Received 4 August 2022, accepted 23 August 2022, date of publication 29 August 2022, date of current version 14 September 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3202543

RESEARCH ARTICLE

VMetaFlow: A Meta-Framework for Integrating Visualizations in Coordinated View Applications

NICUSOR COSMIN-TOADER[1], FERNANDO TRINCADO-ALONSO[1], LUIS PASTOR[1,2], AND MARCOS GARCIA-LORENZO[1,2]

[1]VG-Laboratory, Universidad Rey Juan Carlos, 28933 Móstoles, Spain
[2]Research Center for Computational Simulation, 28660 Boadilla del Monte, Spain

Corresponding author: Marcos Garcia-Lorenzo (marcos.garcia@urjc.es)

This work was supported in part by the Spanish Ministry of Economy and Competitiveness through the VIsualización ANalítica Aplicada (VIANA) and horizontal MEthods for multimodal and complex DAta analysis and VIsualization (MEDAVI) under Grant TIN2017-83132-C2-1-R and Grant PID2020-113013RB-C21 and in part by the European Union's Horizon 2020 Framework Program for Research and Innovation through Human Brain Proyect (HBP) under Grant SGA2-785907 and Grant SGA3-945539, The work of Nicusor Cosmin-Toader was supported by the Spanish Ministry of Science and Innovation under Grant FPU18/05304.

ABSTRACT The analysis and exploration of complex data sets are common problems in many areas, including scientific and business domains. This need has led to substantial development of the data visualization field. In this paper, we present VMetaFlow, a graphical meta-framework to design interactive and coordinated views applications for data visualization. Our meta-framework is based on data flow diagrams since they have proved their value in simplifying the design of data visualizations. VMetaFlow operates as an abstraction layer that encapsulates and interconnects visualization frameworks in a web-based environment, providing them with interoperability mechanisms. The only requirement is that the visualization framework must be accessible through a JavaScript API. We propose a novel data flow model that allows users to define both interactions between multiple data views and how the data flows between visualization and data processing modules. In contrast with previous data-flow-based frameworks for visualization, we separate the view interactions from data items, broadening the expressiveness of our model and supporting the most common types of multi-view interactions. Our meta-framework allows visualization and data analysis experts to focus their efforts on creating data representations and transformations for their applications, whereas nonexperts can reuse previously developed components to design their applications through a user-friendly interface. We validate our approach through a critical inspection with visualization experts and two case studies. We have carefully selected these case studies to illustrate its capabilities. Finally, we compare our approach with the subset flow model designed for multiple coordinated views.

INDEX TERMS Coordinated views, data visualization, exploratory visual analysis, visual programming.

I. INTRODUCTION

The recent technological advances and innovations have increased the available data in many areas. Visualization has proven to be a powerful tool for analyzing and understanding complex data sets. Traditionally, the visualization field has been divided into two subfields depending on the nature of the data. The scientific visualization subfield (scivis) handles spatially structured data, whereas the information visualization subfield (infovis) deals with abstract data [1], [2]. In the infovis context, declarative frameworks for visualization,[1] such as *D3* [4], *ggplot2* [5], *plotly* [6] or *Vega* [7], stand out. They simplify the task of designing visualizations since the developers have to specify *what* to do instead of *how* to do it [8]. Most of these frameworks provide mechanisms to interact with data through data views, which is essential to its understanding [2]. In this regard, it is worth noting Reactive Vega, which offers a robust architecture for declarative interactive visualization [7]. As mentioned, visualization

The associate editor coordinating the review of this manuscript and approving it for publication was Dongxiao Yu.

[1]also known as declarative languages for visualization or visualization grammars [3]

can be applied to several fields, each of them with specific needs and tasks. Although the infovis frameworks, in particular declarative frameworks for visualization, have a general purpose and can be applied to a wide range of problems, they do not usually offer the specific capabilities of the scivis frameworks. Each visualization framework has its advantages and disadvantages, and therefore there is not a unique strategy for all possible scenarios. Covering all the needs of the fields in which visualization can be used effectively is unfeasible. Besides, new techniques are released every year to address specific tasks. This should encourage the designers of these frameworks to provide extensibility in a user-friendly way, allowing users to incorporate new functionality. Many visualization frameworks do not provide extensibility capabilities. Adding this feature to grammar-based visualization frameworks is especially challenging due to their structure and high level of abstraction. For this reason, it is necessary to allow designers to integrate data views created with different visualization frameworks.

Furthermore, in most cases, a single data view is not enough to represent complex data. In this regard, faceting data across multiple views is one of the most relevant approaches for highlighting different data features and handling visual clutter problems. Multiple view strategies can be classified into three groups (see [2] for further details): juxtaposition, data partitioning, and superimposition. Whereas superimposition divides a data visualization in multiple layers, the first two approaches (juxtaposition and data partitioning) require coordinating multiple data views for their effective implementation. A meta-framework to communicate visualization frameworks, including declarative-based approaches, that could seamlessly include new data visualization techniques and provide interoperability among them, could greatly simplify the task of creating domain-specific applications.

Some authors have identified that the learning curve of most of the available declarative frameworks for visualization is still steep, hindering the access to many final users [9]. Data flow diagrams (DFDs) naturally represent sequences of operations and data transformations. Due to their ability to divide complex processes into simpler and meaningful blocks, they have been used to design visual programming frameworks as a higher-level alternative to traditional programming languages (see Section II-C). In the data analysis and visualization field, DFD-based visualization frameworks have also proven to be valuable tools (see Section II-D).

In this paper, we propose VMetaFlow, a data visualization meta-framework for interactive coordinated-views application design. Our system acts as an abstraction layer over existing visualization frameworks providing them with interoperability mechanisms through a novel data flow model. We take advantage of the ability of DFDs to simplify complex problems to modularize the exploratory analysis task, from data processing to visualization. VMetaFlow relies on existing visualization frameworks to create each data view and its associated interactions. Users can select the most appropriate framework for each data view. The only requirement is that the visualization framework must be accessible through a *JavaScript* API. Then, VMetaFlow graphical user interface (GUI) allows users to specify how data flows between views and how these views are coordinated in the same DFD. In contrast to other works, we integrate visualization properties (e.g., data encodings, camera position, axis title, selections) within the DFD and decouple them from the data under analysis. Our approach allows users to model the most common types of interactions in juxtaposed and data partitioned multi-view applications (e.g., zoom and pan coordination between two scatter plots – see [2] for further details) and enables fine-grained control over their scope.

VMetaFlow was designed for users with different levels of knowledge in data visualization and analysis. Advanced users can extend the functionality of our meta-framework by creating the basic components of the DFD, whereas users without experience in the field of visualization can reuse these components to design applications to analyze their data.

Our meta-framework aims to cover two types of visualization applications: exploratory analysis and fixed workflow applications. Both approaches are essential tasks in the data science field. The former allows users to interact with their data to gain preliminary insights and form hypotheses. Whereas the latter allows them to apply predefined methods to known problems for knowledge extraction. These two tasks benefit from defining the application behavior using a DFD. This feature is essential for exploratory analysis because it allows users to interactively modify the workflow structure to test new hypotheses and to include and test new functionalities during the prototype development. Furthermore, having access to the DFD helps users understand the data transformation process and enables fast prototyping.

We argue that visualization itself is not enough to achieve the objectives of the aforementioned application types because relying only on perception could lead to a misunderstanding of data meaning. Consequently, we claim that it is essential to provide statistical and data processing tools to confirm or reject the user intuitions. Our meta-framework addresses this by means of cards in which the user can encapsulate scripts written in several programming languages, such as *R*, *JavaScript*, or *Python*.

The main contribution of this paper is to propose and develop a visual meta-framework and data flow model to integrate third-party visualization frameworks and data processing algorithms, providing them with interoperability mechanisms to design coordinated view applications (see Fig. 1). As mentioned, the only requirement is that the visualization framework must provide *JavaScript* APIs. Data processing algorithms can be written in several programming languages: *R*, *JavaScript* or *Python*. Our data model decouples the visualization properties from the data, allowing users to represent both the flow of data and the interactions between coordinated views. Finally, to support the two types of applications mentioned above and users with different levels of expertise in data analysis and visualization, we have taken into account the following requirements:

1) *Data processing*. VMetaFlow permits the integration of data analysis algorithms within the workflow.
2) *Extensibility and reusability*. Our meta-framework allows incorporating new visualization techniques, data processes, and data types as DFD elements. These elements and the DFD layouts can be stored, shared, and reused.
3) *Easiness of use*. Users can create the DFD by dragging, dropping, and connecting elements into the workspace.

The rest of the paper is organized as follows. Section II reviews the most relevant related works. Then, we present an overview of the meta-framework in Section III. The components and structure of our DFDs are described in Section IV and Section V. In Section VI, we explain the mechanisms that our system provides to support extensibility and reusability. The architectural details of VMetaFlow are shown in Section VII. Section VIII describes the case studies and the external critical inspection carried out to validate our system. Finally, Section IX presents our conclusions.

II. RELATED WORK

Declarative frameworks for visualization (see Section II-A) are widely adopted general-purpose frameworks for data visualization design. Their ability to handle abstract data at a high level allows them to adapt to numerous fields, reducing the designers' effort. The need to enhance these frameworks with domain-specific visualizations motivated the development of VMetaFlow. Our meta-framework allows the seamless integration of visualization frameworks, including declarative approaches. In this regard, our software bridges the gap between declarative frameworks for visualization and scivis frameworks, such as VTK [10], ITK [11] or deck.gl [12]. As mentioned above, the use of multiple data views is one of the most popular strategies to handle data complexity (see Section II-B). Users can select the visualization technique that better fits their needs to create each data view, enabling their cooperation and addressing their diversity.

