Employing a Parametric Model for Analytic Provenance

Yingjie Victor Chen, Purdue University
Zhenyu Cheryl Qian, Purdue University
Robert Woodbury, Simon Fraser University
John Dill, Simon Fraser University
Chris Shaw, Simon Fraser University

We introduce a propagation-based parametric symbolic model approach to support analytic provenance. This approach combines a script language to capture and encode the analytic process and a parametrically controlled symbolic model to represent and reuse the analytic process. Our approach first appeared in a visual analytics system called CZSaw. Capturing the analyst's interactions at a meaningful system action level as CZSaw scripts creates a parametrically controlled symbolic model in the form of a directed acyclic graph capable of propagating changes. Nodes in the graph (variables in CZSaw scripts) are results (data and data visualizations) generated from user interactions. Graph edges represent dependency relationships among results. The user interacts with the variables representing entities or relations to generate the next step's results. Any change to a variable triggers the propagation mechanism to update downstream variables and in turn update data views to reflect the change. The analyst can reuse parts of the analysis process by assigning new values to a node in the middle of the graph. We evaluated this symbolic model approach through solving three IEEE VAST "Challenge" contest problems [IEEE VAST 2008; 2009; 2010]. In each Challenge the analyst first created a symbolic model to help explore, understand, analyze, and solve a particular sub-problem, and later reused the model via its dependency graph propagation mechanism to solve similar sub-problems.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms: Parametric model, analytic provenance

Additional Key Words and Phrases: parametric model, visual analysis process, analytic provenance, symbolic model

ACM Reference Format:
Chen, V.Y., Qian, C.Z., Woodbury, R., and Dill, J. 2013. Employing a Parametric Model for Analytic Provenance. ACM Trans. Interact. Intell. Syst. Special Issue: Interactive Computational Visual Analytics, 4, Article (March 2013), 34 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Every coin has two sides. The visual analytical process can be both structured and contain unexpected twists and turns. According to Thomas and Cook's "Illuminating the Path", the analytical process is described as "structured and disciplined", and the analyst's problem solving process begins with planning. But they also characterize the process as "detect the expected, discover the unexpected" [Thomas and Cook 2005]. Basically, how to solve the problem, what resources to use, and how to allocate time

This work is supported by grants from the Boeing Company ("Visual Analytics in Canada"), Natural Sciences and Engineering Research Council of Canada (Postgraduate Doctoral Scholarship).
Author's addresses:

to various parts of the process are designed during the beginning stage of analyzing. North et al. [2011] define analytic provenance as "understanding a users' reasoning process through the study of their interactions with a visualization". The analysis process itself is just as important as the final product generated from the process. A number of visual analytic systems have been equipped with functions to capture analytic provenance [Benzaken et al. 2011; Gotz and Zhou 2008; Silva et al. 2007; Eccles et al. 2007; Kreuseler et al. 2004]. Based on the user's involvement, Gotz and Zhou [2008] classified these systems into two categories: manual capture and automatic capture of provenance.

However, even well structured analytical processes have moments of disappointment and serendipity. While investigating data, analysts generate unexpected questions that should be investigated either immediately or revisited later. To support iterative analysis by enabling users to review, retrieve, and revisit visualization states [Shneiderman 1996], history mechanisms such as undo or "time-travel" enable revisitation in a variety of applications [Callahan et al. 2006; Derthick and Roth 2001; Heer et al. 2008; Kreuseler et al. 2004; Shipman 2000]. While history-related tools can play an important part in the visualization process, the best history mechanisms for achieving these benefits are not always clear. Designers of visualization tools must consider a large design space of potential features and system architectures when designing history-related tools. Jankun-Kelly [2007] noted that history alone is not sufficient for analyzing the analytical process with visualization tools. In terms of supporting analytic provenance, we believe that recording the reasoning rationale, logic, and strategy underlining the analysis process is at least as important as capturing the chronological states of the process, and probably more so. Among existing systems, Vistrail [Silva et al. 2007] uses directed acyclic graphs to model and visualize the work flow of a visualization task. EdiFlow [Benzaken et al. 2011] maintains a rough level of provenance by tracking the dependencies between database tables and EdiFlow modules.

In this paper, we present the design and implementation of a parametrically controlled symbolic model approach in the visual analytics system CZSaw [Kadivar et al. 2009] to help analysts investigate entities, relations, and text documents. It was inspired by visual analytics (VA) systems for provenance in visualizations [Silva et al. 2007], social network analysis [Nooy et al. 2005], analysis of document collections [Stasko et al. 2007], and, from computer-aided design, parametric design systems [Aish and Woodbury 2005; Woodbury 2010].

