.
3. Data processing requirements

The former implementation of EverBEEN has been designed to provide support for storing user-designed data transfer objects. That means that a user implementing a benchmarking task can design his own data carrier class and pass it to the the EverBEEN Task API for storing. He/she can also access it using his carrier class after retrieving it from persistence layer from another task.

However, as we elaborated in chapter [1], the framework itself is strictly type-agnostic (whenever I use this term, I mean unaware of user types, as opposed to types in general), as the Task API uses the Jackson serialization library to map user-typed objects to JSON. The EverBEEN framework then moves user objects around in marshaled form. The bytecode containing user type definition only ever runs in a separate JVM as a task – it never reaches the framework JVM. The only EverBEEN component other than Task API that works with these objects in a different form than their textual representation is the Mongo Storage, which in its turn only converts them between String and BSONObject.

As such, the EverBEEN framework cannot hope to interact with user data in any way, simply because it is oblivious to their structure and type.

3.1 Partial type accomodation

The first step to making EverBEEN capable of processing user data therefore is to convey it some knowledge on the structure and type of data it is retrieving/processing. Obviously, I cannot hope to reconstruct the user type as-is. This would force me to load user type byte code into the EverBEEN JVMs, which would, in turn, cause inevitable trouble with class versioning – the exact reason why we decided to make the framework type-agnostic in the first place. However, the type/structure inference mechanism must be broad enough to accomodate various uses.

The criteria of flexibility and simplicity are joined in what I will call a dataset. The dataset is essentially a list-like collection of flat records (records strictly composed of primitive Java types). In code, we represent a record as such:

```java
Map<String, Object> record;
```

This representation of a record allows us to quickly access its fields through their field name, as well as it is capable of accomodating all primitive Java types. Indeed, many libraries and/or frameworks, such as Jackson, or Spring JDBC, use this interpretation of a user type as a staging point during marshaling/unmarshaling.

A dataset therefore assumes the following structure:
Arguably, it would be more memory-efficient to store records in a

Map<String, Collection<Object>> alternativeDataset;

but our chosen approach shines in two points:

- It maintains record integrity. My records are stored in a collection. Not across multiple collections as dissociated fields.
- It doesn’t require the allocation of additional objects when a singular record is extracted.

It is, once again, arguable to what extent record integrity really benefits my effort, but the latter point revealed itself as quite important when implementing data processors.

Of course, we might not need the entire user object to perform the processing. In the next section, I’ll explain in more detail how I make use of that fact to limit EverBEEN network traffic resulting from data retrieval.

### 3.2 Type inference

As the user is the author of the types we will be mapping into records, the user must be the origin of type information. There presumably are various ways of approaching the way the user inputs type information into the EverBEEN framework to make it partially type-aware. I mainly considered two.

#### 3.2.1 Persistent vs. Temporary

Based on the longevity of type information, we can divide it to two types: Persistent and Temporary. For temporary longevity, the lifecycle of type information starts with the task that conveys it to EverBEEN and ends when the task terminates (with the exception of its result, as seen in section 3.3). For persistent longevity, the user gets a means of viewing the result groups in persistence layer, their fields etc. and gets to label them with types.
Both of the approaches have many pros and cons, but the essential argument that
decided the choice in my implementation was, in the end, the same one that made
our previous team make EverBEEN type-agnostic. The code of the tasks (user code,
responsible for running benchmarks) can evolve over time. The framework’s lifecycle
exceeds the lifecycle of tasks executed on it by far. This counts sevenfold for the
framework’s persistence layer. There are hardly any guarantees that, over the course
of time, type inference information will stay backwards compatible.

The temporary longevity, which basically means re-inferring the type information with
every launch of a data processing task, intrinsically bypasses issues of backwards com-
patibility. A processing task’s configuration can quite easily evolve with the rest of
the software developed by the user, and potentially change type inference informa-
tion based on the nature of the data the user’s benchmarking tasks produce. With
persistent information, this kind of plasticity disappears very quickly, unless features
such as type inference versioning are implemented etc.

The above arguments made me choose the temporary type inference mechanism over
the persistent one. While I am aware that a persistent type inference would be much
more comfortable for the user, I am convinced that the flexibility of the temporary
approach yields more benefit over time, despite the higher learning curve this infers
on the user while first starting with data processors.

A marginal, yet still important observation to make, is that the temporary approach
also enables different tasks to interpret the same data differently. With persistent
storage, this would likely cause conflicts (unless named type configurations were in-
troduced). With temporary type inference, this causes no problem at all, seen as
every task gets its type mapping directly from its parameters.

### 3.3 Dataset persistence

An exception to the rule of temporary type inference is the DataSetResult. This
is a direct subclass of the EvaluatorResult, and is used to persistently store the
output of data processors. Hence I already know, during the processing, what types
of data I am going to produce, I use the DataSetResult to save type metadata
along with the data itself.

