VIPER: a web application for rapid expert review of variant calls
Supplementary Material

Marius Wöste and Martin Dugas

December 22, 2017

1 Architecture

Figure 1 gives a rough overview over the architecture of the Variant InsPector and Expert Rating tool (VIPER). The application consists of a server managing variant data and visualization, and a web browser interface for user interaction.

1.1 Server

VIPER's main component is the server part. The server component is implemented in Java 1.8 to enable usage on every major operating system (OS). It is responsible for importing and exporting variant data, which is explained in detail in subsection 2.1 and subsection 2.2. Importing variant data includes optional grouping of similar variant calls (see subsection 2.3). The server component stores both ungrouped and grouped variant tables for faster access. After successfully importing data, the variants can be locally accessed via HTTP.

The HTTP component is implemented using the Java Spark Framework (http://sparkjava.com/). Variant calls cannot only be accessed via HTTP, but also annotated with decisions and filtered based on metadata from the input file. Subsection 2.4 contains a detailed explanation about the filtering process. Variant visualization may also be scheduled and accessed through the HTTP component.

1.1.1 Variant visualization

Variant calling is still a challenging task as there are many sources for potential errors such as machine error (Hwang *et al.*, 2015 ; Sandmann *et al.*, 2017). Visualizing genomic regions eases distinguishing false from true positive variant calls. We choose the Integrative Genomics Viewer (IGV, Robinson *et al.*, 2011 ; Thorvaldsdóttir *et al.*, 2013) as a visualization tool because of its many features and its ability to be controlled via HTTP requests.

When starting the VIPER server, a thread is started that is responsible for managing an IGV instance. The IGV instance is started in a headless environment if using Linux and Xvfb. The visualization thread is then used to send visualization commands to the IGV instance via HTTP. For example, the following IGV commands are generated for a deletion

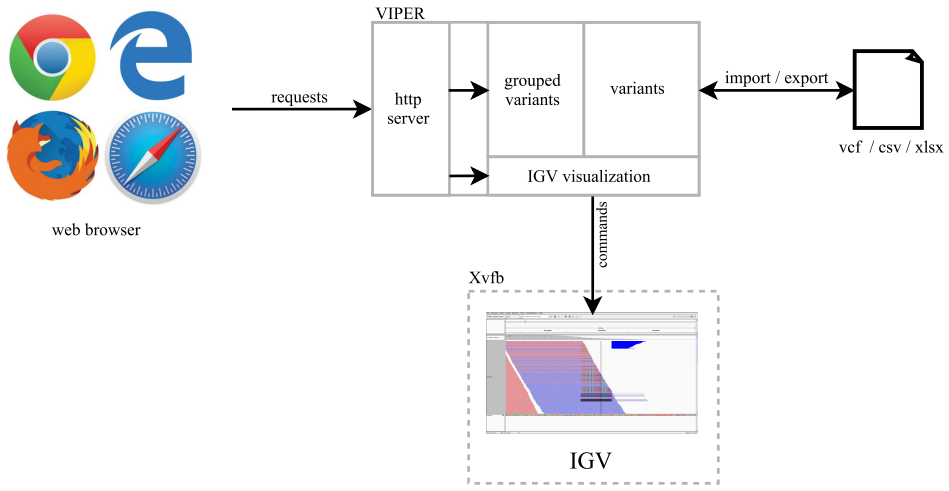

Figure 1: VIPER architecture overview

on chromosome 4 starting at position 1,000,000 and ending at position 2,000,000 found in a file called `sample.bam`:

```
new
load sample.bam
collapse
goto 4:999975-1000025
snapshot sample-4-1000000.png
new
load sample.bam
collapse
goto 4:1999975-2000025
snapshot sample-4-2000000.png
```

This results in IGV creating two image files containing the deletion's breakpoints.

The breakpoint images are computed on demand as the user iterates through the variant calls instead of precomputing all images on server startup. This drastically reduces the time required for server startup and minimizes storage requirements. However, visualizing only the currently inspected variant on demand would lead to waiting times for the user. As users are inspecting variant calls in a linear fashion, we can exploit the locality of this process and precompute a fixed number of variants that the user will inspect next.

Additionally, IGV visualization is configurable through VIPER's web interface. Users may set coloring, grouping, soft-clipping and downsampling options dynamically during the variant inspection process.