Our model – a symbolic model – represents the analysis process as a directed acyclic graph (DAG) which was called *dependency graph* in CZSaw [Kadivar et al. 2009]. The CZSaw dependency graph provides a visualization of the symbolic model with which the user may interact. Here we reserved the term *dependency graph* to describe particular visualizations of the model. In CZSaw's DAG representation, nodes are results (i.e. variable representing data and data visualizations) generated from user interactions, and edges indicate dependency relationships among results. Any content change to such a variable will trigger the propagation mechanism to update downstream (dependent) variables and, in turn, update data visualizations to reflect the change. Our CZSaw research team used this model in the 2010 IEEE VAST Challenge contest and won the Mini Challenge 1 Award: Outstanding Interaction Model [Chen et al. 2010] and the Grand Challenge Award [Dunsmuir et al. 2010].

Our symbolic model consists of two components: a script language with which to capture and communicate the analysis process (by capturing the analyst's interactions) and a DAG to represent and manage the process. Analysts can interact with CZSaw in several ways: manipulating data views, editing scripts, or modifying values in the symbolic model. Meaningful interactions are captured as statements in a scripting language, which are parsed to construct the symbolic model, with changes propagated down the DAG to update the data and visualizations. In this paper, after reviewing related literature, we introduce the design and implementation of the parametric symbolic model in the context of the VA system CZSaw.

2. CONTEXT OF RESEARCH

2.1. Analytic Provenance in Visual Analytics

Visual analytics is the science of analytical reasoning facilitated by interactive visual interfaces [Thomas and Cook 2005]. The analytical reasoning process includes the construction of arguments, convergent-divergent investigation, and evaluation of alternative hypotheses [Shrinivasan and van Wijk 2008]. Reasoning is the process of using existing knowledge to draw conclusions, make predictions, or construct explanations. Three methods of reasoning are deductive, inductive, and abductive approaches. So the analyst must be aware of what has been done, how to find the patterns or outliers, and how to reuse some strategies during the exploration process to perform effective reasoning. Shrinivasan and van Wijk [2008] argue that the externalization of a mental model is not enough to support the entire reasoning process. Instead, externalizing the evidence, causal links, and processes would be helpful to support the revision and falsification phases in the reasoning process.

Coined by North et al. [2011], *analytic provenance* research aims to understand the analytic reasoning process through investigating the user's interactions because the analytic process itself is just as important as the final product generated from the process. North et al. proposed five interrelated stages of analytic provenance: perceive, capture, encode, recover, and reuse. The first stage, "perceive", focuses on understanding how the data is presented to the user: the data structure, its accuracy, readability, and potential for modification. At the second "capture" stage, the linear-based interaction of undo/redo is insufficient for the semantic process structure. So capturing user interactions is essential since researchers have shown that additional semantic information is necessary to adequately represent a user's analysis process [Pike and O'Connell 2009]. The third stage "encodes" the captured interactions in predefined formats. In previous research, different languages [Xiao et al. 2006; Jankun-Kelly 2007; Garg 2008] have been used to encode provenance. The "recover" stage aims to make sense of captured provenance and to get it ready for the "reuse" stage–reapplying the user's insights to a new data or domain. All five stages are essential for the success of promoting analytics provenance. In our VA system design, we aim to capture meaningful user interactions, encode the procedure, build the symbolic model for the analyst to examine, and then reuse the analysis process.

2.1.1. User Interactions in Analytic Provenance. The interactive manipulation of computational resources is part of the reasoning process. Interaction is always situated in the context of some problem or goal-directed activity. The interaction process is actually the process of inquiry [Pike and O'Connell 2009]. In the process of inquiry, the user's analytic contexts help to identify relevant concepts and link concepts into appropriate structures. The inquiry that users of visual analytics systems engage in is often pragmatic, in that useful insight only emerges out of the experience of manipulating information.

The analyst's sensemaking process can be divided into two loops: the information foraging loop (organizing information into some schema) [Pirolli and Card 1999] and the sensemaking loop (developing a mental model from the schema to support or contradict the claims) [Russell and Card 1993]. Such loops are reflected in two levels of interactions: low-level interactions (those between the user and the software interface) and high-level interactions (those between the user and the information space). The user's goal of lower-level interaction is often to change the representation with the intent to uncover patterns, relationships, trends or other features, mostly related with the information foraging loop of the reasoning process. In higher-level interaction, the user's goal is to generate understanding–to select, explore, reconfigure, encode, abstract/elaborate, filter and connect knowledge [Yi et al. 2007]. Higher-level interaction usually occurs during the sense making loop. These two types of interactions are pragmatic and sometimes integrated and even blurred [Pike and O'Connell 2009]. Visual analytics tools need to not only support users' navigation and data management, but more importantly, must also provide the two types of interactions so that the user can shift between the two kinds of loops without interruption.

