Proceedings of the 2014 Federated Conference on Computer Science and Information Systems pp. 1621–1628
DOI: 10.15439/2014F278
ACSIS, Vol. 2

Stormgen - A Domain Specific Language to create ad-hoc Storm Topologies

Siddharth Santurkar, Abhishek Arora and K Chandrasekaran
Department of Computer Science and Engineering
National Institute of Technology, Karnataka, Surathkal, India
siddharth.santurkar@ieee.org; abhishekarora185@gmail.com; kchnitk@ieee.org

***Abstract*—Large-scale distributed data processing has gained significant momentum in research in the past decade. With the introduction of MapReduce, many frameworks have been developed that either implement MapReduce or provide additional functionalities useful in a larger domain. While the framework introduced in the MapReduce paper performs batch-processing of data, Apache Storm performs real-time computation on data. Storm does this with the help of Topologies, and the constituents of the Topology are developed using General-purpose Programming Languages (GPL). A Domain-specific Language (DSL) can provide a higher level of abstraction over GPLs and model the specialized features of a particular domain in a better way. In this paper, we propose the development of Storm Topology generator (Stormgen), a DSL for Storm Topology development, and show how the specifications of this DSL can be utilized during the code generation of exact Storm Topology components in Java. The parser and code generator for Stormgen's syntax are developed using the Eclipse Modelling Framework. The practical use of Stormgen is illustrated with a case study which considers the modelling of a Topology for the Word Count application.**

***Index Terms*—Domain-specific modelling, Languages, Eclipse Modelling Framework, Apache Storm, Code generation**

I. Introduction

THE ADVENT of big data analysis created a need for making systems that handled such data in a fault tolerant and distributed manner. Google's paper [1], which introduced the MapReduce programming model, and Yahoo! [2], with its Hadoop framework, succeeded in meeting this need. Several additions and improvements to Hadoop and other such frameworks have been made, owing to the diversity in the meaning and purpose of the data being analysed. In the case of real time, event-based and unbounded data, such a task must be approached in a manner different from that advocated by the frameworks named above. Apache S4 and Apache Storm, apart from a few other frameworks, have made this possible.

MapReduce eases the development of parallel batch-processing programs that conform to the map-reduce programming paradigm. Storm [3] can be used for real-time processing of data for a wide set of data processing use cases, some of which are listed below:

1) Stream processing, which includes processing of messages, updating of databases, etc.
2) Continuous computation, where data streams can be queried continuously and results can be streamed into clients.
3) Distributed Remote Procedure Calls (RPC), where the processing of a complicated query can be distributed and parallelized.

All these features can be easily implemented using the simple, yet, powerful primitives provided by Storm.

Storm involves the creation of a Topology [4] of computation, i.e. a graph with nodes and directed edges. The directed edges between nodes provide the paths that can be used by the event stream between the said nodes. Each node performs a stream transformation, i.e. accepts an input stream and emits a modified stream. There are also a special set of nodes that are dedicated to fetching the input data stream from an external source, if not creating the streams by themselves.

Hence, the developer has to write the code for each node in the Topology, and assemble the nodes with their connecting edges to finally obtain the graph. All this needs to be done using the Storm Application Programming Interface (API) [5], which is supported by multiple programming languages (Java Virtual Machine (JVM) based and non-JVM based) such as Java, Ruby, Scala, Python, etc. Commonly, various data mining and machine learning tools are deployed on Storm nodes, which generally fit the use cases mentioned above.

The Storm framework is developed under the Eclipse Public License, and is available to open use by companies and other organizations. Git and Altassian JIRA are used for version control and issue tracking, respectively, under the Apache incubator program. Some organizations that have employed Storm are Twitter, Groupon, Alibaba, The Weather Channel and FullContact.

DSLs [6] are small, simple and highly-focused specification languages developed for a clear and small problem domain. They are tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared to the use of GPLs in the same domain. By employing well-known concepts, abstractions and notations derived from the problem domain, they are easy to learn, understand and use, both by developers and domain experts.

Further, DSLs facilitate the use and reuse of domain knowledge. They are not constrained to be like programming languages. One of the main advantages is that they transcend the boundaries of programming. They tend to be more descriptive and verbose than GPLs. Hence, using DSLs, domain experts can directly contribute to the development effort by validating, modifying and even independently developing DSL programs.

978-83-60810-58-3/$25.00

Every time a Storm Topology needs to be created, hundreds of lines of Java code will have to be written by the developer. Anyone who needs to deploy their application on Storm would first have to learn how to develop Storm Topologies using general purpose languages. This could waste a lot development effort and time, that could instead be used for improving the application at hand.