While one might argue that this violates the principles described in section 3.2 it is
commendable to observe that the only data affected by type metadata persistence in
such manner are data that have already been processed, and are stored apart from
other results in a file. The DataSetResult is one of the latest additions to the data
processing toolkit, and is aimed at giving the user the possibility to export processed
data in a different format that the native JSON. It is also the hope that these pre-
aggregated results might be picked up later by chart plotters for visualization.
3.4 Implementation notes

Of course, I could not possibly hope for processing data and, at the same time, maintaining total genericity. Therefore I had to limit what data types is the user going to be able to infer. I will call the allowed types *primitives*, because the map to primitive Java types. To verify that no other type mapping comes in user data during type inference, I implemented a smart enum called PrimitiveType, which is present in the *results* module, along with the ResultMapping. This smart enum is used to map incoming values on its types, and, if none can be found, to react with an exception.

Within the framework itself, the type mapping is stored in the ResultMapping, whose composition is very simple:

```java
public class ResultMapping {
    private final Map<String, String> typeMapping;
    private final Map<String, String> aliases;

    //...
}
```

This helps avoid trouble when serializing/deserializing. Whenever need arises to get the actual runtime class and operate with *RTTI*, the conversion methods from PrimitiveType are used, thus ensuring proper conversion and eventual error handling, should type information be invalid. To view which runtime types are available, the reader is advised to consult the appendix C.

The general contract of ResultMapping is that only fields that have a type mapping are retrieved and can be aliased. If a field is not present in the type mapping, it does not get loaded from the persistence layer at all.
4. Data processing model

In this chapter, I will describe the theoretical model behind generic data processing, as I implemented it in attached source code, as well as clarify why I chose this particular implementation method.

4.1 Arithmetics

In chapter 3, I conceded that generic data processing over fully customizable data types is unnecessarily complex, and explained the decision to instead give the user a means of reducing persistent objects to fields currently of interest and enabling him to cast these to Java primitives. This decision, of course, comes with a whole set of ramifications.

First and foremost, let us agree that as far as aggregation processing goes, the main area of interest are numeric types. Sadly, the Java language doesn’t provide a tool to effectively perform arithmetics on generic numeric types. Also, the type casting mechanism I provided EverBEEN users with handles runtime type information, effectively holding a

Map<String, Class<?>>

mapping and using it to cast data on the go, thus verifying type integrity. The aforementioned facts almost completely exclude the effective use of Java language generics.

The task of enabling data aggregation over generic numerics is thus twofold.

- First, a generic arithmetic unit interface must be designed, which will cover necessary arithmetic operations performable on all primitive Java numerics, yet enable type-specific implementations. This is desirable simply to eliminate reimplementations of type-aware arithmetics for every data processing function.

- Second, an engine needs to be created, which will enable various aggregation metrics to perform computation.

The nature of implementation of above-said generic arithmetic unit naturally influences the operations available in data processing functions, as well as their efficiency. I will examine the computation engine in detail in section 4.2. For now, let us ponder the options of implementing a common arithmetic codebase.

There are mainly two possibilities of implementing generic arithmetics:
**type-specific**

Revolves around defining an interface using Java language generics and implementing arithmetic operations restricted to one given type. Such an implementation must be created for each numeric primitive. This approach does not support operand type combinations.

**type-combining**

Consists of creating a completely generic arithmetic facade with methods similar to

```java
Object operation(Object operand1, Object operand2) {
    ...
}
```

and within the operation, checking the runtime type of each operand to determine the actual steps to implementing the operation. Also, a suitable return type must then be chosen.

While the **type-combining** approach certainly offers more flexibility, I chose the **type-specific** implementation for multiple reasons:

- Differently from the **type-combining** approach, the **type-specific** approach actually enables us to make use of type guarantees provided during data selection, rather than reexamining RTTI again.
- It doesn’t need to perform several RTTI (runtime type information) verifications on every operand, which results in better performance.
- The implementation of a **type-combining** arithmetic unit would result in type-based combinatoric explosion of operand type cases, some of which, for some operations, do not even have clear semantics, e.g.

```java
Object subtract(Object operand1, Object operand2)
```

called with parameters

```java
Object res = subtract(10, BigDecimal.valueOf(157, 2));
```

These use-specific corner-case would need to be handled by either unintuitive default conversions, which can lead to much confusion, or exceptions, which drag in a lot of extra complexity.

Having chosen the **type-specific** approach, I designed the generic interface and its type-specific implementations in the following classes (**aggregator** module):

```java


cz.everbeen.processing.arithmetics.Arithmetics

cz.everbeen.processing.arithmetics.ArithmeticsFactory
```

20
Naturally, choosing this approach leaves all type-combining operations up to the implementations of individual aggregation functions. These have the choice to either implement less typical operations by themselves, or cast the data to conform to their disponible arithmetic units. Either way, it was desirable not to couple the arithmetic unit and the processing engine too tightly. Therefore, above described arithmetic unit is not a part of the following model of the processing engine, as aggregation functions may choose to use the unit or not.