Nowadays, data visualization is a popular tool in many fields. For this reason, a lot of research in this area has focused on simplifying the authoring process so that end users can tailor the visualization to their needs. Within this category, we find from early approaches, such as SAGE [13], to academic prototypes, such as Lyra [14] or iVisDesigner [15], or commercial frameworks with thousands of users, such as Tableau, the successor of Polaris [16]. Unlike these frameworks, VMetaFlow relies on DFDs since they model data workflows naturally (see Section II-C). DFDs have been used in the data visualization context (see Section II-D). Nevertheless, most of them offer a limited set of built-in data views and little support for adding visualizations from other frameworks. VMetaFlow shares and enhances previous data-flow-based visualization frameworks features, focusing on providing multiple-view coordination support to data views implemented with different visualization frameworks.

A. DECLARATIVE FRAMEWORKS FOR VISUALIZATION

Wilkinson's Grammar of Graphics [17] and HiVE [18] were among the first works to propose the use of declarative specifications for visualization. These approaches provided a high level of abstraction and supported rapid analysis, but did not offer fine control of graphics and interactions [8]. More recent declarative visualization frameworks, such as D3 [4], ggplot2 [5], plotly [6] or Vega [7], have a higher degree of customization, with the drawback of being more complex for users with little or no programming experience. *Vega-Lite* offers a higher degree of abstraction, but still requires a strong background in programming [3]. In our meta-framework, visualization experts can create data views using the previously mentioned declarative visualization frameworks. Then, domain experts can use them to create multiple-view applications.

B. COORDINATED MULTIPLE VIEWS

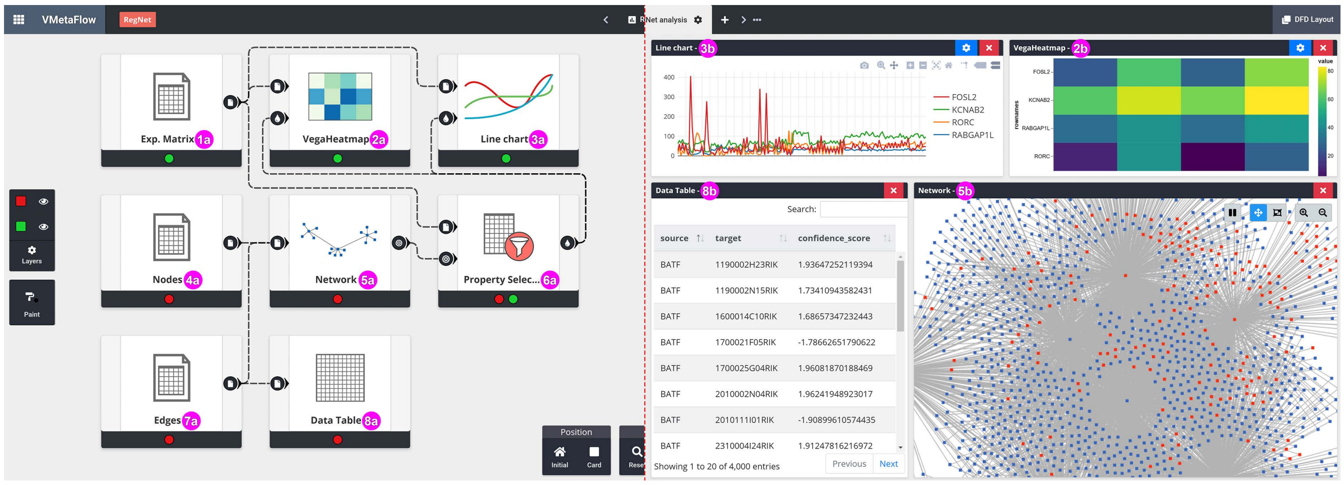
Coordinated multiple views have been a central topic in data visualization and exploratory analysis, as it is a powerful strategy to deal with data complexity [19], [20]. Enabling users to interactively analyze data from multiple and coordinated visualizations boosts the knowledge extraction task. In this regard, multiple works have been proposed that tackle this problem from different perspectives. Prates *et al.* [21] proposed a coordination model based on ontologies. This approach employs semantic representations to relate data items. Those relationships are only contained in the ontology. However, this approach relies on having an ontology about the data to be analyzed, which introduces an overhead that the user has to face. Improvise [22] proposes a coordination model based on *live properties*, that enable the synchronization of values from several visualizations through shared objects called *variables*. To enable coordination, users must link equivalent view properties to the same variable through a menu-based GUI.

Although these frameworks are somewhat extensible, data flow diagrams provide a more natural way to define data pipelines and, as demonstrated by Yu and Silva [23], can be tailored to efficiently design multiple-view interactions.

C. VISUAL PROGRAMMING

Traditionally, a program is generated from a structured sequence of words with a syntactical meaning. Alternatively, visual programs use graphics and two-dimensional layouts as part of the program specification [24]. This approach is easier to understand and work with, as it resembles the human mental representation of problems. Unlike the one-dimensional textual way, visual programming uses higher-level descriptions of the program functionality. Users without programming skills find this approach more accessible [25].

Commonly, visual programming environments use data flow diagrams. These programming environments are based on boxes that encapsulate a piece of functionality and wires that connect them. Data are transformed in the diagram boxes



Data Table - 8b

Search:

source	target	confidence_score
BATF	1190002H23RIK	1.93647252119394
BATF	1190002N15RIK	1.73410943582431
BATF	1600014C10RIK	1.68657347232443
BATF	1700021F05RIK	-1.78662651790622
BATF	1700025G04RIK	1.96081870188469
BATF	2010002N04RIK	1.96241948923017
BATF	2010111I01RIK	-1.90899610574435
BATF	2310004I24RIK	1.91247816216972

Showing 1 to 20 of 4,000 entries

Previous Next

FIGURE 1. Gene Regulatory Network Analysis. This figure introduces an application example used as a first case study. The left side of the figure shows a DFD composed of several cards for loading (1a, 4a, 7a), filtering (5a, 6a) and visualizing data(2a, 3a, 5a, 8a). The force-directed graph filters the data shown in the heatmap and the line chart. We implemented all these data views using different framework. On the other side, a visualization panel with four views was created from the visualization cards defined in the previous DFD. We are using the same numbering for the DFD cards and their corresponding visualization panel. We label cards and visualization with the suffix *a* and *b*, respectively.

and flows from one execution node to the next one following the diagram wires. Early works that followed this approach were applied in a wide range of domains such as music, image processing, or UI construction [26]. Those systems offered a wide selection of variations of the data flow that supported iteration, procedural abstraction, or type checking, among others. Howland *et al.* [27] successfully applied this model to ease out the process of scripting plot events in a role-playing game. The data flow model is also employed in Unreal Engine [28]. This game development system eases the programming aspect of this domain by representing functions, events, and classes as nodes and wiring them together. This simplified perspective is used to allow designers to extend baseline applications created by programmers. Finally, this model is used by the main tech companies in their cloud services, to allow the construction of pipelines for machine learning and data processing [29], [30], [31].

Given the aforementioned benefits and the wide adoption of this model, we propose a DFD-based visualization meta-framework to provide interactivity mechanisms between visualization frameworks and data processes developed with different programming languages.

D. DATA FLOW FRAMEWORKS FOR VISUALIZATION

In the visualization and analysis domain, DFDs have been applied in multiple ways. VisComposer [9] uses DFDs to intuitively create visualizations. Although its ability to model user interactions is limited, it replaces current declarative frameworks for visualization. Mendez *et al.* [32] proposed iVoLVER for visualization authoring. Their DFD-based system was designed to support a wide variety of data types and sources. Some authors have proposed the use of DFD for scivis. In this field, DFDs are primarily used to define 2D or 3D rendering pipelines [10], [11], [33]. Early systems focused on creating workflows that simplified the rendering process of scientific computations. SCIRun [34] provided visualization as a way of aiding the computation steering of scientific algorithms. IBM Data Explorer [35] established the baseline to tackle several problems inherent in using this approach for data analysis and visualization. The authors proposed a data flow system that supports several types of data, visualizations, and processing modules. Moreover, they focus their work on the optimization of the data flow execution. To this end, they introduce a cache to store the intermediate results of each node and a graph analysis step to identify which nodes should be re-executed. The main focus of these systems is the data processing; the visualization is relegated to represent the results of this task. The use of charts for data exploration and queries is severely restricted.

Focusing on the exploratory analysis task, ExPlatesJS [36] and VisTrails [37] are worth noting. The former implements a methodology for separating the visual exploration steps. Whereas the latter provides data provenance support. In addition to the previously mentioned systems, KNIME [38] is an open-source environment for data science that allows collaboration and workflow reusability. Although VisTrails and KNIME are extensible, all the previous systems mainly rely on their built-in modules and visualizations. Moreover, these systems offer limited support for multiple-view coordination. ExPlatesJS and VisTrails do not provide any support to this end. Whereas KNIME follows a publish-subscribe event model; all visualizations can subscribe to filter or selection events and publish them. The scope of their model is limited to composite views, i.e., events are only distributed between visualizations placed in the same visualization window. This model is not extensible to new interactions, and its restricted scope hinders its applicability in the context of exploratory analysis.