For these reasons, any DSL for the Storm domain should create a simple, quick and reliable means for assembling a Storm Topology and deploying the intended application. Of course, the domain knowledge of Storm and its concepts is still necessary, and the application should be deployable within the scope of the DSL. Further, the creation of the Storm Topology in an ad-hoc manner using the DSL can be used for testing the application on this framework.

XText [7] is an open-source framework for developing programming languages and domain-specific languages. It provides a parser, abstract syntax tree generator and a Java code generator. It is a part of the Eclipse modelling project. We used XText to create Stormgen.

The rest of the paper is organized as follows: the related work in this area is given in Section 2. A domain specific meta-model for Topology development in Storm, which is used for developing the abstract syntax of Stormgen, is discussed in Section 3. Based on this meta-model, the textual concrete syntax of Stormgen is presented in Section 4. The transformations from the specifications required for Code generation are elaborated in Section 5. A case study is presented in Section 6, where Stormgen is used to develop a Storm Topology for the popular "Word Count" problem. Finally the conclusions and possible future improvements are discussed in Section 7.

II. Related Work

XText, as a tool to develop DSLs, is gaining large popularity in the research community, due to its simplicity, stability and facilities. In [8], XText was used to create a DSL called SEA_L which is used in Semantic Web enabled Multi-agent Systems. This paper follows a similar analysis performed in [8] to present the DSL developed.

Other instances where DSLs have been created using XText includes [9]. In this work, the XText framework was used to describe the implementation of an assembler editor for the development of assembly code. The editor supports specific assembler instructions. The other features provided by this editor are content monitoring, detection of repeated instructions, and prediction to assist user input. Our DSL also comes with such an editor. XText comes with a customizable user-interface component, which can be used to create an Eclipse-like IDE for the DSL being created. This way, all the useful Eclipse development features can be provided to our DSL.

In yet another research work [10] selected dependability of multi-agent system (MAS) as the domain. The key requirement in this domain is an efficient verification of a Topology model of a power system. As a result, they developed a DSL as a reliability evaluation solution offering a significant rise in the level of abstraction towards MAS utilized by the system. They made use of Eclipse Ecore, as it becomes a common denominator, in which both meta-models and abstract syntax trees are defined. Eclipse Ecore is a meta-modelling framework, part of the Eclipse Modelling Project, and we have made use of this to prepare our meta-models.

Where distributed fault-tolerant systems are concerned, as discussed in [11], Pig Latin is a DSL that is used to create and execute MapReduce jobs on Hadoop. The Pig Latin syntax has the declarative style of SQL and the low-level, procedural style of MapReduce. This language is especially useful for an expert in RDBMS systems and SQL to perform MapReduce jobs without requiring knowledge of the MapReduce programming paradigm and Hadoop. Hence, it lies at a very high level of abstraction. However, our DSL does not provide that level of abstraction, as a user cannot fully exploit the various features of Storm, if restricted only to an SQL-like interface. Instead, a user with knowledge of the features of Storm can develop a Topology that can best fit the problem at hand. Esper is the Pig Latin equivalent in Storm, i.e. it provides streaming of SQL queries on top of Storm.

Other DSLs for Storm [4] include Redstorm [12], Scala DSL, Clojure DSL, etc. Each of these DSLs require knowledge of programming languages like Scala, Ruby and Clojure. Our DSL aims to alleviate the use of GPLs and provides simple constructs close to plain English to develop the Topology. We have evaluated Stormgen against Redstorm at the end of this paper.

III. Abstract Syntax

The abstract syntax of a DSL describes the domain concepts and their relations without any consideration of their meaning. In terms of Model-driven development, a domain model or data model represents the data we want to work with. The data model is generally independent of application logic. The meta-model is used to describe the structure of the domain model. The abstract syntax is described by a meta-model. This constitutes the analysis phase of the development of the DSL.

As mentioned earlier, a Storm Topology consists of a collection of nodes that do some processing and transformation on the incoming data stream. Broadly, there are 2 types of nodes that can exist in a Storm Topology:

1) **Spouts**
   These are nodes that create an input stream of data for the Topology either by generating it randomly on the fly, or by connecting to a third party source of events through a streaming API (for example, the Twitter streaming API [13]). The Spout collects the events from the source and emits them to the rest of the Topology. It can never have a stream input to it from any other node. In essence, it is the source vertex of the Topology. A directed graph, however, can have multiple source vertices. Similarly, a Topology can have multiple Spouts of different types.
2) **Bolts**
   These are nodes in the Topology that do some computation or processing on the incoming data stream(s) and emit the data to the downstream operators. Bolts at