2.1.2. The History-based Approach. In VA systems, a user's history is the record of user interactions during the analytical reasoning process. First, it is cognitively useful for users to be reminded of the sequence of interactions. Second, it helps to archive some useful states at different stages to help organize and report the final outcome. In Heer et al. [2008]'s survey, most of the published VA history mechanisms before 2008 were reviewed and categorized into a range of design decisions that arise when crafting an interactive history system. Based on integrating history management and undo/redo functionalities [Heer et al. 2008], these systems are grouped into model pairs such as "action model vs. state model", "stack model vs. branching model", and "local and global timeline model". In terms of visual representations, there are forms of the text abbreviation approach, thumbnail list, and branching layout. Users can interact with these mechanisms through navigating, editing, annotating, searching, filtering, and exporting.

We agree with Jankun-Kelly [2007] that history alone is not sufficient for analyzing the analytical process. Information foraging loops and sense making loops are usually intermixed with the reasoning process. Their related lower-level and higher-level interactions were mixed in the historical sequence. The record of such mixed actions or states can represent parts of the analytical reasoning process, but is not able to outline the logical structure of reasoning. Therefore, history based visualization is a useful memory tool but remains insufficient for interpreting analytical thinking. How may we represent the logical model for the analyst to rethink and reuse? In this paper, we employ a propagation-based parametric symbolic model to capture, represent, and manage the analytical reasoning process.

2.2. A Parametric Model Approach

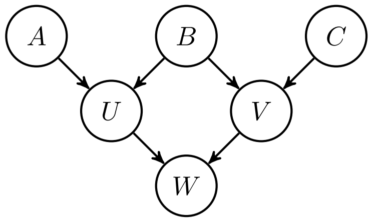
2.2.1. A Similar Metaphor in Parametric Design. According to Wright et al. [2006], the analytics process has a clear structure. We believe that it is useful to compare the structured analytics process with the metaphor which the designers have used in parametric design systems.

Parametric design was introduced in the Sketchpad system [Sutherland 1963]. Its first commercial implementations occurred in computer-aided design (CAD) systems in the 1980's. Since 2000, parametric design systems has become dominant in many design domains such as mechanical engineering (e.g. CATIA), industrial design (e.g. SolidWorks [Dassault Systèmes SolidWorks Corp. 2011]), and architecture (e.g. GenerativeComponents [Bentley Systems, Inc. 2010]). Parametric design systems model a design as a constrained collection of schemata (objects containing variables and constraints amongst the variables). Designers work in such parametric design systems at two essential levels: (1) defining schemata and constraints, and (2) searching within a schema collection for meaningful instances [Aish and Woodbury 2005]. By analogy, in visual analytics, the analyst may also work at two levels: (1) identifying the general analysis strategy, and (2) searching for meaningful instances according to the strategy.

Hoffman and Joan-Arinyo [2005] define parametric systems through a constraint solving approach. The graph-based approach uses a graph to represent the constraint problem. Nodes are objects, and edges are constraints among the objects. The solver analyzes the graph and formulates a solution. One category in the graph approach is propagation-based. In this method, objects and constraints become the variables and algebraic equations. Variables are vertices. Equations connecting input and output variables become the edges that point from the input variables to the output variables. The propagation methods proceed from upstream known information to downstream unknowns, solve these equations and evaluate variables basing on the orientations of the edges. However, the propagation methods may fail to find a solution if there is a loop along the edges (starting from one node V, following a sequence of edges that eventually loops back to V), and the iteration fails on convergence.

Propagation based systems are the most simple type of parametric systems [Aish and Woodbury 2005; Woodbury 2010]. It is easy for the end user to build a conceptual structure of a design, and its algorithms are efficient for the system to solve. In most systems, the directed graph is organized by the user to directly solve a problem (i.e. create a design). A well-formed design should avoid loops, which will guarantee that a solution is found. In CAD, the representation of the propagation-based system is often called a *symbolic model*.

2.2.2. Directed Acyclic Graph to Represent the Symbolic Model. The symbolic model can be represented by a directed acyclic graph (DAG) (Figure 1)–a graph containing nodes and directed edges. The edge links from a predecessor node (upstream) and points toward a successor node (downstream). A sequence of nodes that link from one to another in a single direction forms a path. The terms "downstream" and "upstream" refer to nodes that occur in at least one path of the given node. A well-formed model has no cyclic paths. A node is a schema, an object containing properties. Each property has an associated value. A value could be an atomic value (e.g., integer, string) or a complex object. A node having only one property (value) is a single-property node [Woodbury 2010]. In the following sections, most often a node is a single property node, and the *node's value* is the value of the single property of the node.