Langner *et al.* [39] concluded that connecting all views by default is not self-evident to users from their user study on coordinated and multiple views. Furthermore, Wang Baldonado *et al.* [40] proposed perceptual cues to display the relationship between views as one of the guidelines when using multiple views. However, none of the previously mentioned approaches offer this type of visual support. In contrast, VisFlow [23] was specifically designed to overcome these constraints. Its authors proposed a data flow model called the subset flow model. In this model, users must start with an overview of the data and then progressively divide it into subsets. This approach simplifies the design of data workflows and facilitates the comprehension of their corresponding DFDs. Nonetheless, this approach constrains the expressiveness of the DFDs. Additionally, in the same way as KNIME, VisFlow attaches visualization properties to the tabular subset items to coordinate multiple views. For this reason, VisFlow only allows users to work with tabular data. In this paper, we propose to extend the idea of using DFD to design interactions between data visualizations to the scenario where those views are created using different frameworks. The subset flow model proposed for VisFlow suffers from limitations that prevent its use in this case. We will discuss these limitations in Section IX.

III. VMetaFlow OVERVIEW

Declarative visualization frameworks have boosted the development of applications that use visual embeddings to represent complex data. The diverse nature of the fields and tasks to which data visualization is applied hinders finding a framework that outperforms the rest in all scenarios. Moreover, new visual representations are developed every year, tailored to address the specific needs of tasks in specific domains.

The use of multiple coordinated views has proven to be an effective tool to deal with data complexity. The goal of this research is to provide a meta-framework where users can integrate several data views regardless of the framework employed to implement them and define interactions between them seamlessly. To this end, we rely on a DFD model. This kind of model has been successfully applied to modelling the coordination between data visualizations. Our DFD model was design to tackle the following challenges:

- *Integration effort*: The development time must be spent designing data views and not incorporating them into the platform. Our system requires a low number of lines of code to integrate visual embeddings.
- *Computational overhead*: Adding software layers increases the use of memory and computation time. The use of these resources must be kept to the minimum.
- *Expressiveness*: The data flow model must not limit the definition of any possible interaction between data views.

IV. DATA FLOW DIAGRAM ENTITIES

DFDs naturally divide problems into a sequence of functional blocks. They are directed graphs that represent how information flows from one node to another. VMetaFlow's cards are the graph nodes, and they encapsulate data processing algorithms and visualizations, whereas connections specify how the data are transmitted between them. In our approach, visualization and data analysis experts focus their efforts on designing and implementing the application's functional blocks (cards) using the technologies that better fit their needs. Then, the application behavior is described connecting cards in a higher abstraction level graphical interface.

A. CARDS

Cards are the fundamental component of our model. A card implements a specific functionality, e.g., a scatter plot or a clustering algorithm, using the visualization or data processing technology chosen by the programmer. From a technical perspective, a card is an encapsulated self-contained module and has everything it needs to carry out its activity. Therefore, they can be developed independently. To minimize the integration effort, VMetaFlow's API is constrained to three operations: receive data, send data, and, optionally, change its internal state (properties). Although allowing programmers to store the internal state of the cards is not an essential feature, it improves the expressiveness of the card. For example, it is possible to adapt a cards' behavior to the user selection history. There are two categories of cards:

- *Visualization Cards* are in charge of creating graphical representation from data. Our system grants them access to a document object model (DOM) node. Any *JavaScript* API can be used to draw the data view. Additionally, visualization cards can be used to create a GUI for the application, implementing elements such as drop-down menus or buttons.
- *Data Processing Cards* are responsible for loading, storing, and processing data. They do not have access to the interface, and they run asynchronously.

B. CONNECTIONS

Connections are mechanisms that manage data sharing between the DFD cards by linking cards' input and output docks. All connections are treated in the same manner to simplify the card design. They are characterized by a data structure which is transmitted from output cards to their connected cards inputs. The format of this structure is strictly enforced. This feature is particularly useful for the integration of different visualization and data processing technologies, as it allows homogenizing the communication between them. The inputs and outputs of a card define its expressiveness. Inputs are associated with card functionalities, whereas outputs expose the results of the operations carried out by the card, either automatically or determined by user interaction.

Data tables are a common way to store information in many scientific and business domains. Many visualization frameworks are focused exclusively on these data structures. VMetaFlow natively implements a set of connections to

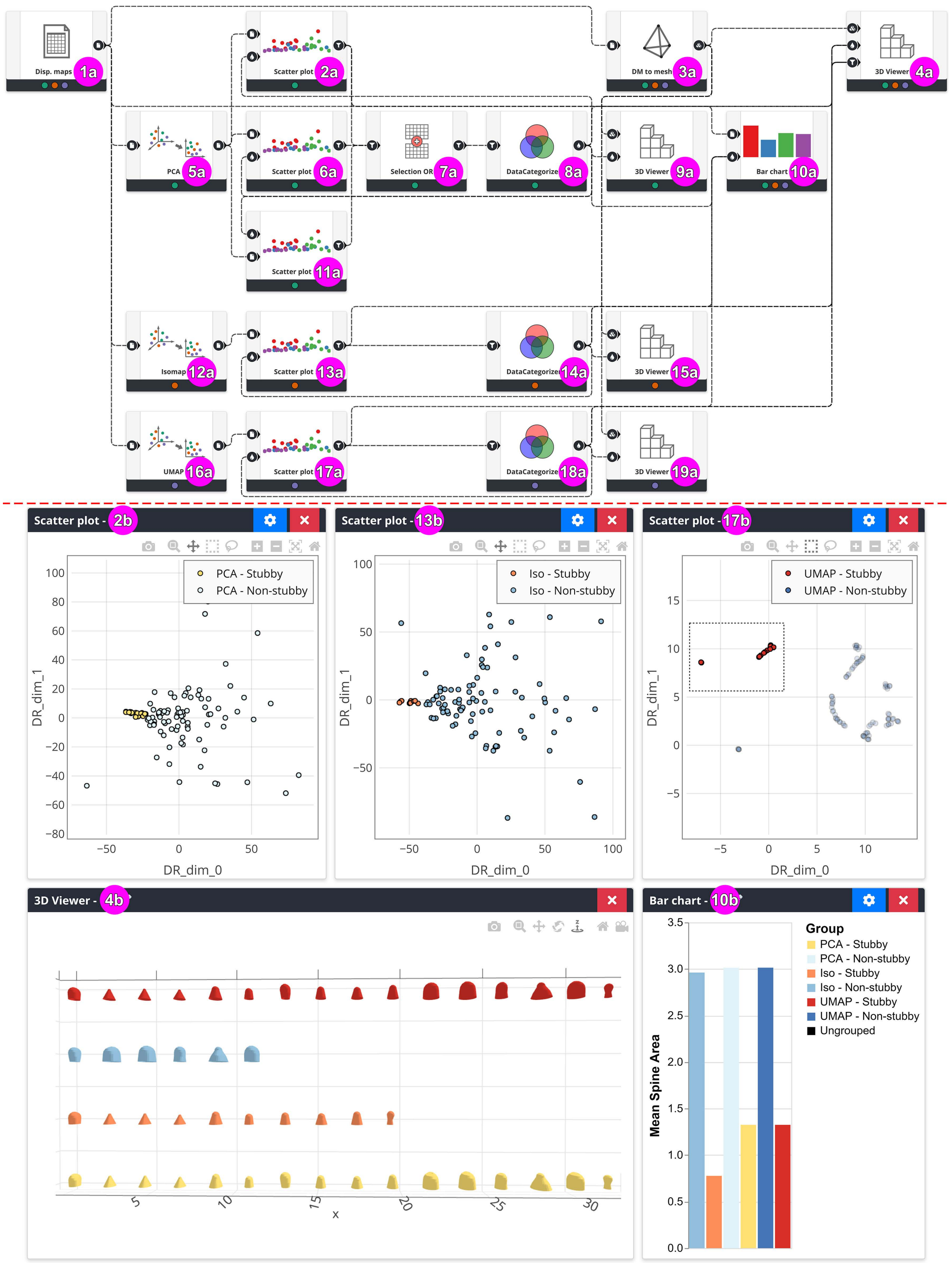


FIGURE 2. Dimension Reduction for Dendritic Spine Clustering. The top part of the figure shows a DFD consisting of a data loading card (1a), transformation of displacement maps to meshes (3a) and dimensionality reduction (5a, 12a, 16a). The data processing cards (3a, 5a, 12a, 16a) are processed on the server. Tabular data are visualized using scatter plots (2a, 6a, 11a, 13a, 17a) and a bar chart (10a). The meshes are visualized using three-dimensional viewers (4a, 9a, 15a, 19a). Data selections created in the scatter plots are used in group creation cards (8a, 14a, 18a). The created groups return to their associated scatter plots and go into the mesh viewers (9a, 15a, 19a, 4a) and the bar chart (10a). The selection combination card (7a) performs an OR operation on the selections coming from the first three scatter plots (2a, 6a, 11a). Finally, the selections from scatter plots (2a), (13a) and (17a) are propagated to the last three-dimensional viewer (4a). On the bottom half, the panel embeds the visualization cards (2a, 13a, 17a, 4a, 10a). We are using the same numbering for the DFD cards and their corresponding visualization panel. Cards are labeled with the suffix *a*, whereas visualizations use the suffix *b*.

handle tabular data efficiently and supports the most common types of card-to-card interactions:

- *Tabular data*. It is an array of data rows and the data columns descriptions. Since tabular data are the most common data type, these connections are optimized following the *changeset* approach used in Reactive Vega [7] (see Section VII-B). Output tabular data connections can be used to implement derived data, filtering, and aggregation.
- *Selection*. It transmits a data query. In contrast to the previous connection, it only shares the IDs of the selected elements and the range of the selection. It can be used to add visualization properties to data subsets.
- *Color*. It transmits groups composed of data items IDs, a color, and a group name.
- *Option*. It allows users to define a set of card properties in the output card and use them to configure the input card. These connections are useful to link the properties of two cards or change visualization options directly in an application panel without modifying the DFD.