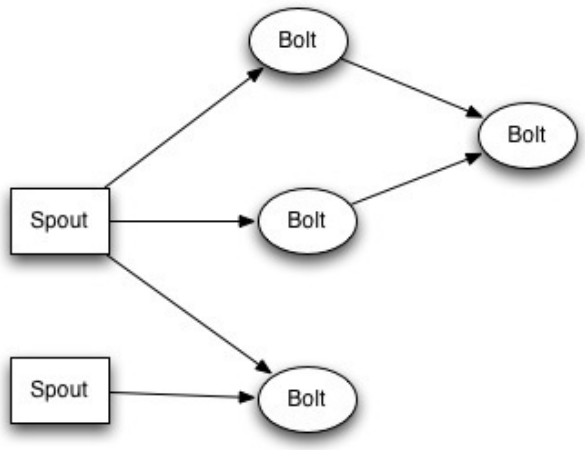

Fig. 1. Spouts and Bolts in a Storm Topology, as given in [4]

the end of the Topology do not emit data downstream. Bolts can do anything from filtering of data, execution of functions, communication with databases, database operations such as aggregations, joins, etc. They can perform either simple or complex stream processing.

A Topology can finally be assembled by adding Spouts and Bolts and the necessary edges between nodes. Fig. 1. shows a sample Topology.

Eclipse Modelling Framework (EMF) comes with two meta-models; Ecore [14] and Genmodel. The Ecore meta-model is used to store the information of the defined classes, whereas the Genmodel is used to store all the information required for code generation. EMF was chosen as the modelling platform mainly because it makes the domain model explicit, providing clear visibility. Moreover, EMF takes care of interface generation and the factory for object creation.

Ecore specifies a set of elements that could be used in the development of the meta-model. These elements are similar to some of the elements in the UML Class diagram:

1) EClass: This represents a class element. It can have zero or more attributes and references.
2) EAttribute: This represents an attribute, consisting of a name and type.
3) EReference: This represents an end-point of association between 2 EClasses.
4) EDataType: This represents the data type of an attribute. For example, *java.util.ArrayList*.

Our abstract syntax provides EClasses called Bolt, Spout and Topology. Fig. 2. shows the Ecore meta-model [14] for our model. The Storm API [5] provides interfaces to create Spouts and Bolts. For every Spout and Bolt a class has to be created and must override all the interface methods.

The aim of the final DSL is to provide very simple implementations of all the well-known characteristics of Spouts and Bolts. All the well-known features are captured in the *API* EAttribute. The Property EAttribute allows the user to define any data member (variable) and the Operation EAttribute lets the user define any function (method).

Some of the important features of the Bolt API are listed below

1) Prepare, which is used for the pre-deployment configuration of the Bolt.
2) Execute, which receives a tuple from the input stream, does some processing on the stream, and emits the result to the output collector.
3) Output Field Declarer (OPFields), which declares what logical type or key the fields in the emitted tuples assume.

Similarly, some of the important features of the Spout API are listed below

1) Open, which is used for the pre-deployment configuration of the Spout.
2) NextTuple, which connects to the source of the data stream, parses the stream into tuples, and emits them to the rest of the Topology.
3) Output Field Declarer (OPFields), which declares what logical type or key the fields in the emitted tuples assume.

Finally, the Topology assembly is done with the help of a TopologyBuilder class of the Storm API. The builder can either add Spouts or Bolts. As Bolts will have incoming edges, the adjacent upstream node needs to be specified as well. While adding a Spout to the Topology, the logical name and the instance of the class containing the source code should be provided. As any given node can be replicated multiple times during deployment, this number is provided as well while adding the Spout. While adding the Bolt, the same parameters as in the case of the Spout need to be provided. The logical name of the upstream operator needs to be defined using the "grouping" EAttribute.

IV. Textual Concrete Syntax

The textual concrete syntax of Stormgen is provided with XText [15]. XText is a language development framework to provide textual modelling languages. It can be used for creating a sophisticated Eclipse-based development environment. XText is based on Extended Backus Naur Form (EBNF) [16] rules. Hence, the design phase in the development of Stormgen constitutes the description of the EBNF rules.

As explained in the Related work section, we make use of the XText features in order to create an Eclipe-IDE user interface for Stormgen. This way, auto completion, syntax colouring, rename refactoring, bracket matching, auto edit,etc are provided for the syntax. By defining EBNF rules, the constraints discussed in the Abstract Syntax section of Stormgen's meta-model are realized. With these capabilities, the new DSL possesses both the structure and the static semantics of the Storm domain. The structure is defined by the method signatures and the semantics by the constraint code.

The rest of this section discusses the structure of the grammar used to specify Stormgen.