4.2 Processing engine

Next, I was confronted with the task of creating the support engine that actually executes aggregation operations. While it was technically possible to create a separate task (module and BPK build) for each metric, this seemed clumsy, seen as the user might wish to combine multiple aggregations in one dataset, and it is hardly accommodating that he/she have to set up multiple tasks to do so. In the end, keeping the implementations of aggregation operations proved to be easier for code maintenance as well.

4.2.1 Target

While it would be great to have at our disposal a data processing toolkit in the magnitude of R Project or GNU Octave, it is also obvious why attempting to implement such a toolkit in the timespan of one thesis is not doable. Nor does it actually serve any purpose.

With the regression benchmarking scenario in mind, where does data processing stand? Obviously, when it comes to statistically analyzing data, EverBEEN’s data processing toolkit is going to fall short in comparison with standard tools. But it is important to keep in mind that the added value of having a data processing toolkit present in the framework lies not in automating statistical analysis, but rather in detecting its necessity.

Indeed, when performing statistical analysis of benchmarked data in hope of determining the reason of an application’s performance shortcoming, one would often proceed based on what he/she discovers about the analyzed data itself. There is no point in attempting to automate this process, or reimplementing existing tools to do so.

Yet, one does not necessarily always want to perform a statistical analysis per-se. The essential benefit of having a data processing tool in EverBEEN lies with furthering possibilities of evaluating compliance of benchmarked software to its quality goals. Seen as these goals are intrinsically a set of well known boundaries within which well defined performance characteristics of benchmarked software should fit,
processing benchmarked values into aggregates defining these characteristics is key to continuously evaluating compliance to quality goals.

The evaluation of software characteristics in above described sense does not require overly complicated statistics. Most of these characteristics can be produced using fairly simple aggregation methods, such as averages and quantiles. Therefore, this is the kind of data processing I attempted to enable when designing below described processing engine.

4.2.2 Concentrator/Aggregator model

As I mentioned in section 4.2, separating each type of data processing into an isolated task would lead to poor maintainability and significant functional limitations. Thus, I created a generic aggregator module containing the processing engine, as well as implementations of arithmetic operations and individual data processing functions. Albeit contained in the same module, data processing functions are modularized to form an extension point. While it is unlikely (yet still possible) that users would extend data processing functions, as opposed to creating an evaluator task tailored to their specific needs, the presence of the extension point makes it significantly easier to add new data aggregating strategies as EverBEEN develops.

All implementations being present in one module (and, subsequently, in one task), I had to give the user a means of conveying information on what kind of processing should the task perform. While I entertained the idea of creating a minimalistic expression language to enable the user to perform simple arithmetic operations on arguments passed to aggregating functions, I abandoned the idea for the sake of simplicity. This decision is arguably a shortcut and having such a language would benefit the user. Yet, the design of such language must be carefully considered, as being too liberal about options given (e.g., function composition) might promote complex computations over simple, linear-pass aggregations which I am focusing on in this thesis. Therefore, I have placed this feature on the project’s roadmap for future reconsideration. As the implementation stands currently, the user can specify two lists of function-like expressions: a concentrator list and an aggregator list.

The idea of concentrator/aggregator division is not a novelty – it is heavily inspired by common database analytic queries, such as map-reduce, or

SELECT
    aggregator1( value1),
    aggregator2( value2),
    concentrator1( value3),
    concentrator2( value4)
FROM
    abc
WHERE

    xyz

GROUP BY

    concentrator1( value3),
    concentrator2( value4);

in traditional relational databases. The semantics of a concentrator are to potentially reduce the domain of a variable to a smaller set or, in other words, to "categorize" it. The aggregator amasses records for different category combinations (cartesian product of concentrator value domains) and performs statistical calculations on them.

The concentrator and aggregator contracts can be found in

```java
    cz.everbeen.processing.Concentrator
    cz.everbeen.processing.Aggregator
```

interfaces. Their implementations correspond to individual processing functions. Implementations that require work with numerics are themselves responsible for instantiating their own Arithmetics instances and supply additional computational support in case that Arithmetics do not suffice.