$$\begin{tikzcd}
A \arrow[dr] & & B \arrow[dl] \arrow[dr] & & C \arrow[dl] \\
& U \arrow[dr] & & V \arrow[dl] & \\
& & W & &
\end{tikzcd}$$

$$\begin{aligned} U &= \text{fun1}(A, B) \\ V &= \text{fun2}(B, C) \\ W &= \text{fun3}(U, V) \end{aligned}$$

Fig. 1. A directed acyclic graph of a symbolic model

Each node in this graph has one (and only one) unique name (or ID). The user and the system access the node by referring to its name. A (successor) node's property value can be derived from other (predecessors) nodes' properties. Such a value is assigned by a *constraint expression* and is computed by evaluating the constraint expression. It is called graph-dependent because the evaluated result depends on the values of its predecessor nodes. A constraint expression is a well-formed formula comprising objects, function calls, and operators. The constraint expression defines edges among nodes. The direction is derived from the predecessor to the successor node, indicating the successor node (property) holds a constraint expression that uses the predecessor node (property). We also conventionally say that data flows from predecessor nodes to successor nodes when the constraint expression of successor nodes are evaluated. In a number of cases a successor may have multiple predecessors, which will result in multiple edges all pointing toward the successor. The system ensures that the whole model is consistent. Whenever a node's value is changed, the system will evaluate all of the node's downstream nodes (i.e., evaluating constraint expressions and populate results). A property may hold an explicit value (where it does not depend on other properties). Such a property is then known as *graph-independent*. Otherwise, the property is called *graph-dependent*, which means that its value depends on other properties [Woodbury 2010].

Depending on the values of its properties, a node can be a source node, sink node, or internal node. A source node is a node with no graph dependent properties. Data only flows out of such a node to its successor nodes. A sink node is a node with no successor; its properties are not used in other nodes' (properties') constraint expressions. Data flows into the sink node but no data flows out. An internal node is a node in the middle of the graph, and it is both a predecessor and successor. In Figure 1, A, B, and C are source nodes; U and V are internal nodes; W is a sink node.

Different parametric design systems use different parametric models. Explicitly visualizing the parametric model helps the designer understand the logical structure of design decisions. Most major parametric design applications provide such a visualization. In GenerativeComponents [Bentley Systems, Inc. 2010], the parametric model is visualized by a DAG called *symbolic model*. In Grasshopper [McNeel 2010], which provides generative modeling capability to the NURBS modeling application Rhino, the visualization is called *history graph*. SolidWorks' *feature manager*, CATIA's *specification tree*, and Pro/E's *model tree*, conflate two types of information from one tree-like structure visualization: (1) the design's feature hierarchy, and (2) the creation history of features. Rolling up and down along the tree allows the designer to revisit and modify different parts of the design history.

2.2.3. Other Models.

Many process flow and work flow based systems use DAG to represent the flow [Johnston et al. 2004; Dennis 1974]. Data tuples flow through an acyclic directed graph of processing operations. In the DAG of a data flow system, the nodes contain processing operations such as arithmetic or comparison operations, while the edges are pointing to the direction of data flow from one operation to another. In our symbolic model, nodes are objects with values. The edges are constructed by the node values (constraint expression) which represent the dependency among objects.

The Visual Data Flow Programming Languages (VPL) [Hils 1992] support users writing programs by graphically manipulating elements instead of textual programming. Some systems, such as Prograph [Cox 1990], provide the programmer icons or other graphical representations to write a program. The program is then debugged and executed in the same visual environment.

VisTrails applies the data flow idea in the context of scientific visualization. It employs a DAG to present the data flow of visualization, in which nodes are data process functions and edges indicate the directions of visualization data flow. It also uses a tree graph to record and visualize the evolution of the data flow. Combining the data flow and history management, VisTrails maintains a detailed provenance of the scientific visualization exploration process. Scientists can easily explore different parameter settings for a data flow as well as navigating through different versions of data flow.

In many spreadsheet systems, although it is not explicitly shown to users, the DAG is the underlying representation of the relations among spreadsheet cells. One cell may stores a formula that uses other cells' values, which is very similar to the concept of nodes in the symbolic model.

3. THE SYMBOLIC MODEL IN CZSAW

In developing CZSaw, we chose to use the parametric propagation approach to represent the analytical reasoning process. To be consistent, we use the name *symbolic model* for the DAG, and the name *dependency graph* to label the DAG's visualization in CZSaw. The propagation mechanism is the computational engine that manipulates the data and visualizations.

3.1. Improvements upon Previous Research

The initial concept of the symbolic model (dependency graph) was introduced in an earlier paper[Kadivar et al. 2009]. After two years' improvement, for the first time in this paper, we systematically discuss in detail the symbolic model, how it is constructed, how it works and handles error, how a user interacts with it, how the model interact with other CZSaw components, and how to use this model to support visual analytic provenance.