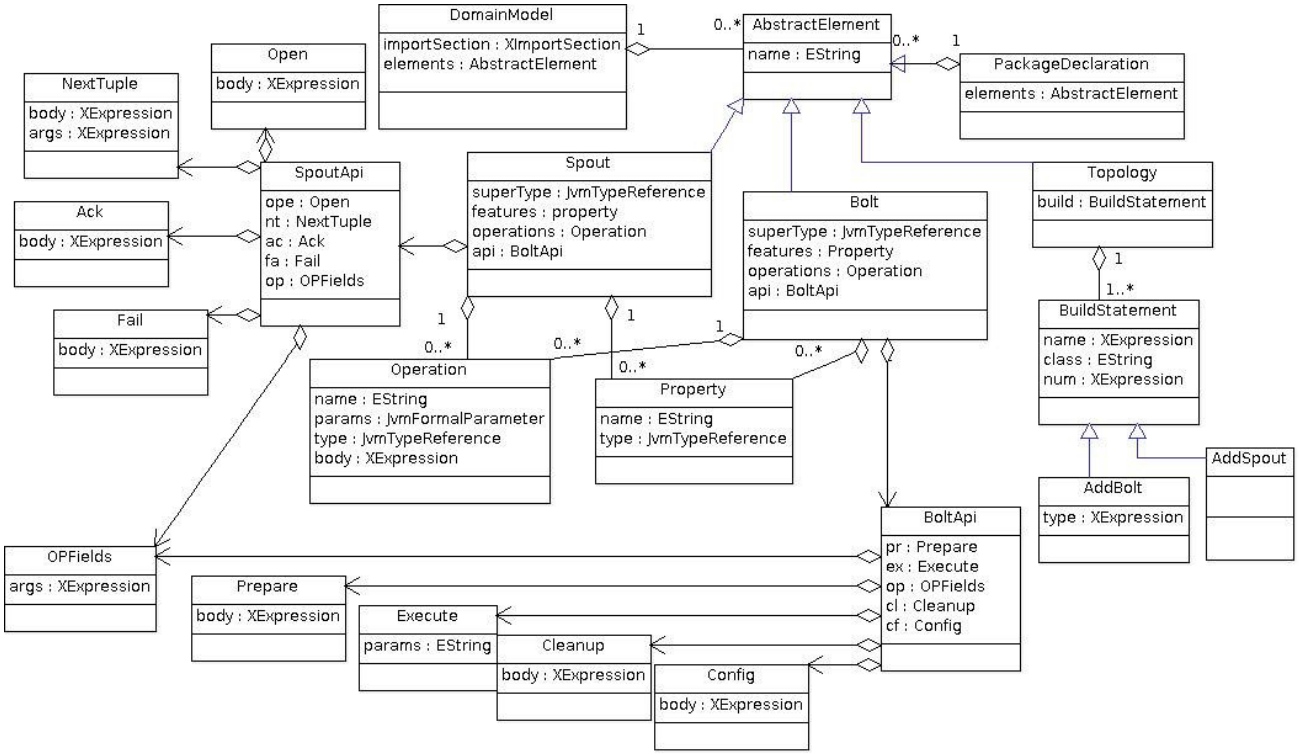

Fig. 2. Meta-model for Stormgen

Every Stormgen file should include the specifications of all the 3 major entities in the DSL, i.e. Spout, Bolt and Topology or they can contain a subset of the entire specification (For 1 or more entities).

In the beginning of the domain model all the non-Storm library related imports need to be specified. Due to the use of various JVM Elements here such as the Identifiers, Function parameters, and blocks of Java code, we made use of Xbase [17] to specify all the terminals in our grammar.

Once the imports have been specified, the rest of the file should model either a Spout, Bolt or Topology. Example 4.1 shows the implementation of this feature.

Example 4.1 (Imports and file header):

```
Domainmodel:
    importSection=XImportSection?
    elements+=AbstractElement*;
AbstractElement:
    PackageDeclaration | Bolt | Spout
    | Topology;
```

In the case of a Bolt, the meta-model constraints have to be followed. Every Bolt needs to have a well-defined name, followed by the type of Bolt primitive it is trying to implement. For example, it could implement the BaseRichBolt or IRichBolt types provided by the Storm Bolt API. The user can then optionally include basic members and methods following Java-like syntax. The rest of the code should include all of the abstract methods of the API. Example 4.2 demonstrates the Bolt syntax.