In addition to these natively available connections, VMetaFlow also supports the definition of new connections to meet other specific needs and data types.

C. DATA FLOW MODEL

Yu and Silva [23] proposed the first DFD model to design multiple coordinated view applications. Unfortunately, the lack of flexibility of their data subset model hinders the system extensibility, preventing its use to coordinate visualizations developed with different frameworks. VMetaFlow's data flow model conceptually classifies connections into three groups: data to be processed, visualization properties, and card options. Unlike the subset flow model, VMetaFlow does not append visualization properties to the data. They are defined as separate data. This separation is essential for keeping the integration effort low and increasing the model expressiveness. The use of separate connections with explicit data types unambiguously reveals what is expected as input and offered as output. The card programmers can clearly define what the card needs and what information produces. Our system checks types automatically to prevent programming errors. The DFD creation is also simplified since needed inputs and offered outputs are explicitly shown to the designer. Assigning visualization properties to table items severely constrains the model expressiveness. Global properties of the visualization cannot be associated with individual items. Operations such as linking the navigation of two data views are not possible. Moreover, this approach is only valid for tabular data but not for other data types. Separating the data and visualization properties allows us to extend the system to additional types, such as images or b-reps (boundary representations, see Fig. 2).

Another essential feature of our model is that it allows loops. End user's actions on one card can cause another card to change the visualization properties of some data items, and this change can be reflected on the first card. VisFlow does not allow loops. Therefore, in the previous scenario, it would require two data views, one to capture the user actions and another to show the results.

V. GRAPHICAL DESIGN FRAMEWORK

VMetaFlow provides a GUI to boost the integration of data views and processes. Both DFD and panels are defined through this interface.

A. DFD LAYOUT

The data processing and visualization pipeline is defined in the DFD layout tab. Available cards are listed in a modal window, and they can be added to the DFD layout via drag and drop operations. A set of input and output docks and a set of options characterize each card. Connections are typed data shared between cards. The icons on the left (inputs) and right (outputs) show a particular card connection dock type. Fig. 3 shows the process of connecting two cards. Setting up their options and connecting their input and output docks with other card docks builds a graph that defines the application behavior. The left image in Fig. 1 and the top image in Fig. 2 illustrate DFDs of two applications.

B. PANELS

Some works, such as ExPlates, integrate visualizations into the DFD. This approach simplifies the design and exploratory analysis tasks. When creating multi-view applications, designers must consider the cognitive overload of end users [40]. To alleviate visual cluttering, other tools, such as KNIME, separate the multiple-view visualization from the DFD window. Similarly, our GUI separates the data views from the DFD tab using multiple panels. Panels are frames where the results of the DFDs can be visualized. As mentioned in Section I, VMetaFlow aims to cover two types of visualization tasks: exploratory analysis and fixed workflow applications. Data exploration applications benefit from showing the DFD to the final users because it allows them to follow the knowledge extraction process. However, in the static workflow case, users perform a set of actions in a well-known and fixed manner. In this scenario, DFD might distract them from their final goal. The right image in Fig. 1 and the bottom image in Fig. 2 show two panels that arrange visualizations included in their respective DFDs. The number of views that can be shown to the user effectively is limited. With this regard, panels are a way to reduce the number of views shown to the end users at one time. Data views can be grouped based on different criteria, such as a common task or feature. This strategy leaves more room for the visualizations and the DFD in their respective tabs. And it prevents the saturation of the visual channel, enabling users to cope with complex visualizations and covering a broader type of applications.

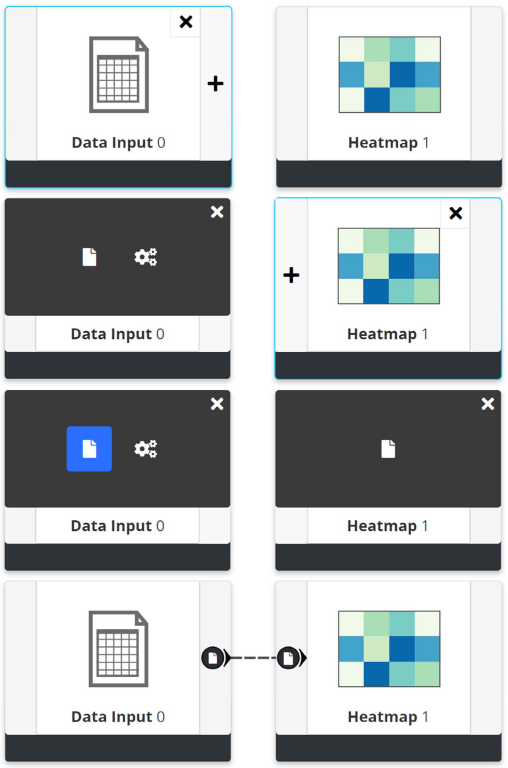


FIGURE 3. Connecting two cards. The upper row figures show two unconnected card icons in the DFD layout tab. The card on the left is in charge of loading a tabular data set, whereas the card on the right is a visualization card. First, the user clicks on the output card to see the available connections (second row). Then, the user selects the data dock on the output card and clicks on the input card to see the compatible dock (third row). The bottom image shows the two cards connected in the DFD layout tab.

VI. EXTENSIBILITY & REUSABILITY

Our system only requires programming skills to create new cards and connections. The DFD and panels can be designed through our GUI. Adding *extensibility* and *reusability* capabilities to the system enables users with no programming skills to develop their applications. Advanced users can create new functionality and share them with the community. In this section, we discuss the mechanisms that VMetaFlow implements to provide extensibility and reusability.

A. APPLICATIONS

One of the main reusability features that VMetaFlow offers is the capability of exporting the DFD layout. We have created a JSON-based DFD interchange format for this purpose. This format includes DFD cards and connections along with the panels. We bundle it in a compressed file together with the implementation of the non-native cards and connections and their respective dependencies. We call applications to these packages, and they can be imported in other VMetaFlow deployments or user accounts.

B. EXTENDING CARDS

New cards can be added to VMetaFlow through its GUI. New visualizations can be created using any web-based framework, such as plotly or Vega. Our Card Creation Assistant (see Fig. 4) reduces the integration effort, guiding the design of new cards in 4 steps:

Dependencies. In the first step, the card designers select the *JavaScript* library files. If the desired library is not available on the platform, it can be uploaded at this point.

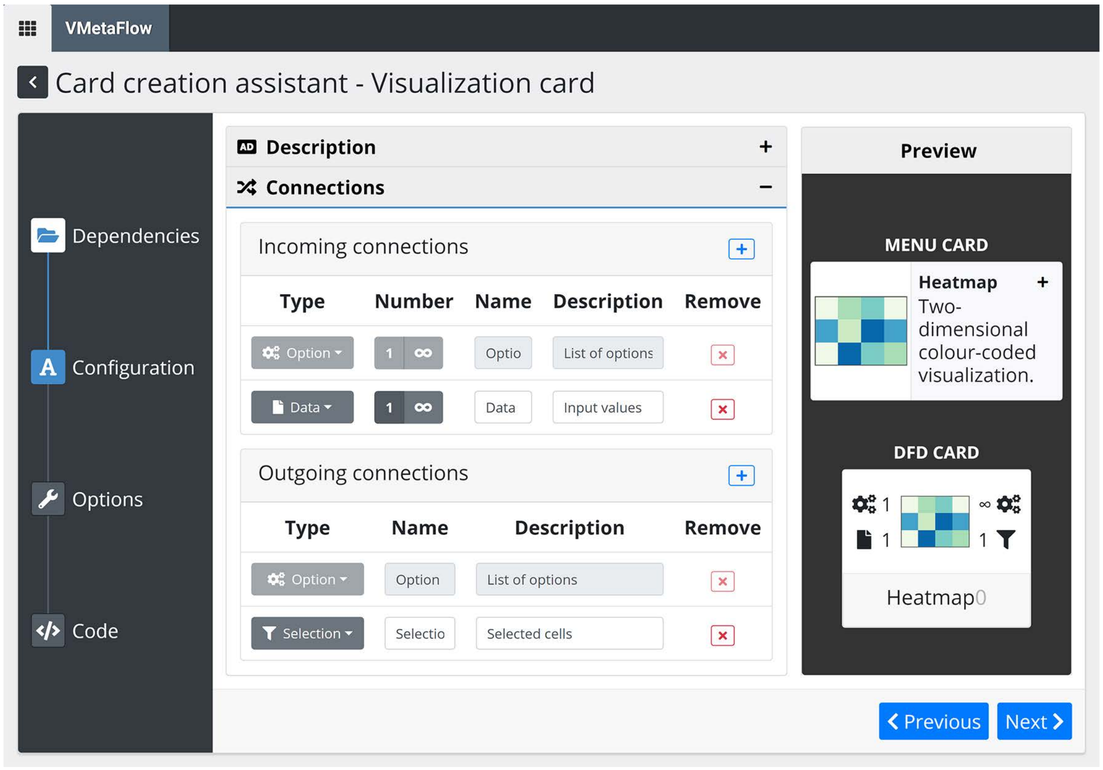
Description. In the second step, the card's description and supported connections are defined. The designer has to specify the card's name, identifier, and description. The connection docks can be added in this step by providing their type, name, and description. Additionally, the cardinality of the input docks has to be defined. This parameter establishes the number of incoming connections allowed in a given dock. Lastly, there are default input and output options docks. These connections allow synchronizing options between cards and define the card options in other cards of the DFD.