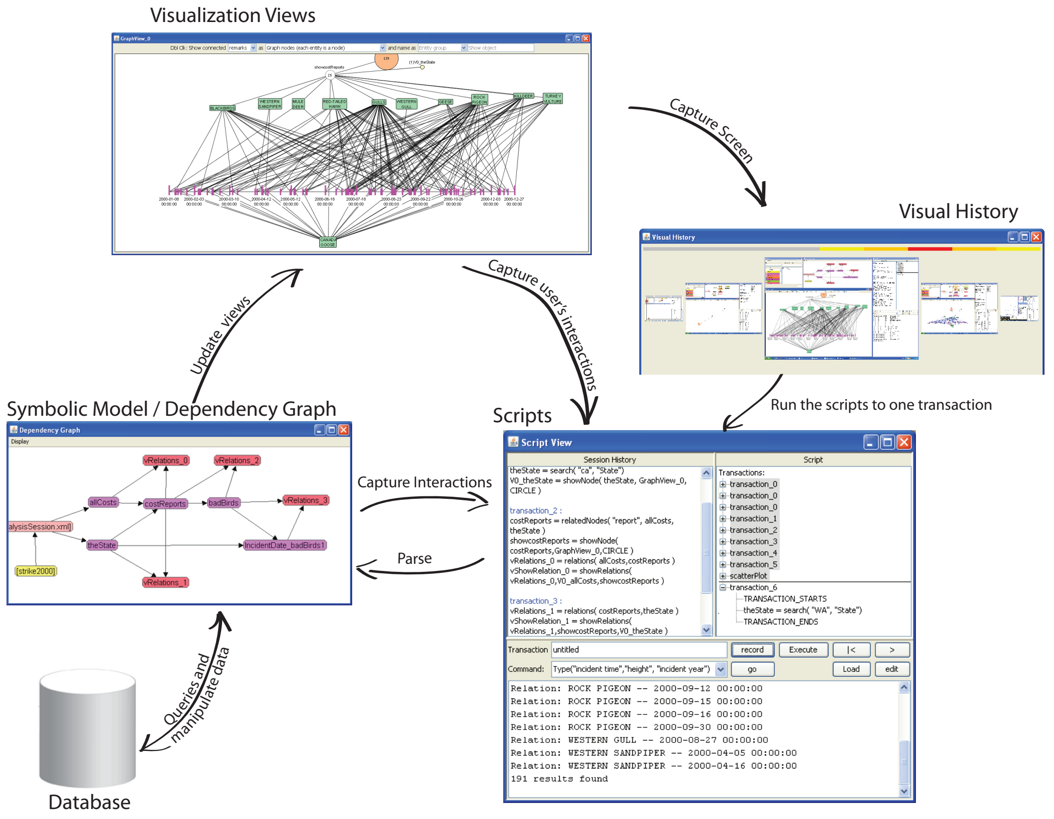
The innovative contributions of this paper largely focus on this systematic discussion of the construction and usefulness of the symbolic model approach to support VA provenance. Our 2009 paper [Kadivar et al. 2009] was an overview paper–generally written to introduce all aspects of CZSaw. At that time, we only had a few script functions to demonstrate basic text analysis and replay capability. Subsequently, through working on several visual analytical tasks including VAST challenges, we have significantly enriched the script commands, studied the possibility of extending the symbolic model, and enabled an additional analysis method – through interaction with the dependency graph. We believe these developments subsequent to the 2009 paper, warrant a more detailed report on the scripts and model and on how they help with the visual analytics provenance in different tasks and domains.

3.2. Propagation-based Model in the Structure of CZSaw

Figure 2 shows the current general structure of CZSaw. The database element uses the MySQL database management system to store text records, entities, and entity–to–record relations. Varied views visualize the data and provide the interface for analysts to manipulate the data. The visual history [Kadivar 2011] provides a visual representation of chronological analysis states that allows the analyst to visually inspect the analysis process. CZSaw's data views include the *document view* (read content of documents), *list view* (a list of entities are placed in a list), *hybrid view*, and *semantic zoom view*[Dunsmuir et al. 2012].

The dependency graph is the DAG visualization of the symbolic model. The symbolic model is built based on statements in the scripts and directly controls the data and visualization within data views, and script statements are generated from analyst interactions.