Example 4.2 (Bolt syntax):

```
Bolt:
'Bolt{'
  'Name:' name=ValidID,
  ('Type:' superType=JvmTypeReference)?,
  ('Members:' (features+=Property)+)?,
  ('Methods:' (operations+=Operation)+)?,
  ('Prepare:' pr=Prepare)?,
  'Execute:' ex=Execute,
  ('Cleanup:' cl=Cleanup)?,
  ('OutputFields:' op=OPFields)?,
  ('Config:' cf=Config)?
'}'
```

Within the Bolt API, while the implementation of the Execute feature is mandatory, the rest of the features are not. Within the Execute attribute, the emit function (outputting a tuple downstream) is called by defining the 'emit' keyword, followed by the sequence of parameters to emit.

Example 4.3 (Bolt API emit statement):

```
Execute:
'execute {'
```

```
    ('emit ('(params+=ValidID
       (',' params+=ValidID)*)')')
    '}';
```

The Spout syntax has a similar structure to the Bolt syntax, as shown in Example 4.4

Example 4.4 (Spout syntax):

```
Spout:
  'Spout{'
    'SpoutName:' name=ValidID,
    ('Type:' superType=JvmTypeReference)?,
    ('Members:' (features+=Property)+)?,
    ('Methods:' (operations+=Operation)+)?,
    ('Open:' ope=Open)?,
    'NextTuple:' nt=NextTuple,
    ('Ack:' ac=Ack)?,
    ('OutputFields:' op=OPFields)?,
    ('Fail:' fa=Fail)?,
  '}'
```

Like the Bolt, the Spout needs to be provided by name and type. The Spout could implement IRichSpout or BaseRichSpout types. The Spout can also have optional Properties and Features. Within the Spout API, the NextTuple feature needs to be included, but the rest are optional. The emit property is defined in the NextTuple function.

Finally, the grammar for building the Topology is given in Example 4.5.

Example 4.5 (Topology Syntax):

```
Topology :
    'Topology' name=ValidID '{'
     build+= BuildStatement*
    '}'
    ;
BuildStatement:
    AddSpout | AddBolt;

AddSpout:
   'addSpout{
    'SpoutName:name=XStringLiteral',
    'SpoutInstance:clas=QualifiedName',
    'Parallelism:num=XNumberLiteral
    '}';

AddBolt:
    'addBolt{
      'BoltName:name=XStringLiteral',
     'BoltInstance:clas=QualifiedName',
     'Parallelism:num=XNumberLiteral
     'Upstream:(Name:up=XStringLiteral,
      Type:(type = 'shuffle'|
      type= 'fields')+
     '}'
Config : ('default'|'custom')
```

The Topology grammar lets the user add any number of Bolts and Spouts to the Topology. Apart from specifying the primitive arguments of both Bolt and Spout, the logical name of the adjacent upstream node to every Bolt is provided. Additional configuration information needs to be applied to the Topology, such as the number of Worker processes per node, code concerning the submission of the Topology to the Storm daemon processes, etc. Using the default option with the 'Config' field (as shown in Example 4.5), the default configuration is generated for ad-hoc Topologies.

XText can generate EBNF rules from a given meta-model but we prefer to define EBNF rules manually to supply some preferred syntactical restrictions and constraints such as defining relations in a specific order (XText cannot extract the order from the meta-model because the meta-model does not have such an attribute by itself), defining at least one or more than one relation, etc.

V. Code Generation

It is not sufficient to complete the DSL definition only by specifying the notations and their representations. The complete definition requires that the semantics of the DSL's concepts are mapped to Java constructs. The mapping is provided through model to code transformations where the final executable software code for exact Storm Topology creation is obtained. Code generation for the instance models are provided by the Xtext Framework [15].

Many of the existing model driven engineering approaches accomplish code generation by writing strings to text files. XTend is a flexible and expressive dialect of Java, which compiles into readable Java compatible source code. XTend prepares a compiled output of Java source code that is similar to the equivalent hand-written code, both in structure and performance. Unlike other JVM languages XTend has no interoperability issues with Java.

Like XPand [19], XTend [18] is a template engine, which allows creating textual output using EMF models. XTend requires an EMF meta-model and one or more templates to translate the model into text. Once the requirements are provided and an EMF model [20] is defined, the code generator can be deployed. XTend traverses the abstract tree created by XText and generates the code along the way.

However, compared to XPand, XTend has the following additional benefits as explained in [21]

1) XTend is fast because XTend code is translated to Java code without adding overheads or dependencies at runtime.
2) XTend is debuggable as XTend code is translated to Java code. Hence, advanced Java debugging tools can be used. Additionally the Eclipse debugger provides the option to debug either the XTend source code or the generated Java code.
3) Better Integrated Development Environment (IDE) support.
4) As templates in XTend are expressions which yield some value, multiple templates can be composed and

the results can be passed around and processed.
5) Better extendibility.