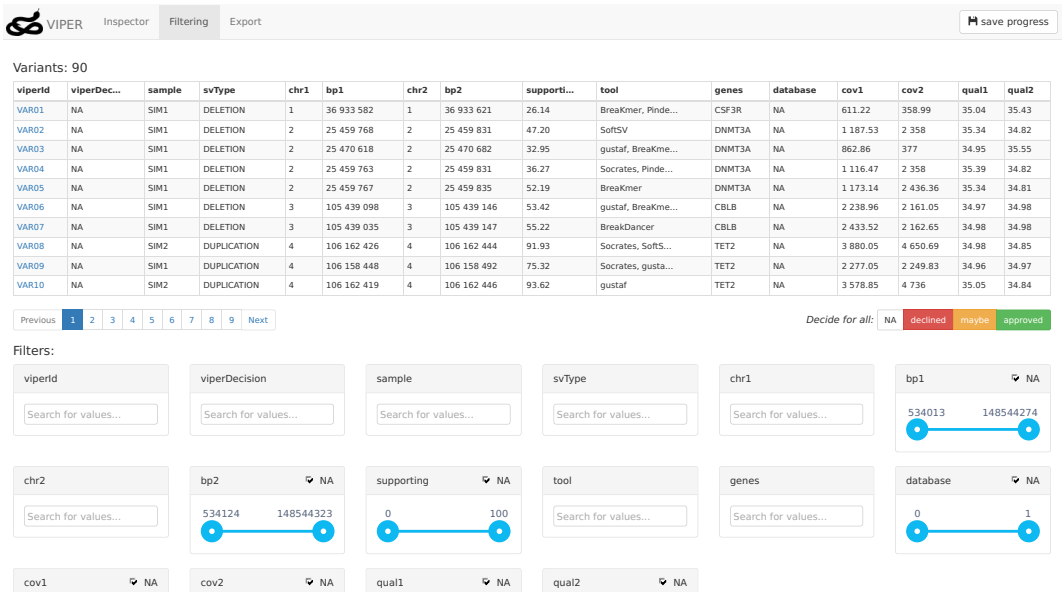

Figure 2: VIPER's *Filtering* page. Allows inspection of variants in a table as well as applying filters based on the table's columns.

1.2 Web browser interface

1.2.1 Design

VIPER's main goal is to minimize time required for manual variant inspection and decision making. Classifying variant calls into true and false positives is usually based on metrics such as coverage and quality, and visualization with tools such as the IGV. To minimize time required for a single decision, both variant metadata and breakpoint visualization are presented on a single *Inspector* page. Thus, making a decision reduces to clicking a single button.

However, iterating through variants one at a time may be too time-consuming for larger variant datasets. As a result, we added a *Filtering* page that displays all variants in a single table. This page enables users to apply filters based on the table's columns and set a decision for all variants matching the current filtering criteria. Figure 2 shows the *Filtering* page.

1.2.2 Implementation

We chose to implement VIPER's frontend as a web interface because web browsers are available for all major OSs, and there are a plethora of frameworks and libraries that ease developing frontend functionality.

The web interface is implemented as a single page application using AngularJS 1.6.5 (https://angularjs.org/) and Bootstrap 3.3.7 (https://getbootstrap.com/docs/3.3/). We chose AngularJS because it enables HTML templates to be rendered on the client-side exclusively, thus seperating frontend and backend. The browser client can then use VIPER by making calls to the HTTP API only, making the web interface highly dynamic. This also

enables a potential usage of different frontends in the future.

2 Handling variant data

VIPER makes certain assumptions about variants in order to simplify dealing with variant data. For VIPER, a variant is considered to consist of a sample name, a variant type and two loci. The sample name denotes a `BAM` file name that the variant was found in. The variant type is an arbitrary string and is used for the optional grouping denoted in subsection 2.3. The two loci describe the variant's breakpoints. A typical variant might look like this:

```
{
  // name of bam file
  sample: "PATIENT32-WGS",

  // type of variant
  type: "DELETION",

  // first breakpoint
  chr1: "4",
  bp1: 4000000,

  // second breakpoint
  chr2: "4",
  bp2: 5000000
}
```

However, using only positional information is usually not enough to make a meaningful decision about a variant's truth. As a result, we allow arbitrary metadata to be added to the variants to show useful information and optionally enable filtering. We use a table-like structure to store variant data. The columns *sample*, *type*, *chr1*, *bp1*, *chr2* and *bp2* are mandatory as described before. Columns *cov1*, *cov2* and *genes* may be added, describing average coverage in breakpoint regions and overlapping genes respectively. An example variant table with metadata is shown in Table 1.