The idea of the internal extension point I mentioned earlier in this section is to use the Java ServiceLoader to load available function implementations. However, the creation of a concentrator or an aggregator may be quite complex and may, in the process, throw configuration exceptions. This is why ServiceLoader is not used to directly load function implementations. Instead, I designed a pair of factory interfaces, which are easily constructed and can, in their turn, produce concentrator and aggregator instances:

```java
    cz.everbeen.processing.ConcentratorFactory
    cz.everbeen.processing.AggregatorFactory
```

These factories are organized into a "Chain of Command" pattern (as seen in Design Patterns [2]), according to their #loadPriority() rating. As seen in the factory protocols, the factory objects contain an evaluation method called #evaluatesAlias(String expression). When the chain of command receives an expression, it iterates through all available factories (ordered by descending priority) and uses the #evaluatesAlias() method to determine the first factory which can be used to instantiate the desired concentrator/aggregator.
5. Implemented processors

I implemented several data processors (concentrator and aggregator implementations) in the aggregator module. Here are their descriptions.

5.1 Concentrators

Let us remember that the idea behind concentrators is to reduce the domain of a variable, thus compacting the entire dataset.

5.1.1 id

Invocation:

VAR_NAME

The simplest imaginable concentrator, the id concentrator only relays the value it is passed. It exists so that processed data can be grouped by values with uncompacted domain. Also, for some data types, such as string, no other concentration policy is available, anyway. Grouping is done by actual value.

5.1.2 norm

Invocation:

norm(VAR_NAME, INTERVAL_COUNT)

Normal distribution over the variable. Splits the domain of VAR_NAME in current dataset into INTERVAL_COUNT intervals evenly. Grouping is done by interval.

5.2 Aggregators

Aggregators perform aggregation functions on subsets of the dataset with distinct values on concentrator results.
5.2.1 count

A simple counter for processed entries. Invocation:

count()

5.2.2 sum

Sums numeric values for identic concentrator values. Invocation:

sum(VAR_NAME)

5.2.3 avg

Calculates the average on numeric values. Invocation:

avg(VAR_NAME)

For the parameters that can be passed to the *aggregator task*, please consult the appendix [B].
6. Benchmarking automation

While the central theme of this thesis is to implement data processing capabilities, I also spent a great deal of time analyzing the possibilities of automating the control of the EverBEEN framework. I would like to dedicate this chapter to explaining why data processing and control automation themes are intertwined and what use cases do they enable the framework to perform.

6.1 Why automation?

Of the time I spent working on this thesis, I consecrated a great deal to attempting to set up an integration testing scenario for EverBEEN, using Jenkins CI Server for automated Maven builds and automatic EverBEEN redeployment in a pseudo-distributed environment. While this part had some success, as can be seen on http://cit.darklight.cz, the overall goal of integration testing proved way beyond the time horizon of this thesis. The hope was to use these automated builds and redeployments to test the new EverBEEN revision against a set of integration tests, mainly aimed at launching tasks and task contexts, verifying results, launching aggregation metrics, verifying their results etc.

As ever, integration testing of a complex networked application proved too time-consuming and I had to abandon the pursuit. One of the main reasons was that differently from WillBEEN, which featured a sophisticated command-line interface, EverBEEN lacked any means of control-flow automation. As we performed a complete overhaul when implementing EverBEEN, rather than just refactoring, we had to cut a lot of features, the command-line interface being one of them.

As I was pondering testing automation, I noticed another, perhaps even more important use-case for machine-driven control-flow: Continuous Integration. Let us ponder the advantage of having a data processing tool integrated into EverBEEN, should the user only ever launch data processing manually. If a manually initiated analysis is what the user is aiming for, wouldn’t he/she be better of with simply exporting his/her data from EverBEEN and loading them into an offline analysis tool, such as R Project or GNU Octave? The major advantage I perceive in EverBEEN having an integrated, albeit small, data processing toolkit is that data processing tasks or task contexts are inherently repeatable, with little to no extra effort on the user’s part. This makes them particularly suitable for Continuous Integration scenarios, as opposed to data exports and offline data processing which, although entirely possible, require the user to do a lot of setting up and handling launch triggering etc.

While Continuous Integration is a scenario we kept in mind while implementing EverBEEN, we reckoned that it is possible to implement a benchmark with a listener/poller, which would trigger a task context upon receiving/detecting payload from
the CIT and user-implemented tasks would handle the rest. Once this possibility was outlined and ensured, we quickly dismissed any other support for this use-case for lack of time.

While this way of integrating EverBEEN into continuous integration is definitely possible, it is by far not the easiest one, let alone the most comfortable for the user. Again, to integrate EverBEEN and his/her benchmarks, the user would be required to develop most of the orchestration on his/her own, including the listener/poller benchmark.

Compared to this approach, which requires extra work from the user, as well as extra knowledge of the EverBEEN framework and his/her CIT, a well-designed, easy to use control-flow automation tool would solve above issues with relative elegance.

### 6.2 Means of automation

As I mentioned in section 6.1, WillBEEN featured a pretty rich command-line interface, whose main purpose was to allow controlling WillBEEN from batches. This command-line interface implemented in C for speed. Indeed, when it comes to batch-executions, Java is not the optimal technology to use, due to long JVM boot sequence.