Hence, the code generation for Stormgen is done using XTend.

Every EClass in the meta-model has a corresponding definition in the grammar. The grammar rules have to then be mapped during code generation into various components of the target generated program. As mentioned in the previous section, the 3 main components that have to be included in the Domain model are Spout, Bolt and Topology. The code generator has an "inferrer" defined for each of these 3 components.

The IRichBolt interface is used to implement the Bolt in the model. So during generation, the corresponding statement needs to be included, along with the required import. This is followed by the generation of the constructors.

Example 5.1 (Generating constructors):

```
members += element.toConstructor[
 for (feature : element.features)
 {
   parameters += toParameter
   (feature.name, feature.type)
 }
 body = [
   for(feature:element.features)
   {
     append ("this."+feature.name+
     "="+feature.name+";")
   }
  ]
]
```

Properties have to be translated into corresponding Java class member variables and Operations have to be translated to corresponding Java class functions

Finally the Bolt API attributes are translated into Java functions, which override the functions in the IRichBolt interface. All the attributes have to be appended with correct function call and arguments, having imported the corresponding arguments from the Storm library.

The Spout code is generated in a similar manner to that of the Bolt. The IRichSpout interface is implemented to provide all the necessary functions to override. The constructor of the Spout class is generated. All the instances of the Property EClass are translated into Java class members and all the instances of the Operation EClass are translated into Java class functions. Finally all the overridden methods are defined with the required prototype and the fully qualified arguments.

Topology has different grammar constructs from Bolt and Spout. Hence the code generation is slightly different. In order to build a Topology, a TopologyBuilder instance from the Storm API needs to be created. The Topology Builder instance can then be used to add Spouts and Bolts, constituting the assembly process. The adjacent upstream nodes for the Bolts are also specified. This builder instance is implicitly created, and the DSL provides EClasses like AddSpout and AddBolt to add the Spouts and Bolts to this implicit instance.

Over and above this, there is a lot of code that is used to configure the Topology for a local mode execution. All this is hard-coded into the generator, and generated for every possible Topology created using this DSL.

VI. Case Study: Word count

Word count [22] is a very popular problem that is especially used to understand the functioning of various distributed, fault-tolerant data processing systems, including Storm.

In word count, a very large document is provided as input to the framework and the expected output is a report of the frequency of every single word in the document. This is a fairly simple application to develop, and we will be using this example to underline the simplicity and power of Stormgen.

First we need to analyse the problem and understand how it can be modelled into the Storm domain. In essence, the Storm domain involves the creation of Topologies using Spouts and Bolts. So we now need to understand what components are required to develop a solution to this problem.

We need a real time data stream for processing. So we decided to simulate the same by constructing a RandomSentenceGenerator Spout which has a list of sentences, and randomly emits a sentence every time the 'NextTuple' routine is called by the framework.

Now that our data stream consists of Tuples, each being a randomly selected sentence, these sentences need to be split (removal of whitespaces) into words. To carry out this operation we construct a SplitSentence Bolt which accepts an input tuple containing a sentence, splits the sentence into words and emits each word downstream. This operation of splitting the sentence is written directly in Java.

The SplitSentence Bolt subscribes directly to the RandomSentenceGenerator Spout. As the process of splitting sentences doesn't need to be done at any specific instance of the SplitSentence Bolt, normal shuffle grouping is used to group the tuples to the instances of this Bolt.

Now that the SplitSentenceBolt generates a stream of words, these words need to be counted. For this, we construct the WordCounter Bolt, which accepts a word and increments the word's count, which is stored in a local Hash Map. This Bolt is the final Bolt in the Topology and does not emit any data subsequently. This Bolt subscribes to the SplitSentence Bolt. However, there should be a constraint enforced here. If different instances of the *same* word go to different instances of the WordCounter Bolt, then obviously the total count reported by each Bolt will be wrong. So it is important to ensure that *all* instances of the same word should go to the same instance of the Bolt to ensure a correct count. This can be achieved by using the Field stream grouping in Storm.

For convenience, in the DSL for our domain model, we have created 4 files:

1) RandomSentenceSpout.strgen
2) SplitSentenceBolt.strgen
3) WordCounterBolt.strgen

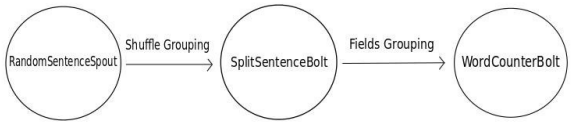

Fig. 3. The word count Topology

4) WordCountTopology.strgen

Note that all the entities could also be defined in a single Stromgen file.