sample	type	chr1	bp1	chr2	bp2	cov1	cov2	genes
PATIENT32	DELETION	4	4000000	4	5000000	37.50	22.50	*NA*
PATIENT76	DELETION	2	25464548	2	223731457	2381.10	55.93	DNMT3A,ACSL3
PATIENT123	TRANSLOCATION	4	106193716	X	129171548	34.10	23.80	TET2,BCORL1
PATIENT147	SNV	X	10000	X	10000	1.5	2.3	*NA*

Table 1: Example variant table containing mandatory columns and metadata columns describing coverage and overlapped genes.

While these assumptions are adequate for variants with exactly two breakpoints, e.g. deletions or inversions, complex rearrangements with more than two breakpoints can not

be expressed in a canonical manner. Also, by definition single nucleotide variations (SNVs) span a single locus only. This problem can easily be solved by assinging an SNV's single locus to both variant breakpoints. The described variant definition has been sufficient for our analyses, but may be extended in the future if necessary because of its inherent limitations.

As each column might contain multiple values per variant, e.g. genes and optionally grouped values as explained in subsection 2.3, every column is assigned one of the following types:

- `STRING`
- `STRING_COLLECTION`
- `NUMERIC`
- `NUMERIC_COLLECTION`

This distinction of types eases appropriately displaying and filtering variant data. The `STRING` type represents arbitrary non-empty character sequences. The `NUMERIC` type represents 64-bit floating point numbers. It is noteworthy that a `NUMERIC` value may be empty (*NA*). To simplify dealing with empty `STRING` values we represent these using the character sequence "`NA`". The `COLLECTION` types represent multisets containing values with `STRING` or `NUMERIC` type.

2.1 Import

As variants are stored in a table-like structure, it is straightforward to represent the data as a CSV file with the first line containing the column names. Both `STRING` and `NUMERIC` types are represented by their canonical character sequence representations. The `COLLECTION` types are encoded by multiple values seperated by an arbitrary configurable character.

Reading a variant table from a CSV file consists of loading the file's content, determining column types and parsing values. Loading a CSV file is performed by a third party library. The column types are determined comparing each column's values against a floating point regex.

If all column values match a floating point number or a character seperated list of floating point numbers the type is set to `NUMERICAL` or `NUMERICAL_COLLECTION` respectively. Should any column value not be recognized as a floating point number representation, the column's type is set to `STRING` or `STRING_COLLECTION`, depending on any value containing a delimiter character. After determining column types all values are parsed and put into data structures using the Java Standard libraries.

VIPER also supports reading VCF files since it is a wide-spread variant description standard (https://samtools.github.io/hts-specs/VCFv4.2.pdf). However, since VCF files describe variants in a more general way than our definition, complex rearrangements exceeding two breakpoints can not trivially be imported.

We use samtools' HTSJDK (https://samtools.github.io/htsjdk/) to iterate through the variants defined in the VCF file. Breakpoint locations are accessed using HTSJDK's corresponding methods. Additional columns are added for mandatory fields (e.g. REF), genotype

and info fields that are defined in the header of the VCF file. The header also contains information about the data type of every field, enabling direct conversion to VIPER's data types.

However, the VCF file specification allows inclusion of calls concordant to reference. These may optionally be excluded to only investigate non-reference calls.

2.2 Export

VIPER currently supports exporting the variant table to a CSV file as well as an XLSX file. This process is straightforward as the variants are already stored as a table in-memory. Third party libraries yield the features required for writing the variant table.

2.3 Grouping

In datasets with many samples, it may be of interest to associate a variant with multiple samples, e.g. normal-tumor pairs. VIPER allows optionally grouping together variants that expose similar characteristics.

As VIPER assumes every variant to include two loci, we can define a similarity relation as follows: Let v_1 and v_2 be variants with a variant type and two breakpoint loci each. We call v_1 and v_2 similar if they share the same type, the loci lie on the same chromosomes and the breakpoint positions are at most $d \in \mathbb{N}$ bp apart, where d is a configurable parameter. However, this similarity is not enough to be used as a grouping criterion as it is no transitive relation. Whenever there are variants v_1, v_2 and v_3 with v_1 similar to v_2 and v_2 similar to v_3, we consider v_1 similar to v_3 regardless of the breakpoint distances. This ensures every variant is assigned exactly one group after the grouping process. Algorithm 1 shows pseudocode for splitting variants into similar groups.