While a native executable is a valid way to implement such a feature, EverBEEN already has a quite solid utility wrapper implemented around the instantiation and configuration of Hazelcast – EverBEEN’s clustering back-end – and a rather elaborated facade grouping the control-flow logic on the entire cluster. Implementing a batch client in another language would mean to reimplement all of these features as well, leading to code duplication and giving opportunity for discrepancies to occur.

The above reasons led me to the idea of creating a REST-ful web service offering an API for control flow manipulation. This solution marries efficiency with universality, as HTTP requests can come from a wide variety of sources, while allowing me to reuse the control-flow facade implemented in EverBEEN, as I could implement the service in Java. Additionally, the general contract of REST-ful web services is wide-known, thus making the automation tool more intuitive.
7. REST interface

As I explained in chapter 6, having a REST-ful web service to enable automated control of the EverBEEN cluster would greatly benefit the Continuous Integration use-case, which I pledged to promote in the introduction of this thesis. Despite the fact that the Task API is still largely in testing and does not cover the full control options of the EverBEEN cluster, I decided to include it in the thesis to better illustrate the possibilities achievable once it fully develops.

7.1 Selected technologies

To best make use of the implemented control facade that EverBEEN implements, I was bound to use Java for the Task API server-side implementation. Therefore, it was decided that I will be implementing it as a standard Java Servlet application, so that it is deployable on all standard Servlet containers. Next, I had to decide what format to use for protocol.

Implementing an entirely custom protocol has a lot of drawbacks, including error-proneness, code duplication client-side and a lot of extra work with little benefit. Thus, the choice was really between JSON and XML. Both present a strong case.

XML    XML is way more standardized and offers better means of validation. In addition, the namespace-awareness of the language enables for better extendability. Last but not least, the possibility of generating Java code from XSD using binding technologies such as JAXB makes it possible to declare a strict protocol and, by integrating JAXB conversions into the build, have certitude that the service complies.

JSON    Compared to XML, JSON is much less verbose, better readable and much easier to write by hand. This provides for additional flexibility, given that JSON snippets can, if necessary, be written by hand, thus enabling effective use of the Task API from batch execution environments such as BASH. One of the most interesting advantages of JSON is the synergy with AJAX. Shortly after implementing EverBEEN, we agreed that the Tapestry framework, which we used to implement the Web Interface, was a poor choice, as it is slowly becoming obsolete. Having the Task API provide JSON responses would provide seamless support for integration with an AJAX-based front-end, thus integrating the two presentation layers (Task API and Web Interface) into one.

The brevity of JSON and its potential synergy with AJAX eventually tipped off the scales, so I decided to implement the Task API as a JSON REST-ful service. To facilitate potential connections from Java clients, I extracted the protocol objects (with JSON mappings) into a separate module called rest-protocol, whereas the implementation of the server itself can be found in the rest-interface module.
7.2 Covered functionality

During the course of this thesis, the focus of functionality covered by the Task API has primarily laid with enabling the integration testing, therefore, the covered functions more or less match that scenario.

**Info handler**
Provides information about the cluster and available services.

- **@GET /info/config**
  displays cluster configuration information

- **@GET /info/status**
  displays cluster connection status

- **@GET /info/members**
  displays active cluster members

- **@GET /info/services**
  displays information about disponible services

**BPK handler**
The REST interface supports uploading and downloading and listing BPKs using these mappings in the BPKHandler:

- **@GET /bpk/{groupId}/{bpkId}/{version}**
  downloads BPK

- **@GET /bpk**
  list BPKs

- **@PUT /bpk**
  uploads BPK

- **@GET /bpk/{groupId}/{bpkId}/{version}/td**
  list task descriptors in BPK
**Task handler**
The Task handler provides information and control-flow over tasks in the EverBEEN cluster.

@GET /task/{taskId}
   display task status

@GET /task/{taskId}/logs
   display task logs

@GET /task
   list active tasks

@PUT /task/{groupId}/{bpkId}/{version}/{tdName}
   run task by named descriptor in BPK

**Result handler**
Result handler enables user to download results.

@GET /result/{resultId}
   download result
8. Extracurricular features

8.1 Software Repository binding configuration

In the former EverBEEN implementation, the binding interface and port of the Software Repository component weren’t configurable. This has interfered with my attempts at setting up an integration testing scenario for EverBEEN (using Jenkins CI Server). I therefore implemented additional configuration options, enabling binding customization (IPv4, IPv6).

This feature has its uses even outside of testing scenarios – for instance, one might wish to implement benchmarks which require strict separations of EverBEEN network traffic from that of the benchmark itself. In such cases, additional control over interface binding is also desirable.

8.2 Task Descriptor XSD cleanup

While rereading the task-descriptor.xsd, I came across some unused attribute group definitions. Therefore, I cleaned them up to avoid further confusion.