Options. The following step provides a way to create the card's options. Options are a set of parameters that define the card's behavior. For example, in a *box plot* the axis, titles, range, variables, etc.

Code. In this step, the designers have to establish the card behavior. The visualization cards are implemented in *JavaScript* using the web-based visualization API selected in the first step. They are required to complete the *init* and *update* functions. The *init* function is called once and initializes the visualization, whereas *update* is called when any input connection or state property changes its value. Both functions receive the same set of parameters: (i) the *DOM* container where the visualization will be embedded, (ii) the *Input* object that receives all the incoming connections values, (iii) the current card internal *State*, (iv) a *DataHandler* instance and (v) a callback function (*setProperty*). The *State* parameter grants access to the current internal variables and to the output values. In fact, output connections share the selected *State* variables with other cards. Developers can add or modify the card's internal *State* and output connection values using the *setProperty* synchronous callback. Finally, the *DataHandler* object optimizes the tabular data treatment (for further details, Section VII-B).

The pipeline to create processing nodes is similar. The main differences are that data processing scripts only have a function and can be written in *R*, *Python* or *JavaScript*. *R* and *Python* scripts run on the server-side, whereas *JavaScript* cards run on the client-side. In order to enable interoperability between *JavaScript* and the other two programming languages, *JavaScript* objects are transformed to *named list* in *R* and to *dictionaries* in *Python*. In the same manner, the *process* function has to be implemented, but it does not have access to a DOM container and the *setProperty* callback is replaced by *setResult*. *setResult* is used to update the card's internal state and to notify the main thread once processing has concluded. Additionally, the *setProgress* callback can be optionally used to inform about the progress made in the data processing.

Besides creating new cards, the card collection can be extended by importing third-party cards. This functionality



Card creation assistant - Visualization card

Incoming connections

Type	Number	Name	Description	Remove
Option	1 ∞	Optio	List of options	✕
Data	1 ∞	Data	Input values	✕

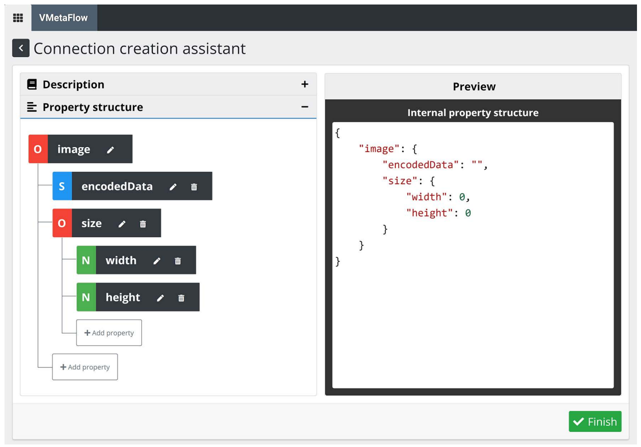
Outgoing connections

Type	Name	Description	Remove
Option	Option	List of options	✕
Selection	Selectio	Selected cells	✕

Heatmap
Two-dimensional colour-coded visualization.

Heatmap0

FIGURE 4. Card Creation Assistant GUI. The image shows the second step of the creation process. The user has added an input data connection and a selection connection. The default option connection is automatically created. Card icons' final appearance are displayed in the preview container.



Connection creation assistant

```
{
    "image": {
        "encodedData": "",
        "size": {
            "width": 0,
            "height": 0
        }
    }
}
```

FIGURE 5. Connection Creation Assistant GUI. The property structure is defined in the second step of this assistant. This structure is a hierarchy of fields with an associated name and type.

follows the application approach described in the previous section.

C. EXTENDING CONNECTIONS

Connections define typed data shared between cards. Section IV-B describes the natively available data types. Nevertheless, custom connections can be implemented through the Connection Creation Assistant. This GUI subsection is composed of two simple steps (see Fig. 5):

Description. Users have to define the connection name, the name of the property to be shared and its associated icon.

Structure. The shared data structure is defined. No coding skills are needed to define the connection's data structure.

Although *JavaScript* is a loosely typed programming language, the system checks the data structure integrity before sharing it through a connection to prevent errors.

VII. ARCHITECTURE

Our meta-framework is an additional software layer placed over existing data visualization frameworks and, thus, increasing the use of computational resources. In this section, we describe the architectural details and how the system reduces this overhead. In order to ensure reproducibility, the full implementation of VMetaFlow can be downloaded from `https://github.com/VMetaFlow/VMetaFlow`.

VMetaFlow follows a client-server architecture. *JavaScript* was mainly used to develop both the front end and back end. The server runs in *Node.js* to simplify the development of stand-alone applications and reduce the communication complexity with the client since both use the same data structures [41]. The back end stores the user's data, cards, connections, and applications, whereas the client is in charge of most of the data processing. However, the server executes the *R* and *Python* scripts. This separation allows users to shift the most time-consuming tasks to the server.

A. WORK SCHEDULER

Our work scheduler was designed to avoid unnecessary updates. Cards are queued to be updated when: (i) an option of the card changes, (ii) the user interacts with the card (only for visualization cards) or (iii) the card receives new data. If the card state changes, the system checks whether the updated properties are connected with other cards and propagates the event. The work scheduler queues the cards connected to an updated property only if they are not already in the queue. Before executing any card, the system ensures that all their ancestors have finished their work. Aiming at applying this execution model, the system stores the DFD as a directed graph.

B. HANDLING TABULAR DATA

Nowadays, the visualization of tabular data is especially relevant in the information visualization field. For this reason, VMetaFlow optimizes the treatment of this type of structures. Whereas the card's internal state stores non-tabular data variables, the IndexedDB available in most commercial Internet browsers stores tabular data [42]. This standard grants access to a website-specific *NoSQL* database from the browser. The *DataHandler* instance, available in the card's *init* and *update* functions (see in Section VI-B), provides an interface to access, modify, and create tabular data. We implement the Reactive Vega *changeset* approach [7] to optimize the memory usage and computational performance of our system.

VIII. RESULTS

In this section, we illustrate the capabilities of our DFD model. First, we provide an example of the integration effort required to create a bar chart. Then, we present two case studies. For the first case study, we implement a gene regulatory network analysis to test the expressiveness of VMetaFlow. This case study was originally proposed to validate the VisFlow's subset flow model [23]. VMetaFlow replicates the case study, using visualization cards developed with different frameworks enabling their seamless interaction. In the second case study, we present a workflow design for exploratory analysis from a preliminary stage of ongoing research. We highlight the VMetaFlow's ability to combine visualization with automatic data mining techniques. For this example, we have built a more complex data flow. Additionally, new cards have been implemented to visualize new data types.

Critical inspections are common forms of evaluating visualization frameworks and techniques [43]. To this end, our meta-framework was examined by four external experts in visualization and computer graphics. In this section, we will report their feedback.