Input: Set of variants I sharing the same type and chromosomes, $d \in \mathbb{N}$
Output: Set of variant sets O

```
O ← ∅
L ← I
while L ≠ ∅ do
    G ← {any single v ∈ L}
    while G changed do
        G ← G ∪ {v′ | v ∈ G, v′ ∈ L, |v.bp1 − v′.bp1| ≤ d ∧ |v.bp2 − v′.bp2| ≤ d}
    end while
    L ← L − G
    O ← O ∪ {G}
end while
return O
```

Algorithm 1: Grouping variant calls

A new variant table is then generated based on the grouping. Columns with `NUMERIC` values keep its type using the median of values in each group as new column values. All other column types are converted to their equivalent `COLLECTION` types by computing the unique

set of combined values for each group. After the grouping process the columns *viperId* and *viperDecision* are added to enable easy identification of variants and setting decisions for each variant.

However, the grouping process requires the resolution of variants to be similar for all variant calls within the dataset since it uses breakpoint distances as a similarity metric. For example, grouping would not be applicable for a dataset containing both SNV calls as well as CNV calls that have been detected using coverage information. The user must choose the maximum distance d appropriately or disable grouping as a result.

2.4 Filtering

To enable quick iteration through all variants, it is helpful to be able to inspect only subsets of the whole dataset matching certain criteria such as very low coverage. VIPER enables this filtering by providing a single filter for each column of the variant table. For columns with `NUMERIC` values a simple interval check is performed. The bounds of such intervals as well as defining if *NA* values are allowed is configurable by the user. For each column with `STRING` values a set of allowed values can be configured by the user. A value passes this filter if its value is contained in the set. However, for usability and performance reasons a variant always passes the filter if the set of allowed values is empty. `COLLECTION` filters work like their single value conterparts, with a collection passing whenever any value of the collection passes a single value filter. A variant passes the filtering stage if all its column values pass the respective filters.

While this filtering is very basic, the column *viperDecision* can be selected as a filtering criterion as well, enabling users to use previous decisions for stateful filtering.

3 Availability

As both VIPER server and IGV are implemented in Java, running the VIPER server only requires a Java 1.8 environment. Because the web interface is implemented using AngularJS, it is heavily reliant on the browser's Javascript engine. A modern web browser is required as a result. VIPER has been succesfully tested on Ubuntu 16.04, Debian 8, Windows 7 and OS X using Firefox 55, Chrome 59, Internet Explorer 11 and Safari 10.1 respectively. It was achieving adequate performance on a desktop machine with 8 GB RAM with ∼130,000 calls.

4 Application to exploratory analyses

This section briefly explains the application of VIPER to two dataset analyses.

4.1 MDS SV analysis

The first dataset VIPER was applied to consists of 117 NGS samples from 111 patients diagnosed with MDS (6 patients sampled twice). DNA extracted from the patient's blood

was sequenced using Illumina's NextSeq 550. A paired-end amplicon-based approach was chosen to sample exons of 15 genes with high coverage.

The target region is ∼125 kb in length and a high average coverage of $3,675\times$ was achieved. Reads have been aligned using `bwa mem` (v0.7.8) with `-M` option.

The primary analysis for this dataset consisted of indel and SNV analysis that yielded many mutations with VAFs $< 20\%$. We assume potential SVs to possibly have low VAFs as well as a result. As an exploratory follow-up analysis we applied the following SV calling tools to the dataset:

- Pindel (v0.2.5b9)
- Socrates (v1.13.1)
- SeekSV (v1.2.1)
- Sprites (v0.3.0)
- BreaKmer (v0.0.7)
- gustaf (v1.0.0)
- Delly2 (v0.7.6)
- BreakDancer (v1.4.5)
- SoftSV (v1.4.2)
- CREST (v1.0.0)

All tools have been called with their respective default parameters. The tools called between 62 and 3460 raw variants each, with 18803 variants called in total. Since SV calling tools are usually developed for WGS we expect many false positives because of the much higher coverage in our dataset. Simple filtering strategies such as filtering by quality or coverage did not significantly reduce the number of calls. However, manual inspection of several calls using IGV revealed that in many cases having a quick glance at the breakpoint sites allowed identification of false positives, for example deletion calls in long homopolymers.