8.3 Improved logging

In the interest of easing troubleshooting when I was working on the Task API, I improved error logging in several components, including Task API, Object Repository, Software Repository and Host Runtime.

8.4 Bugfixes

- Fixed bug where default value wasn’t properly loaded for configuration properties initialized as empty strings, rather than null.
- Fixed misleading in-code documentation on returned id after the submission of a single task.
- Removed NullPointerException problem when printing persistence query objects.
- Due to MongoDB adding a "_id" field to records, there were problems when deserializing object with default Jackson unmarshaler settings. Removed the issue by cropping the database’s internal "_id" directly after object retrieval.
9. Notes on digital content

As during this thesis, I made modifications to the source code of the EverBEEN framework, I attached the entire EverBEEN codebase and compiled binaries, in their modified state, to this thesis. To avoid confusion, I would like to briefly describe the digital content attached to the thesis, as well as identify where the changes have been made.

The electronic content attached to this thesis is structured as follows:

```
/_dist/ contains compiled everBeen and bpkPlugin jars/wars and task BPKs
/_doc/ contains PDFs of this thesis and the everBeen project documentation
/_src/ contains source code of the everBeen framework and data processors developed for this thesis
```

9.1 Source

The source section is organized as follows:

```
/_src/
  \_bpkPlugin/ source code of the BPK Maven plugin used for packaging BPKs as part of a Maven build
  \_dataProcessors/ source code of data processors created in this thesis
  \_everBeen/ source code of the EverBEEN framework
```

9.1.1 EverBEEN source code additions

The EverBEEN framework source code is, for the major part, in the same state in which it found itself in September 2013, with the exception of a few components that required modifying in order to provide support for data processing:

- **mongo-storage** – Some bugfixes for smooth recursive object querying took place there
• **results** – Where the entire type model, including PrimitiveType and ResultMapping have been added.

• **task-api** – A lot of support needed to be added to this module to enable type inferences and aliasing, especially when it came to JSON marshaling/unmarshaling support.

• **util** – Additional support for JSON deserialization was added to JSONUtil

• **rest-protocol** – Brand new package including protocol object definitions for the Task API.

• **rest-interface** – Also brand new, contains the implementation of the Task API handlers.

### 9.1.2 Data processors

The **dataProcessors** part of the project is entirely new, so I will briefly describe each module and its role in data processing.

**data-processing-commons** By far the most important of the packages, as it includes the common codebase for all data extractors/processors. It includes the DataProcessor class which handles the creation of persistence fetch queries, as well as operates the ServiceLoaders to load the appropriate logic implementation. This module also contains the generic data processor task descriptor templates - fromResults.td.xml and fromDataset.td.xml.

**export-xml export-json export-csv** As the name suggests, these are export modules for different formats. Each of them has its own data serialization implementation and its own taskDescriptorCustomization.xsl, which it uses during the Maven build to override the task descriptor templates from **data-processing-commons**.

**aggregator** This module contains all of the aggregation logic, including ServiceLoader operation for concentrator and aggregator factories, default factory implementation themselves, resulting data cube synthesis etc. Just as the export modules, the aggregator module too implements its own taskDescriptorCustomization.xsl, but uses it to add additional parameters, rather than just rename the task.

**testing-data** A module that contains the definitions of a testing data class and a factory implementation that generates testing data and fills them with random values. This package does not make much sense by itself, it is mostly used when included by the **dummy-data-generator** as a testing data model. There is, however, a reason why it is separate. Originally, I had hoped to use this as a protocol, shared to integration testing tasks, as well as data generators, as a common codebase.
**dummy-data-generator**  As the name suggests, this is a task that enables the user to fill a result group of choice with some garbage data, which he can then query. This generator was used for all of the export and aggregator testing.

### 9.2 Binaries

Associated binaries are found in the dist folder.

```
/ 
  \_dist/ 
    \_ebeen/ 
      \_been.jar main everBeen JAR 
      \_been-wi.war everBeen Web Interface war 
      \_been-rest.war everBeen Rest API war 
  \_dproc/ 
    \_export-xml.1.0.0-DPROC.bpk XML export task 
    \_export-json.1.0.0-DPROC.bpk JSON export task 
    \_export-csv.1.0.0-DPROC.bpk CSV export task 
    \_aggregator.1.0.0-DPROC.bpk reusable aggregator task 
  \_dummy/ 
    \_dummy-data-generator-1.0.0-DPROC.bpk task for dummy data generation
```

### 9.3 Documentation

In the doc folder, I included a compiled PDF of the EverBEEN Project Documentation [3], as well as a digital copy of this thesis.