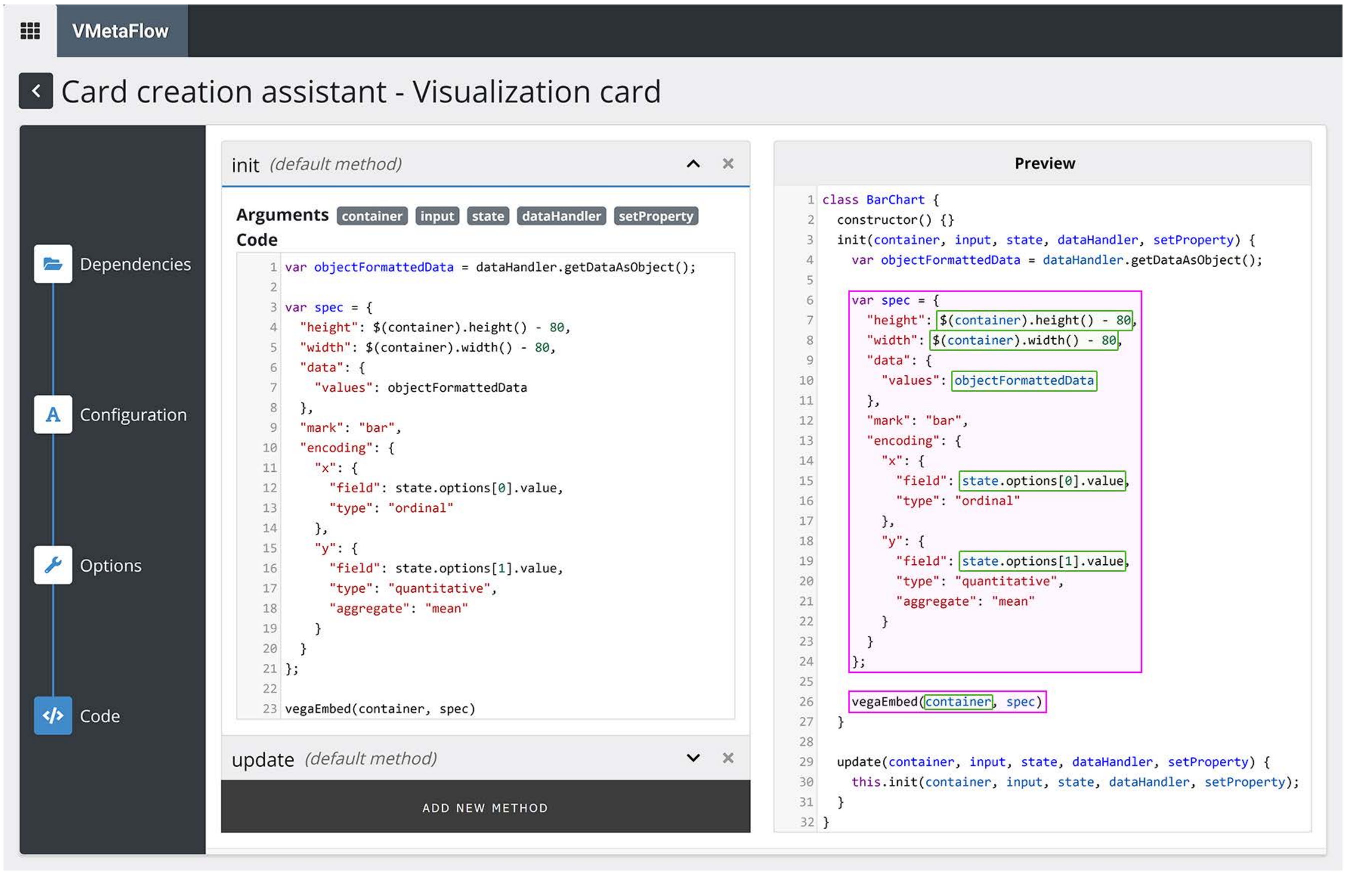
A. CREATING A BAR CHART

In Section VI-B, we discussed the complete process of integrating new cards, Fig. 6 shows an example of the coding step of this process. This figure illustrates the minimum *JavaScript* code required to integrate a basic bar chart defined with Vega-Lite. The first step in the *init* method is to obtain data from the *data handler*. Next, we code the visualization in the format dictated by Vega-Lite. For the dimensions of the visualization, we use the size of the container that will display it. In addition, two inputs have been defined in the previous step to allow user interaction with the visualization. The value of these fields is extracted through the *state* variable and used in the visualization specification. Thus, the users can control which data fields they use for grouping and aggregation. Lastly, we use the Vega-Lite API to produce the bar chart in the container. For the *update* method, we recreate the visualization by calling the *init* method.

B. GENE REGULATORY NETWORK ANALYSIS

In this case study, we mimic the analysis task defined to validate VisFlow [23] to show our system's flexibility and potential to implement any visualization workflow. This task is common in the genetics domain since it shows the regulations between the genes, namely which genes activate or repress others. The input data set is a validated regulatory network for Th17 cells [44], that play a central role in the progress of autoimmune diseases and cancer [45]. The application consists of four coordinated views: a gene regulatory network, a heatmap that shows the gene expression matrix, a line chart for the gene expression profile and a data table with further information on each gene. Each view was implemented with a different framework to show how our system provides interoperability capabilities. The gene regulatory network is a directed weighted graph whose nodes are genes, and the edges are transcription factors. The weights represent the confidence score of regulation. This graph was implemented using *VivaGraphJS* [46]. The heatmap was designed with Vega-Lite [3] and illustrates the gene expression matrix, where rows are genes and columns are experimental conditions. The line chart is a complementary graph that displays the selected gene expression profiles, namely the rows of the gene expression matrix. We used plotly [6] for its implementation. Finally, we created the data table with *DataTables* [47]. The code of most cards was taken from public and private repositories, and most of the development time was employed to integration tasks and testing. The developer estimates to have spent 75 minutes in integration tasks. The card implemented with VivaGraphJS was the most challenging since it required the development of interface controls.

Fig. 1 shows the proposed DFD and the resulting visualization cards. The nodes on the first column of the DFD are in charge of loading data: the expression matrix (1a), and the directed graph nodes (4a) and edges (7a). The expression matrix data feed the heatmap (2a) and the line chart (3a), whereas the nodes and edges are visualized in the network



```
var objectFormattedData = dataHandler.getDataAsObject();

var spec = {
  "height": $(container).height() - 80,
  "width": $(container).width() - 80,
  "data": {
    "values": objectFormattedData
  },
  "mark": "bar",
  "encoding": {
    "x": {
      "field": state.options[0].value,
      "type": "ordinal"
    },
    "y": {
      "field": state.options[1].value,
      "type": "quantitative",
      "aggregate": "mean"
    }
  }
};

vegaEmbed(container, spec)
```

```
class BarChart {
  constructor() {}
  init(container, input, state, dataHandler, setProperty) {
    var objectFormattedData = dataHandler.getDataAsObject();

    var spec = {
      "height": $(container).height() - 80,
      "width": $(container).width() - 80,
      "data": {
        "values": objectFormattedData
      },
      "mark": "bar",
      "encoding": {
        "x": {
          "field": state.options[0].value,
          "type": "ordinal"
        },
        "y": {
          "field": state.options[1].value,
          "type": "quantitative",
          "aggregate": "mean"
        }
      }
    };

    vegaEmbed(container, spec)
  }

  update(container, input, state, dataHandler, setProperty) {
    this.init(container, input, state, dataHandler, setProperty);
  }
}
```

FIGURE 6. Example of a bar chart visualization created using Vega-Lite. The card creation assistant during the coding step. At this point, the default methods are implemented (*init* and *update*). On the right side, the complete code preview is shown to aid the user. We highlighted the code corresponding to Vega-Lite and the data provided by VMetaFlow in pink and green, respectively.

graph (5a) and the data table (8a). Users can select nodes in the network graph. These selections are used to filter (6a) the data shown in the line chart and the heatmap. We assigned DFD nodes that process or visualize network data and nodes that display gene expression matrix data to the red and green layers, respectively.

C. DIMENSION REDUCTION

The DFD of the second case study implements an exploratory analysis workflow designed for a preliminary stage of ongoing research. Dendritic spines (spines) are protrusions found over the dendritic surface of cerebral cortex pyramidal cells [48]. The shape of the spines is related to synaptic plasticity, but their function is unknown. The workflow shown in top side of Fig. 2 explores the possibility of applying dimensional reduction techniques for spine analysis.

The analysis starts loading a planar parametrization (displacement map) of all spines' surfaces [49]. Each displacement map is a 65 × 65 pixel color image. Unlike 3D meshes, dimensionality reduction can be directly applied to these structures since they have a fixed size. In this study, we want to compare the performance of several dimensionality reduction techniques when clustering the planar parameterization of dendritic spines according to their shape. In summary, the process we followed for this task consisted in (i) applying the dimensionality reduction technique to the displacement maps, (ii) visualizing the result using a scatter plot, (iii) creating clusters from the data and (iv) analyzing whether all spines in each cluster have the same shape.

The displacement map set of all the spines is stored in a tabular structure for simplicity. Card (1a) is in charge of data loading. These data are transferred to dimensionality reduction cards: (5a) performs a Principal Component Analysis (PCA), (12a) uses Isomap, and (16a) uses a Uniform Manifold Approximation and Projection (UMAP). These algorithms have been implemented using the same card type, configuring its options to select the technique and the number of dimensions in the projected space (3 for PCA and 2 for the rest). To visualize the results, we use three scatter plots for PCA (2a, 6a, and 11a), one for Isomap (13a), and UMAP (17a). The scatter plots propagate their selection to cards designed to create groups of elements (8a, 14a, 18a). The scatter plots linked to the PCA combine the three selections using an OR operator (7a) before (8a). Then, we recover the surface mesh of every spine from its planar representation (3a) to allow the visual inspection of the clusterization in their corresponding 3D-viewer (9a, 15a, 18a). Finally, to compare the clusterization performed with the three dimensionality reduction methods, we use a 3D viewer (4a) and a bar chart (10a) that allows data superimposition. To avoid visual cluttering, (4a) only shows the data selected on (2a), (13a), and (17a).

We group the visualization cards into four panels to reduce the number of views shown at the same time. Three panels are used to cluster spines using different projections. The last panel compares the clusterization results. We illustrate this final panel on the bottom side of Fig. 2. As a result of the above analysis, we conclude that UMAP should be the preferred DR algorithm for an automatic spine classifier.

As in the previous example, most cards were implemented using code taken from public and private repositories, and most of the development time was spent integrating and testing. The developer estimates to have spent 175 minutes on integration tasks. 90 minutes were spent in the development of the clusterization card (8a, 14a, 18a) since the developer had to implement the card's GUI with *JavaScript* from the ground up.

D. EXTERNAL CRITICAL INSPECTION

We have contacted four experts in visualization to provide a critical evaluation. None of them has been involved in the project, nor have they seen the system beforehand. Half of them work in academia, and the other two work for two different private companies in the visualization field. They have a high level of education (three of them have a PhDs, and the other a master's degree). Three of the participants have more than ten years of experience, whereas the other has five. The same methodology was followed with all participants. We had a presentation session with an open debate. All sessions have lasted between 50 and 90 minutes. Finally, the experts were asked to complete a questionnaire.