Fig. 3. shows how the Topology looks like. In the WordCountTopology.strgen, we construct the Topology with the given Spouts and Bolts. The contents of the WordCountTopology.strgen file are shown in Example 6.1. The generated Java code for the Topology is given in Example 6.2. The examples of the other 3 code files are not included due to space constraints. Only if *all 4* of the source files are specified correctly will the correct Java application code will be generated.

Example 6.1 (Word count Topology Stormgen model):

```
package org.Topologies {
  Topology WordCountTopology {
    addSpout{
      SpoutName:"Spout",
      SpoutInstance:
        Spouts.RandomSentenceSpout,
      Parallelism:10
    }

    addBolt{
      BoltName: "split",
      BoltInstance:
        Bolts.SplitSentenceBolt,
      Parallelism: 5,
      Upstream: ("Spout",shuffle)
    }

    addBolt{
      BoltName: "count",
      BoltInstance:
        Bolts.WordCounterBolt,
      Parallelism: 5,
      Upstream: ("split",fields)
    }

    Config: default
}
```

Example 6.2 (Word count Topology generated java code):

```
package org.Topologies;

@SuppressWarnings("all")
public class WordCountTopology {
 public static void main(String[] args){
  backtype.Storm.Topology.TopologyBuilder
   builder = new backtype.Storm.Topology.
        TopologyBuilder();

  Config conf = new Config();
  conf.setDebug(true);
  builder.setSpout("Spout",
        new Spouts.RandomSentenceSpout(),
        10);
  builder.setBolt("split",
        new Bolts.SplitSentenceBolt(),
        5).
        shuffleGrouping("Spout");
  builder.setBolt("count",
        new Bolts.WordCounterBolt(),
        5).
        fieldsGrouping("split");

  if (args != null && args.length > 0){
    conf.setNumWorkers(3);
    StormSubmitter.submitTopology(args[0],
      conf, builder.createTopology());
  }
  else {
    conf.setMaxTaskParallelism(3);
    LocalCluster cluster =
       new LocalCluster();
    cluster.submitTopology("word-count",
       conf, builder.createTopology());
    Thread.sleep(10000);
    cluster.shutdown();
  }
 }
}
```

As it is evident from this case study, Stormgen greatly simplifies the assembly of a Topology. With the simple, verbose syntax of Stormgen we could generate the equivalent Java code for the whole Topology. Further, Stormgen is abstracted to capture just the domain concepts. Hence, only the Storm-API related specifications need to be specified using the DSL. The rest of the Java code can be generated once this specification is provided.

VII. Evaluation

As mentioned in the Related Work, there are several DSLs available for Storm, such as Redstorm, Scala DSL, Clojure DSL, etc. They provide a similar level of abstraction as Stormgen. However, the user is expected to know Scala, Ruby or Clojure, which are GPLs, but provide a simpler syntax than Java. Stormgen provides a simple, intuitive syntax for the development of the core components of the Topology. This syntax is not based on any GPL. Additionally, Stormgen permits the programmer to introduce Java code into the DSL whenever any specific functionality has to be incorporated.

The primary use case that Stormgen is developed for, is the deployment of third-party applications on Storm. As men-

tioned earlier, Storm is normally used for larger data mining or machine learning applications that need real-time, fault-tolerant and distributed data processing. Further, additional development effort has to be expended on understanding how to develop Storm topologies using Java. Instead, once the user has learnt the domain concepts, Stormgen can be directly used to create the necessary components of the topology and payload the various units of the third-party application on these components.

Redstorm [12] is a DSL for Storm developed using JRuby. All the components have to be written in Ruby. While JRuby does permit the direct use of Java code, it requires special configuration when access to non-bundled Java libraries is required. Non-bundled Java libraries will be required when a third-party application is being deployed on Storm. Stormgen solves this problem by permitting ordinary Java imports to be mentioned in the DSL file, which is added in verbatim to the generated code. This alleviates the need for any special configuration and the user can directly focus on the Topology development.

VIII. Conclusion and Future Work

In this paper we have presented our DSL, Stormgen, describing the motivation behind developing it, and further explaining in detail what went on in the whole software development process. In the analysis phase, the Abstract syntax was specified with the help of domain-specific meta-models. These meta-model concepts were then mapped to the concrete textual Syntax during the design phase. The next phase involved mapping the textual syntax to code by the construction of the code generator. Finally, we tested Stormgen with the popular WordCount application. All this was done entirely using tools such as Ecore, XText and Xtend, provided by EMF.