3.2.1. Interaction and Data Flow among CZSaw Components. Figure 2 shows the data flow from user interactions to the system's reaction. The user interacts with CZSaw through all four major components: visualization views, the dependency graph (symbolic model), scripts, and the visual history. An user's interaction is converted to a sequence of script statements called a *transaction*, and parsed to update the symbolic model (and thus the dependency graph) [Kadivar et al. 2009]. Then the symbolic model propagates to update related nodes. If a node contains a constraint expression, the



Session History

```
theState = search( "ca", "State")
V0_theState = showNode( theState, GraphView_0,
CIRCLE )

transaction_2 :
costReports = relatedNodes( "report", allCosts,
theState )
showcostReports = showNode(
costReports,GraphView_0,CIRCLE )
vRelations_0 = relations( allCosts,costReports )
vShowRelation_0 = showRelations(
vRelations_0,V0_allCosts,showcostReports )

transaction_3 :
vRelations_1 = relations( costReports,theState )
vShowRelation_1 = showRelations(
vRelations_1,showcostReports,V0_theState )
```

Script

```
Transactions:
+ transaction_0
+ transaction_0
+ transaction_0
+ transaction_1
+ transaction_2
+ transaction_3
+ transaction_4
+ transaction_5
+ scatterPlot
- transaction_6
    TRANSACTION_STARTS
    theState = search( "WA", "State")
    TRANSACTION_ENDS
```

Transaction: untitled

Command: Type("incident time","height", "incident year")

```
Relation: ROCK PIGEON -- 2000-09-12 00:00:00
Relation: ROCK PIGEON -- 2000-09-15 00:00:00
Relation: ROCK PIGEON -- 2000-09-16 00:00:00
Relation: ROCK PIGEON -- 2000-09-30 00:00:00
Relation: WESTERN GULL -- 2000-08-27 00:00:00
Relation: WESTERN SANDPIPER -- 2000-04-05 00:00:00
Relation: WESTERN SANDPIPER -- 2000-04-16 00:00:00
191 results found
```

Fig. 2. The general structure and dataflow of CZSaw

constraint expression will be evaluated to query and manipulate data. The evaluated value will be reflected in the visualizations in data views. From the user's perspective, a new visualization is created, or an existing visualization is updated. The updated CZSaw system status, including both the dependency graph and data views, is captured into screenshots and saved in the visual history.

While analyzing the data, the analyst usually spends most of the time in visualization views. The user can click or right click on a visualization and choose an analytic function through a pop-up dialog window. Depending on the function, the system generates one transaction with one or several lines of script and create or update one or several nodes in the symbolic model. The clicked visualization (`showNode` or `showRelation` objects in section 3.4.1) will be part of the parameters in the constraint expression of newly created nodes if necessary. Users with programming skills can also manually code new transactions in the script view.

It is also possible to interact with the dependency graph. The analyst can query and change node values in the graph view. Such interactions will also be recorded into scripts which are executed to update the dependency graph and data views.

The visual history component allows the analyst to drive the CZSaw system to a certain history status by re-running the scripts [Kadivar 2011]. The executed scripts construct the symbolic model, which update the data views to the corresponding state. To avoid generating a history of history, interactions within the history view do not create new transactions or alter the scripts.

3.3. Basic Objects in CZSaw

CZSaw was initially inspired by Jigsaw [Stasko et al. 2007] and was designed to deal with similar types of data – collections of unstructured text documents. These text records are pre-processed with entities extracted. An entity could be a person, location, date time, organization, etc. In essence, a text record can be seen as a collection of entities. The text record defines relations among entities. Text records are related to entities via a direct inclusion relationship (the document contains the entity in its text). Entities are also related if they are contained in the same document (co-citation [Small 1973]). Documents are related if they contain the same entity (bibliographic coupling [Kessler 1963]). Based on such relations, we define three basic types of object in CZSaw: `entity`, `relation`, and `record`.

— `Entity`: An `entity` in CZSaw is a tuple containing {(int)unique ID, (string) entity type, value, (optional string) entity name}. The value can be a string, a number, or a date. Entity type can be one of {person, location, date, organization, or an user defined thing, etc.}. Two different entities may have the same type and value. For convenient purpose, an entity may carry a name for reference if it's value is too complicated to see and remember. If the name is omitted, the entity's value will be used for display in views.

— `Record`: A record is a tuple of {(int) unique ID, (string) record name, (list) entities, (optional string) text}. While dealing with relational databases tables or spreadsheets, rows in the table are converted into records [IEEE VAST 2008; 2009]. Objects in one row make the entities list in the record. While dealing with text documents [IEEE VAST 2010], a record can be extended by containing a text string. The entities list contains all entities extracted from the document. The string is the text content of the document. Through CZsaw's document view, an analyst can read the string of the records to realize the real meaning carried by the document and relations among entities. Since this concept is originated from text documents with entities extracted, a record is also called a report or a document depends on the task while we use CZsaw.

— `Relation`: A `relation` object defines a relation between entities, an entity and a record, or between records. Normally it is a pair of {{entity, record}, {entity, entity}, or {record, record}}. These relations are defined by direct inclusion {entity, record}, co-citation {entity, entity}, and bibliographic coupling {record, record}. When necessary, relations can be explicitly defined and contain directions and strengthes, to make them triples of {object A, object B, number Strength}.