### 9.4 Deployment and use guidelines

To deploy EverBEEN you will need a JRE, version 1.7.x or higher. The executable JARs of the entire project can be found in dist/ebeen. The easiest way to run the framework is to run all services on a single node and keep all settings default, like such:

```
java -jar dist/ebeen/een.jar -rr -sw -r
```
To connect to **EverBEEN** with its Web Interface, the easiest way is to run the **jetty** server embedded into the Web Interface’s WAR archive:

```
java -jar dist/ebeen/been-wi.war
```

The embedded server binds on `0.0.0.0:8080` by default, so be sure to disable anything potentially running on that port. Alternatively, one can of course deploy the `been-wi.war` to an existing servlet container, if available, to avoid port conflicts.

Once the Web Interface is running, it needs to be connected to the running **EverBEEN** cluster. This is done on the **connect** screen, which is automatically displayed once the user connects to the Web Interface for the first time (http://localhost:8080/). The default settings should connect to a cluster launched in the above described manner.

For more details on the deployment and more complex ways of launching and setting up the framework, please consult the **EverBEEN** Project Documentation [3], also present on attached CD.

To upload data processor **BPKs**, use the Web Interface’s **Packages** tab. To launch **tasks** from the **BPKs**, use the **Benchmarks & Tasks** tab. For better understanding of the **task properties** found in data processors, please consult appendix [3] Alternatively, if hover the mouse cursor over a property name in the Web Interface shows a descriptive tooltip informing about its purpose.

Should you wish to test the Task **API** module, it is necessary to deploy the `-been-rest.war` to a servlet container. The default connection settings for cluster binding are the same as for the Web Interface and should work in default settings. The connection settings for the Task **API** are changed in the web application’s `context.xml`.

### 9.5 Task development guidelines

For tutorials on how to create **EverBEEN** tasks, please consult the tutorial section on http://www.everbeen.cz. Please be aware that any tasks created using the **Maven** archetype will be referencing the official release versions of **EverBEEN** - currently 3.0.0. To develop tasks against Task **API** versions from this thesis, one must change the dependency versions from 3.0.0 to 3.1.0–DPROC. Additionally, the 3.1.0–DPROC version must be installed in the local **Maven** repository. The **BPK** plugin did not undergo any changes and technically does not need rebuilding, but it is advisable to do so as well. Installation into local repository goes as follows:
cd src/everBeen/
mvn clean install
cd ../bpkPlugin
mvn clean install

The builds may take a while, but this will install the 3.1.0-DPROC version of *Ever-BEEN* into your local repository, from whence it can be loaded to satisfy dependencies of tasks under development.
Conclusion

To sum up on the success ratio of my effort in this thesis, let us look back at the original motivation behind this and evaluate what did I manage to cover and which parts find themselves lacking.

9.6 Retrospective

As far as data exporting goes, I believe the provided formats more than cover potential needs of most users. Should it not suffice, an internal extension point is provided for EverBEEN developers to easily add more formats. In terms of analysis possible manners of implementing type-generic data processing, I considered several approaches and chose one which, to my mind, offers the best marriage of efficiency and flexibility. With the exception of the occasional sloppy implementation, which I sometimes needed to either refactor retrospectively or reimplement from scratch, I believe that the implementation of the generic data processing engine itself was successful as well.

What I find unsatisfactory as far as my realization goes is mostly the lack of improvement in user-friendliness and the poor spectrum of actual data processors (concentrators and aggregators) whose implementation I skipped mostly for debugging JSON serialization and deserialization of type mappings and, of course, for the exploration of my secondary objective - Continuous Integration.

As far as integration testing and CI is concerned, I am almost convinced I spent too much time on the issue, to the detriment of the data processing functionality itself. Although the Task API is an exciting feature that could truly improve user experience in CI, attempting to develop it at the same time as integration tests for EverBEEN proved too time-consuming to pursue anyway.

Despite the shortcomings in depth of realization, I am happy about the implementation patterns I put into place (ServiceLoader extension points, task inheritance prototypes etc.) Over the course of development of the software accompanying this thesis, they proved functional and helpful. It is my hope that they will continue to ease the way for future implementation enhancements.

9.7 Roadmap

As I am sure any thesis does, even mine comes with a very broad roadmap, filled with exciting features, that couldn’t make the cut due to time shortage. I can not possibly fit them all in the conclusion, but I will at least mention the most exciting.
9.7.1 Short term goals

**Merge effort**  My colleague and former EverBEEN team member Tadeáš Palusga is currently working on a thesis that strives to modularize EverBEEN and introduce OSGi to better handle software dependencies within the cluster. This means huge changes to the framework internals and, consequently, will require a similarly huge joint merging effort to integrate the changes we both made separately.

**Better interface for type mapping**  The type mapping interface in the form of a text box is spartan at best. While I was aware of that and at least changed the textual representation to a less obtrusive form (the former type mapping required inputting JSON in the text box), I could not think of a way of improving the visualisation without either violating package integrity and best practices, or spending 50 or more hours implementing an extendable visualisation component system for task implementations. I believe the future of this feature will make itself clearer after the merging effort, once we decide the fate of the Web Interface.