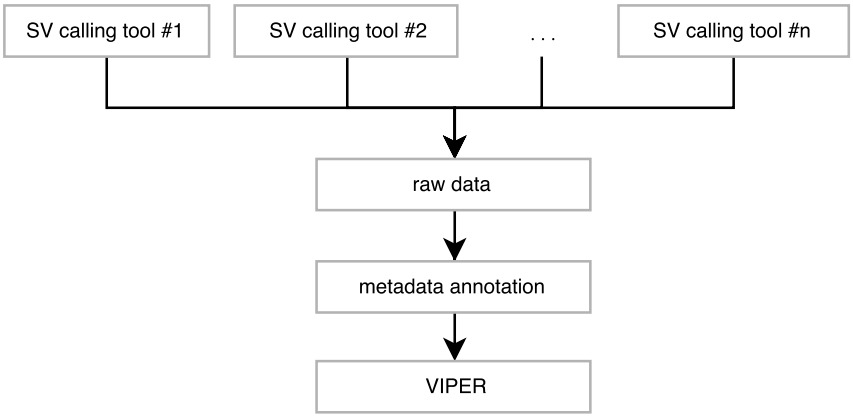

Figure 3: VIPER's application in MDS SV analysis pipeline.

This leads to the simple pipeline illustrated in figure 4.1. The first step consists of performing all tool calls and collecting all raw calls. Metadata annotation including coverage, quality and genes affected is then performed using `R` packages `biomaRt` and `Rsamtools`. The annotated calls are then fed to VIPER for decision making by the analyst.

Since all mentioned tools provide single-nucleotide resolution by using split-read information, applying VIPER enabled classification into true and false positive candidates by inspecting soft-clipping and alignment information at breakpoint sites. Only 389 calls were identified as candidates for actual mutations. Some of these candidate calls have also been

observed during the previous indel and SNV analysis and were confirmed to be true positives by further experimental validation (using Sanger sequencing).

4.2 mtDNA SNV/indel analysis

The second dataset where VIPER was used consists of 491 control-tumour sample pairs from patients with medulloblastomata. Using WGS yielded high-coverage mtDNA ($4,521\times$) data for SNV and indel analysis. All data is aligned using `bwa mem` (v0.8.7, `-T` set to 0). Variants are called using the following 8 tools:

- GATK HaplotypeCaller (v3.3.0)
- Platypus (v0.8.1)
- VarScan (v2.3.9)
- LoFreq (v2.1.2)
- FreeBayes (v1.0.2-6)
- SNVer (v0.5.3)
- SAMtools (v1.3)
- VarDict (git commit #de269ed961)

Several basic characteristics are then used to perform filtering. Variant calls must meet the following criteria to be included in further analysis:

- Number of reads with alternate allele ≥ 20
- Coverage $\geq 50\times$ at variant locus
- Variant allele frequency ≥ 0.01
- Average base quality ≥ 15 at variant locus
- Difference between mean base quality reference and mean base quality alternative allele < 7

All calls meeting these requirements are then compared to the databases 1000 Genomes, COSMIC and dbSNP (ESP6500, ExAC and ClinVar provide no data for mtDNA). The tools mtSNP, Phylotree, mtDB and MitoMap are also used to predict classification into polymorphisms and pathogene mutations.

Variants are categorized into polymorphisms, artefacts and mutations by manual inspection of the collected information. VIPER was then applied to manually inspect the categorized 11968 variant breakpoint sites to increase categorization confidence.

References

Hwang, S., Kim, E., Lee, I., and Marcotte, E. (2015). Systematic comparison of variant calling pipelines using gold standard personal exome variants. *Scientific Reports*, **5**, 17875.

Robinson, J. T., Thorvaldsdóttir, H., Winckler, W., Guttman, M., Lander, E. S., Getz, G., and Mesirov, J. P. (2011). Integrative genomics viewer. *Nature biotechnology*, **29**(1), 24–26.

Sandmann, S., de Graaf, A., Karimi, M., van der Reijden, B., Hellström-Lindberg, E., Jansen, J., and Dugas, M. (2017). Evaluating variant calling tools for non-matched next-generation sequencing data. *Scientific Reports*, **7**, 43169.

Thorvaldsdóttir, H., Robinson, J. T., and Mesirov, J. P. (2013). Integrative genomics viewer (IGV): high-performance genomics data visualization and exploration. *Briefings in Bioinformatics*, **14**(2), 178–192.