Like two different persons may have the exactly same name, theoretically two different entities (with different ID) may have the same type and value. These unique ids are automatically generated at the time of importing the data into CZsaw's MySQL database. If entities are extracted automatically from text document records, typically there will not exist two or more different entities with the same type and value. Using CZsaw's entity refinement commends, the user can split one entity into two different entities, or merge two entities as one entity.

3.4. Script Language in CZSaw

We built a script language containing commands to perform simple analytics tasks in CZsaw. CZsaw allows an analyst to operate on a group of entities by encapsulating the output of an operation (it may contain one or several objects) into a variable in a CZSaw script.

3.4.1. CZSaw Script Objects. CZsaw script can directly access basic objects of entity, record, and relation. Other than data, CZsaw script also can manipulate visualizations.

To give the analyst the maximum control, data and their visual representations are separated in CZsaw. Basic CZSaw script objects include:

— `View`: an object representing one visualization window. CZSaw can have multiple visualizations; each has its own window referred to as a CZSaw `view`, and has properties such as its location, size, and default layout.
— `EntityVariable`: an object representing a set of entities of arbitrary type (may be empty). It could also include records. Records in the script are treated as a kind of entity with the type "record". Analysts often need to deal with sets of entities. For example, to find entities related to a given entity (or entities), the result will be a set of entities, which can then be referenced by a named object `EntityVariable` in a script statement. An `EntityVariable` can contain zero to many entities.
— `RelationVariable`: an object representing a set of relations (may be empty).
— `ShowNode`: an object representing one visualization of one `entityVariable` in one `view`. We want to separate the content (entities in an `entityVariable`) and the display format of the content. If needed, we may visualize the same content multiple times. For example, the same set of entities can be visualized either in different views (windows) or in the same view window, but with a number of differing appearances (such as one node only, as a group node, or a list). We can then use different `showNode` objects to manipulate these different visualizations. A `showNode`'s properties contains the visualizing `entityVariable`, the `view` object where the `entityVariable` will be visualized, visualizing types, and show/hide status. The visualizing type property controls the form of the visualization. The `entityVariable` could be visualized as scattered nodes (entities as separated nodes), one single node for the group of entities, a list, or one node with varied size reflecting number of entities in the `entityVariable`. While visualizing in scattered nodes, to avoid showing one entity multiple times in one view, the entity will be only shown as one node in one view, no mater how many times it is included in different `showNodes`. But when visualized as grouped nodes or list, one `entityVariable` could be displayed multiple times in one window through packed into multiple `showNodes`.
— `ShowRelation`: an object representing one visualization of one `relationVariable` corresponding to two `showNodes`. Its basic properties contain the `relationVariable`, two `showNodes`, and show/hide status. The two `showNodes` should be in one `view`, and may possible be the same `showNode`. The visualization form of a `showRelation` relies on the ShowNodes' visualization type. Relations could be visualized as multiple edges with one end connecting to nodes of one showNode, and the other end connecting to the nodes in the other. If one `showNode` in displayed as a grouped form, then all the edges will be pointing to the `showNode`. If two `showNodes` are all in grouped form, then the `relationVariable` will be a single edge connecting the two `showNodes`. For maximum flexibility, the `relationVariable` does not necessarily contain all (or only/any) relations of the entities in the `entityVariables` of the two `showNodes`.

With `showNode` and `showRelation`, the user can separate the visualization from the data and have much more flexibility to manipulate and design the visualization.

3.4.2. Translate Analysis Process into CZSaw Script. In Jigsaw, a typical process involves searching for an entity, reading its related documents, and continuing to get related entities from one document [Stasko et al. 2007]. Iterating this process produces a graph of related entities and documents. The process can be described as:

(1) Searching for an entity of given value and entity type.
(2) Finding all documents related to this entity.
(3) Reading one document in the document view.
(4) (Clicking on the document to) view all related entities.

We can translate the above process into several lines of script statements. For example, the first two steps can be translated into the following three lines of commands:

```
EntityVariable Entity0 = search(‘‘person’’, ‘‘eve’’);
            // find any entity with the type ‘‘person’’ and value ‘‘eve’’
EntityVariable  Doc0 = relatedNodes(Entity0,‘‘record’’);
            //find all documents related to people in ‘‘Entities0’’
RelationVariable Rel0 = relations(Entity0, Doc0);
        //find all relations between entities in ‘‘Entity0’’ and ‘‘Doc0’’
```

In the above commands, the function `search` returns all entities of type “person” which match the value “eve” from the data source. The result is assigned to the `entityVariable Entities0`. The user may use wildcard “%” in the parameters to get entities that match partial values or types. The function `relatedNodes` identifies all records that are related to (i.e. contain) any entity within the given argument `Entity0`. `Doc0`’s value thus is a set, possibly empty, of “records”.