All the participants considered that VMetaFlow provides a meta-framework to integrate coordinated view visualizations. They noted that our DFD approach allows users to easily define card-to-card complex behaviors. They also highlighted their value for rapid prototyping and the exploration of new ideas. Additionally, the two industry experts emphasized the usefulness of allowing the cooperation between workers with different levels of knowledge in visualization and data analysis. Three of the participants explicitly remarked that they consider VMetaFlow simple and user-friendly. One of them suggested adding native support for more data types besides tabular data and the other suggested adding a tutorial on how to create the first DFD. We consider these suggestions as limitations of the implementation rather than the conceptual framework.

The fast-prototyping capabilities make VMetaFlow especially attractive in academia. Industry experts recognized the system value as proof of concept. However, they underline several problems that hinder its incorporation into the industry. They consider that a vast set of predefined cards is needed to enable effective use of the meta-framework by users with no programming skills. Since the flexibility of VMetaFlow is one of the system's most relevant advantages, they propose to support natively and efficiently more data types (not just tabular data) to improve capability over existing frameworks. Finally, all experts showed concerns about the data size that can be handled by VMetaFlow. We will further discuss this point in the following section.

IX. CONCLUSION

Multiple views and interaction have proven to be two of the most valuable approaches to handle complexity in the data visualization field. We designed our system to boost the prototyping of multiple-coordinated-view applications. Each view can be created using the most adequate visualization framework (or even an ad hoc data view implemented by the user), and VMetaFlow simplifies the interaction among them. Cards are the minimal functional unit of our system. Data views and their corresponding interactions are embedded in individual view cards. Users must divide their problem into visualization and data processing cards which can be implemented using the optimal visualization or data processing technologies to solve each problem. Interactions between views can be described at a higher level in the application DFD. This approach promotes modularity and enables extensibility and reusability, which are essential for fast prototyping. Additionally, extensibility and reusability allow this meta-framework to tackle a wide variety of tasks in different scientific and business domains. VMetaFlow's DFDs can be used in three different ways to support the types of applications described in Section I: (1) users can interactively manipulate both the DFD and the application panels (exploratory analysis), (2) users can only visualize the layout (fixed workflow applications), and (3) users can manipulate the layout and examine the DFD, without changing it (exploratory analysis and fixed workflow applications).

We separate the data views from the DFD tab to reduce user cognitive overload. However, displaying the visualization on the DFD simplifies interactive exploratory analysis. We are currently working on offering both possibilities, following a similar approach to VisFlow. This framework implements a transition animation between the DFD tab and the display panel that allows users to easily understand the correspondence between DFD nodes and views.

In VMetaFlow, interactions between views are explicitly defined in the DFD. Several authors have pointed out that the relationship between coordinated views is not always evident [39], [40], [50]. Displaying interactions in the DFD helps with the implementation, debugging, and use of coordinated view-based applications.

In this paper, we proposed a data flow model that overcomes the limitation of the subset flow model (see Section IV-C for further details). Separating visualization properties from data items provides flexibility to include new data types. Additionally, it allows sharing global visualization options and properties. Linking cameras is a common technique used to juxtapose side-by-side views that show different data with the same encoding. This behavior cannot be implemented following a subset data flow. DFD loops are an essential feature to increase the model expressiveness. On the other hand, less flexible models led to simpler DFDs. The VisFlow's DFDs can be easily understood by novice users. This is a relevant advantage during data exploration.

The data model of our DFD-based system can implement any complex juxtaposition (which is the most common multi-view strategy) behavior once the cards are created. Other multiple-view strategies are supported, but constrained. User can implement DFDs to partition visualizations into a fixed number of views. Other type of partitions, such as hierarchical partitions, have to be implemented in a single card. In this case, the user would not benefit from our modularized DFD approach. Nonetheless, partitioning requires using the same view in all partitions. Hence, it does not make sense to use different frameworks in each data view. Similarly, layer superimposition is limited. Currently, visualization cards cannot be superimposed. Card overlapping can be easily included in a future version of our system, but more sophisticated superimpositions have to be implemented following a single card approach. As mentioned in previous sections, our system allows creating cards with multiple data inputs. These cards can superimpose data coming from different sources. Nevertheless, creating interactive multi-layer data views using different frameworks is an interesting open problem that will be studied in future research.

There exists a limit on the number of multiple views and the interaction complexity that users can manage. We added multiple panel support to alleviate this problem, allowing designers to arrange the visualizations. Each panel can focus on specific data features or tasks, and users can easily change between them. There are other techniques to handle complexity, such as deriving data, filtering, and aggregation, which can be included in our system through cards.

Domain experts with little visualization experience can create applications reusing existing cards, whereas designers with experience in visualization framework APIs can extend the system functionalities by developing cards. The two industry experts highlighted the usefulness of this approach. However, they found it limited in its current state. They consider that the current number of predefined cards is insufficient for its use by nonexpert users in industry. One of them recommends creating an open access repository accessible from our system to create a community of users and contributors.

Although VMetaFlow has been implemented for handling tabular data efficiently, users can define their connection data types through the Connection Creation Assistant. It must be pointed out that the non-native data treatment is not optimized. We store the non-native data types in the internal state of the card that generates or modifies them. This approach is adequate for small data sets or if the data are not modified. To promote VMetaFlow over other systems, surveyed external experts recommend extending our meta-framework adding native support to other data types, such as graphs, and finding an efficient way to handle non-native data types.

Regarding data analysis capabilities, we have shown in the case studies that VMetaFlow enables to carry out essential operations in this domain, such as dimension reduction. Furthermore, our meta-framework is not limited to those techniques, and any algorithm (implemented in *R*, *Python* or *JavaScript*) can be embedded in a processing card. Some data processing cards run on the server-side, allowing computationally demanding tasks to be executed in powerful dedicated servers. We plan to continue developing this idea, preparing the system to run not only processing cards on the server but visualization ones. We will optimize the system to keep the data transfer between the server and the client to the minimum.

Despite the popularity of web-based visualization frameworks, all surveyed experts agree on the constraints of these systems when handling large data sets. We believe that a future version of VMetaFlow can get around this limitation by adding a third card type. These cards will perform filtering, aggregation, and data derivation on the server-side, only transferring it to the client-side when necessary. This approach will alleviate the computational constraints of web-based visualization frameworks.

ACKNOWLEDGMENT

The authors would like to thank the computer resources at Artemisa, for the technical support provided by the Instituto de Física Corpuscular, IFIC (CSIC-UV).

REFERENCES

[1] S. K. Card, J. D. Mackinlay, and B. Shneiderman, Eds., *Readings in Information Visualization: Using Vision to Think*. San Francisco, CA, USA: Morgan Kaufmann, 1999.

[2] T. Munzner, *Visualization Analysis and Design*. Boca Raton, FL, USA: CRC Press, 2014.

[3] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, "Vega-lite: A grammar of interactive graphics," *IEEE Trans. Vis. Comput. Graphics*, vol. 23, no. 1, pp. 341–350, Jan. 2017.

[4] M. Bostock, V. Ogievetsky, and J. Heer, "D^3 data-driven documents," *IEEE Trans. Vis. Comput. Graphics*, vol. 17, no. 12, pp. 2301–2309, Dec. 2011.

[5] H. Wickham, "Data analysis," in *ggplot2*. Cham, Switzerland: Springer, 2016.

[6] Plotly Technologies Inc. (2015). *Collaborative Data Science*. Montreal, QC, USA. Accessed: Sep. 15, 2021. [Online]. Available: https://plot.ly

[7] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer, "Reactive vega: A streaming dataflow architecture for declarative interactive visualization," *IEEE Trans. Vis. Comput. Graphics*, vol. 22, no. 1, pp. 659–668, Jan. 2016.

[8] J. Heer and M. Bostock, "Declarative language design for interactive visualization," *IEEE Trans. Vis. Comput. Graphics*, vol. 16, no. 6, pp. 1149–1156, Nov. 2010.

[9] H. Mei, W. Chen, Y. Ma, H. Guan, and W. Hu, "VisComposer: A visual programmable composition environment for information visualization," *Vis. Inform.*, vol. 2, no. 1, pp. 71–81, 2018.

[10] M. D. Hanwell, K. M. Martin, A. Chaudhary, and L. S. Avila, "The visualization toolkit (VTK): Rewriting the rendering code for modern graphics cards," *SoftwareX*, vols. 1–2, pp. 9–12, Sep. 2015.

[11] M. McCormick, X. Liu, J. Jomier, C. Marion, and L. Ibanez, "ITK: Enabling reproducible research and open science," *Frontiers Neuroinform.*, vol. 8, p. 13, Feb. 2014. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fninf.2014.00013

[12] Uber. (2016). *Deck.GL: Webgl2 Powered Geospatial Visualization Layers*. [Online]. Available: https://deck.gl/

[13] S. F. Roth, J. Kolojechick, J. Mattis, and J. Goldstein, "Interactive graphic design using automatic presentation knowledge," in *Proc. Conf. Companion Hum. Factors Comput. Syst.*, New York, NY, USA, 1994, pp. 112–117.

[14] A. Satyanarayan and J. Heer, "Lyra: An interactive visualization design environment," *Comput. Graph. Forum*, vol. 33, no. 3, pp. 351–360, Jun. 2014.

[15] D. Ren, T. Hollerer, and X. Yuan, "IVisDesigner: Expressive interactive design of information visualizations," *IEEE Trans. Vis. Comput. Graphics*, vol. 20, no. 12, pp. 2092–2101, Dec. 2014.