**Synchronous persistence**  So far, EverBEEN only supports what we called ”eventual persistence”. That means that by the time a task persists results, it has no guarantees on whether they will actually be persisted, seen as they may very well linger in the cluster’s distributed memory, while the persistent storage is actually offline. This directly interferes with the possibility of triggering data processing after a build and thus, a way of notification on actual result persistence needs to be devised.

9.7.2 Long term goals

**Task API completion**  The attached version of the Task API implementation is by far not complete – it only covers basic functionality associated with launching tasks, downloading results and uploading BPKs. It is, however, a feature that has the potential to broaden the use-case spectrum of EverBEEN dramatically.

**HTTP callback module**  The Task API provides options for synchronous cluster state polling, but as far as integration with CIT’s is concerned, EverBEEN could benefit from having a configurable HTTP callback module, which it could use to notify its partner CIT about cluster events, such as the completion of a task or task context, task context failures, finished data transfers etc. These notifications could in turn trigger builds on the CIT, thus achieving effective two-way communication.
Bibliography


Appendices
A Glossary

benchmark
A special task capable of producing task contexts and scheduling them in the EverBEEN framework.

BPK
An EverBEEN software package. Contains user code and may contain task descriptor and task context descriptor templates.

task context
Configuration and execution ordering of a grouping of tasks. Configured by an XML called task context descriptor. Contexts group tasks to express data dependencies, ensure execution ordering, guarantee simultaneous startup etc.

task context descriptor
XML configuration of the execution of a task context.

dataset
A generic record collection, implemented by Collection<Map<String, Object>>, accompanied by RTTI in the form of Map<String, Class<?>>.

Host Runtime
An overseer module responsible for running tasks on the cluster node it runs on (spawning task JVMs/processes, concentrating logs etc.)

Object Repository
The high-level component residing over the EverBEEN cluster’s persistence layer. Handles operation on shared data structures and provides access to persistent storage.

result
A record produced and persisted by a task during its execution. The record must extend the Result class (found in Task API).

Software Repository
A component that distributes BPKs across the cluster to requesting Host Run-times.

Task API
An API through which EverBEEN functionality is provided to user code. For Java tasks, this is the Task extension point and associated utilities, such as the ResultFacade.

task
The smallest user code execution unit. Consists of the code itself and a task descriptor – an XML configuration of the task’s execution.

task descriptor
XML configuration of the execution of a task

task properties
XML <property .../> elements containing static properties of a task’s JVM.
B  Data processor properties

This appendix serves as an overview of data processor task properties.

All data extractors/processors share the same selection variables. They can either be run with the following properties using the fromResults.td.xml.

B.1  Selectors

taskId
ID of the task whose results we want to analyze

contextId
ID of the task context whose results we want to analyze

benchmarkId
ID of the benchmark whose results we want to analyze

fromDate
Lower bound for limiting selected results’ date of creation

toDate
Upper bound for limiting selected results’ date of creation

groupId
ID of the result group; Mandatory

typeMapping
A ’|’ separated list of

KEY -> PRIMITIVE_TYPE_ALIAS

mappings. For more information on available primitive types, see appendix C.

aliasMapping
A ’|’ separated list of

ALIAS -> KEY

mappings. Serves for renaming variables.

or using fromDataset.td.xml with a datasetId, which is the ID of a dataset previously generated by another task.
B.2 Processors

Data processors possess two additional task properties:

**concentratorFields**
A ‘|’ separated list of

```concentrator(VAR...)```

statements. Concentrators reduce the chosen variable’s domain to a smaller one. Alternatively, only `VAR` can be chosen. This is an "identity concentrator", which keeps the variable as is.

**aggregatorFields**
A ‘|’ separated list of

```aggregator(VAR...)```

statements. Aggregators perform simple statistics on passed values.

B.3 Generators

The dummy data generator task requires two following task properties:

**groupId**
Identifier of the target group where generated results should be stored.

**count**
Number of results to generate.
## Primitive types

<table>
<thead>
<tr>
<th>NAME</th>
<th>ALIAS</th>
<th>ARITHMETIC</th>
<th>JAVA TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>int</td>
<td>Y</td>
<td>java.lang.Integer</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>Y</td>
<td>java.lang.Long</td>
</tr>
<tr>
<td></td>
<td>bigint</td>
<td>Y</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>FLOAT</td>
<td>float</td>
<td>Y</td>
<td>java.lang.Float</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double</td>
<td>Y</td>
<td>java.lang.Double</td>
</tr>
<tr>
<td></td>
<td>bigdec</td>
<td>Y</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>STRING</td>
<td>string</td>
<td>N</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>DATE</td>
<td>date</td>
<td>N</td>
<td>java.util.Date</td>
</tr>
</tbody>
</table>