As explained throughout the paper, Stormgen allows the user to develop Topologies for Storm by simply applying domain knowledge and concepts to the domain model. Stormgen also allows the user to import external Java code, so that any other Java applications can be deployed seamlessly into Storm.

In future, Stormgen can be upgraded to incorporate support for additional Bolt/Spout interfaces apart from the BaseRich and IRich Bolt/Spout. While the existing implementation is sufficient to cover most use cases, providing the support for the other interfaces would cover each and every feature provided by Storm. Finally, using the Eclipse Graphical Modelling Framework (GMF) [23], a graphical DSL can be developed to supplement the textual DSL. The graphical DSL would provide a simple graphical user interface (GUI) to draw the graph of the Topology and a convenient technique to configure each node and stream (edge). This would provide a more intuitive visualization to the Storm Topology development.

References

[1] Dean, Jeffrey, and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”, *Communications of the ACM* 51.1 (2008): 107-113. http://dx.doi.org/10.1145/1327452.1327492

[2] White, Tom. “Hadoop: The Definitive Guide: The Definitive Guide.” O’Reilly Media, 2009.

[3] Marz, Nathan. "Storm-distributed and fault-tolerant realtime computation." *Open Source Conference (OSCON)*. 2012.

[4] Marz, Nathan. "Storm wiki." https://github.com/nathanmarz/Storm/wiki (2012).

[5] Marz, Nathan. "Storm Javadoc." http://nathanmarz.github.io/storm/doc-0.8.1 (2012).

[6] Fowler, Martin. Domain-specific languages. Pearson Education, 2010.

[7] Eysholdt, Moritz, and Heiko Behrens. "Xtext: implement your language faster than the quick and dirty way." Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. ACM, 2010. http://dx.doi.org/10.1145/1869542.1869625

[8] Demirkol, Sebla, et al. "A DSL for the development of software agents working within a semantic web environment." Computer Science and Information Systems 10.4 (2013): 1525-1556. doi:10.2298/CSIS121105044D

[9] Kartalija, Sasa, et al. "One solution of implementation assembler editor on the Java platform using the XText framework." Telecommunications Forum (TELFOR), 2012 20th. IEEE, 2012. http://dx.doi.org/10.1109/TELFOR.2012.6419547

[10] Kowalski, Marcin, and Kazimierz Wilkosz. "A Domain Specific Language in Dependability Analysis." Dependability of Computer Systems, 2009. DepCos-RELCOMEX’09. Fourth International Conference on. IEEE, 2009. http://dx.doi.org/10.1109/DepCoS-RELCOMEX.2009.14

[11] Gates, Alan F., et al. "Building a high-level dataflow system on top of MapReduce: the Pig experience." Proceedings of the VLDB Endowment 2.2 (2009): 1414-1425. http://dx.doi.org/10.14778/1687553.1687568

[12] Superenant, Colin. "Red Storm" https://github.com/colinsurprenant/redstorm (2012).

[13] Benhardus, James, and Jugal Kalita. "Streaming trend detection in twitter." International Journal of Web Based Communities 9.1 (2013): 122-139. doi:10.1504/IJWBC.2013.051298

[14] Stephan, Matthew, and Michal Antkiewicz. "Ecore. fmp: A tool for editing and instantiating class models as feature models." University of Waterloo, Tech. Rep 8 (2008).

[15] Behrens, Heiko, et al. "XText user guide." DostupnÃľ z WWW: http://www.eclipse.org/Xtext/documentation/1_0_1/xtext.pdf (2008).

[16] Garshol, Lars Marius. "BNF and EBNF: What are they and how do they work." acedida pela ÃŽltima vez em 16 (2005).

[17] Efftinge, Sven, et al. "Xbase: implementing domain-specific languages for Java." ACM SIGPLAN Notices. Vol. 48. No. 3. ACM, 2012. http://dx.doi.org/10.1145/2480361.2371419

[18] Bettini, Lorenzo. Implementing Domain-Specific Languages with XText and Xtend. Packt Publishing Ltd, 2013.

[19] Klatt, Benjamin. "Xpand: A closer look at the model2text transformation language." Language 10.16 (2007): 2008.

[20] Budinsky, Frank, ed. Eclipse modeling framework: a developer’s guide. Addison-Wesley Professional, 2004.

[21] Five good reasons to port your code generator to Xtend - http://blog.efftinge.de/2013/06/five-good-reasons-to-port-your-code.html

[22] Dean, Jeffrey, and Sanjay Ghemawat. "Distributed programming with Mapreduce." Beautiful Code. Sebastopol: O’Reilly Media, Inc 384 (2007).

[23] Eclipse Consortium. "Eclipse Graphical Modeling Framework (GMF)(2007)."