[16] C. Stolte, D. Tang, and P. Hanrahan, "Polaris: A system for query, analysis, and visualization of multidimensional relational databases," *IEEE Trans. Vis. Comput. Graphics*, vol. 8, no. 1, pp. 52–65, Jan. 2002.

[17] L. Wilkinson, *The Grammar of Graphics*. New York, NY, USA: Springer-Verlag, 2005.

[18] A. Slingsby, J. Dykes, and J. Wood, "Configuring hierarchical layouts to address research questions," *IEEE Trans. Vis. Comput. Graphics*, vol. 15, no. 6, pp. 977–984, Nov. 2009.

[19] J. C. Roberts, "State of the art: Coordinated & multiple views in exploratory visualization," in *Proc. 5th Int. Conf. Coordinated Multiple Views Explor. Visualisation (CMV)*, Jul. 2007, pp. 61–71.

[20] M. Gleicher, D. Albers, R. Walker, I. Jusufi, C. D. Hansen, and J. C. Roberts, "Visual comparison for information visualization," *Inf. Vis.*, vol. 10, no. 4, pp. 289–309, Oct. 2011.

[21] J. M. Prates, L. P. Scatalon, R. E. Garcia, and D. M. Eler, "Coordinating multiple views using an ontology-based semantic mapping," in *Proc. 17th Int. Conf. Inf. Visualisation*, Jul. 2013, pp. 192–197.

[22] C. Weaver, "Building highly-coordinated visualizations in improvise," in *Proc. IEEE Symp. Inf. Vis.*, Oct. 2004, pp. 159–166.

[23] B. Yu and C. T. Silva, "VisFlow—Web-based visualization framework for tabular data with a subset flow model," *IEEE Trans. Vis. Comput. Graphics*, vol. 23, no. 1, pp. 251–260, Jan. 2017.

[24] B. A. Myers, "User interface software tools," *ACM Trans. Comput.-Hum. Interact.*, vol. 2, no. 1, pp. 64–103, Mar. 1995.

[25] B. A. Myers, "Taxonomies of visual programming and program visualization," *J. Vis. Lang. Comput.*, vol. 1, no. 1, pp. 97–123, Mar. 1990.

[26] D. D. Hils, "Visual languages and computing survey: Data flow visual programming languages," *J. Vis. Lang. Comput.*, vol. 3, no. 1, pp. 69–101, Mar. 1992.

[27] K. Howland, J. Good, and J. Robertson, "Script cards: A visual programming language for games authoring by young people," in *Proc. Vis. Lang. Hum.-Centric Comput. (VL/HCC)*, 2006, pp. 181–186.

[28] Epic Games. (2021). *Unreal Engine*. Accessed: Sep. 15, 2021. [Online]. Available: https://www.unrealengine.com

[29] Google. (2021). *Google Cloud Dataflow*. Accessed: Sep. 15, 2021. [Online]. Available: https://cloud.google.com/dataflow

[30] Microsoft. (2021). *Azure Machine Learning Studio*. Accessed: Sep. 15, 2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/machine-learning/migrate-overview#pipeline

[31] Amazon. (2021). *Aws Data Pipeline*. Accessed: Sep. 15, 2021. [Online]. Available: https://aws.amazon.com/es/datapipeline

[32] G. G. Méndez, M. A. Nacenta, and S. Vandenheste, "IVoLVER: Interactive visual language for visualization extraction and reconstruction," in *Proc. Conf. Hum. Factors Comput. Syst.*, May 2016, pp. 4073–4085.

[33] F. Ritter, T. Boskamp, A. Homeyer, H. Laue, M. Schwier, F. Link, and H.-O. Peitgen, "Medical image analysis," *IEEE Pulse*, vol. 2, no. 6, pp. 60–70, Dec. 2011.

[34] S. G. Parker, M. Miller, C. D. Hansen, and C. R. Johnson, "An integrated problem solving environment: The SCIRun computational steering system," in *Proc. 31st Hawaii Int. Conf. Syst. Sci.*, vol. 7, Jan. 1998, pp. 147–156.

[35] G. Abram and L. Treinish, "An extended data-flow architecture for data analysis and visualization," in *Proc. Vis.*, Oct. 1995, pp. 263–270.

[36] W. Javed and N. Elmqvist, "ExPlates: Spatializing interactive analysis to scaffold visual exploration," *Comput. Graph. Forum*, vol. 32, nos. 3–4, pp. 441–450, Jun. 2013.

[37] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "VisTrails: Visualization meets data management," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2006, pp. 745–747.

[38] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, K. Thiel, and B. Wiswedel, "KNIME—The Konstanz information miner," *ACM SIGKDD Explorations Newslett.*, vol. 11, no. 1, pp. 26–31, Nov. 2009.

[39] R. Langner, U. Kister, and R. Dachselt, "Multiple coordinated views at large displays for multiple users: Empirical findings on user behavior, movements, and distances," *IEEE Trans. Vis. Comput. Graphics*, vol. 25, no. 1, pp. 608–618, Jan. 2019.

[40] M. Q. W. Baldonado, A. Woodruff, and A. Kuchinsky, "Guidelines for using multiple views in information visualization," in *Proc. Work. Conf. Adv. Vis. Interfaces*, New York, NY, USA, 2000, pp. 110–119.

[41] OpenJS Foundation. (2021). *Node.js*. Accessed: Sep. 15, 2021. [Online]. Available: https://nodejs.org

[42] A. Alabbas and J. Bell. (Jun. 2021). *Indexed Database API 3.0*. W3C. W3C Working Draft. [Online]. Available: https://www.w3.org/TR/2021/WD-IndexedDB-3-20210618

[43] T. Isenberg, P. Isenberg, J. Chen, M. Sedlmair, and T. Möller, "A systematic review on the practice of evaluating visualization," *IEEE Trans. Vis. Comput. Graphics*, vol. 19, no. 12, pp. 2818–2827, Dec. 2013.

[44] M. Ciofani, A. Madar, C. Galan, M. Sellars, K. Mace, F. Pauli, A. Agarwal, W. Huang, C. N. Parkurst, M. Muratet, and K. M. Newberry, "A validated regulatory network for Th17 cell specification," *Cell*, vol. 151, no. 2, pp. 289–303, Oct. 2012.

[45] T. Korn, E. Bettelli, M. Oukka, and V. K. Kuchroo, "IL-17 and Th17 cells," *Annu. Rev. Immunol.*, vol. 27, no. 1, pp. 485–517, 2009.

[46] K. Andrei. (2021). *Vivagraphjs*. Accessed: Sep. 15, 2021. [Online]. Available: https://github.com/anvaka/VivaGraphJS

[47] SpryMedia. (2021). *Datatables*. Accessed: Sep. 15, 2021. [Online]. Available: https://datatables.net

[48] R. Benavides-Piccione, I. Fernaud-Espinosa, V. Robles, R. Yuste, and J. DeFelipe, "Age-based comparison of human dendritic spine structure using complete three-dimensional reconstructions," *Cerebral Cortex*, vol. 23, no. 8, pp. 1798–1810, 2013.

[49] M. S. Floater and K. Hormann, "Surface parameterization: A tutorial and survey," in *Advances in Multiresolution for Geometric Modelling*, N. A. Dodgson, M. S. Floater, and M. A. Sabin, Eds. Berlin, Germany: Springer, 2005, pp. 157–186.

[50] M. Steinberger, M. Waldner, M. Streit, A. Lex, and D. Schmalstieg, "Context-preserving visual links," *IEEE Trans. Vis. Comput. Graphics*, vol. 17, no. 12, pp. 2249–2258, Dec. 2011.

NICUSOR COSMIN-TOADER received the B.S. degree in computer engineering from the Universidad Politécnica de Madrid, Madrid, Spain, in 2017, and the M.S. degree in computer vision from Universidad Rey Juan Carlos (URJC), Madrid, in 2019, where he is currently pursuing the Ph.D. degree in computer graphics and high performance computing.

Since 2017, he has been a Research Assistant with the VG-Laboratory, URJC. His research interests include data analysis and visualization, computer vision, and deep learning in medical imaging and signal processing.

FERNANDO TRINCADO-ALONSO received a higher education degree in telecommunications engineering from the Universidad de Alcalá, Madrid, Spain, in 2009, and the Ph.D. degree in advanced electronic systems and intelligent systems from the Universidad de Alcalá, in 2017.

His research interests include virtual reality, brain–machine interfaces for rehabilitation, machine learning, and signal processing. He also worked on scientific visualization in the context of the Human Brain Project, as a member of the VG-Laboratory.

LUIS PASTOR is currently a Full Professor at the School of Computer Science and Engineering, Universidad Rey Juan Carlos (URJC). During the last years, he has coordinated a number of research projects in the areas of virtual reality-based surgical training (holding a patent being commercially exploited nowadays), computer graphics, visualization, and neuroinformatics. His research interests include computer graphics, virtual reality, and visualization for medical and neuroscience applications.

MARCOS GARCIA-LORENZO received a higher education degree in computer science from UPM, Madrid, Spain, in 2002, and the Ph.D. degree from UPM, in 2007. In his Ph.D. degree, he worked on simulation of elastic objects in interactive applications.

From November 2007 to February 2008, he was a Research Fellow at the GV2 Group, TCD, Dublin, Ireland, working with Prof. Carol O. Sullivan. Then, he worked as a Visiting Lecturer at URJC, Madrid, where he has been an Associate Professor in computer science, since 2011. His research interest includes the areas of computer graphics. Recently, he has focused on visualization and data science.