The script is automatically generated as the analyst interacts with data views and the dependency graph. Each user interaction is translated into a block of one or more script commands that is termed a *transaction*. The analyst can proceed either interacting with views or by directly typing script commands into a dialog box. The analyst can also manually edit scripts using any text editor, which provides analysts with additional programming power to help solve complex problems. The user can save transactions into one text script file, which can be reloaded for future replay [Kadivar et al. 2009].Multiple versions of the scripts can be saved into difference files. However, currently CZsaw can not manage multiple versions of the scripts at once.

In general we want the scripting language to perform operations meaningful to the analysis process. Gotz and Zhou [2008] proposed capturing action-based subtasks for insight provenance. CZSaw captures user interactions at the action level, and records actions that change the state (visual or data) of the system. Some example interactions are, for example, the analyst searching for an entity, or finding related entities and display them. Micro interactions, such as moving the mouse and keying in characters, are not recorded. Some script commands, such as `search` and `relatedNodes` describe system actions on the data structures. Other commands describe system actions that directly result in data view changes such as `showNodes` and `hide`. The detailed demonstration of converting interactions into transactions can be seen at section 4.1.

3.4.3. CZSaw Script Functions. Kadivar et al. [2009] introduced the basic method of converting user interactions into transactions. By initially following an interaction pattern similar to that of Jigsaw, we have defined several basic commands (such as “search”, “related”) in order to search for an entity and find related documents and entities. Subsequent to the work reported in that paper, we undertook an extensive study on applying CZSaw to a variety of VA tasks (e.g. VAST challenges), and designed scripts with data types and functions meaningful to visual analytics, especially with entity-relation problems. For example, while working on the VAST 2008 cellphone challenge, with a group of entities, we wanted to search for an entity related to most of the entities in the group. To accomplish this, we created an interaction (and corresponding script function) called `commandRelated`. The interaction was further enriched in the VAST 2009 Flitter network challenge by adding a lower and upper limit on the range. These entity refinement commands were created while working on the VAST 2010 challenge.

We category CZSaw script commands as follows:

— *Data view commands:* Control visualization states, such as show/hide, layout, and aggregation levels of the visualization (e.g., showing a set of entities as scattered nodes, in a list, or grouped as a single node).
— *Data query commands:* Query and filter entities or relations from CZSaw's database. Examples include searching for entities by value comparison, searching for entities related to a set of entities, or establishing relations between two entity sets. In CZSaw, analysts do not need to deal with a very crowded graph containing all of the entities. They can work selectively on a set of entities.
— *Entity refinement commands:* Extract, merge, edit, and link entities. CZSaw relies on entities and relations to generate visualizations. Text documents involved in real world problems are usually messy and contain inconsistencies and errors. Automated entity extraction is often incomplete. Thus, we provide entity management commands (extract new entities, merge entities, link entities to records, and unlink entities from records) that allow analysts to refine entities on the fly within the analysis. The analyst can manually refine entities while reading a document or working within other views. These entity refinement commands can alter the data in the source database, which triggers the value change of the *data source node* in the symbolic model. The propagation mechanism is then invoked to update content and layout in views to reflect these changes.
— *Data compute commands:* Such commands compute `EntityVariables` and `RelationVariables`. Examples include Boolean operations, such as union and intersection. For example, the intersection of two `EntityVariables` returns an `EntityVariable` that contains entities that are in both input `EntityVariables`.

For easy programming, in the current version, CZSaw embeds BeanShell [Niemeyer 2005] as its scripting engine due to its java-like syntax. BeanShell also supports common scripting conveniences, such as loose types, commands, and method closures. This BeanShell engine allows CZSaw scripts to easily access CZSaw internal objects and APIs, which also enables CZSaw developers to make quick tests during the development process.

A list of general commands are listed in the appendix A.1. This list is not a full list of all CZsaw commands. Many of these commands are designed based on analysis needs. While working on new visual analytics problems, we kept on creating new commands to full fill the task.

3.5. Map the CZSaw Scripts to the Symbolic Model

CZSaw creates the symbolic model based on parsing and executing scripts. Executing scripts may create new nodes and links in this model or may update values of existing nodes. Most variables in CZSaw are single property nodes, which means their only property is their value. CZSaw visualizes this symbolic model, which we call the dependency graph, as a DAG.

Some commands assign simple values to or directly use atomic values for a variable. Such commands will generate a source node. For example, the following command creates a source node `personName` with the value of string "David".

```
string personName  =  ‘‘David’’;
```

In CZSaw's symbolic model, source data is represented by a special source node, which we call it a *data source node*. The node is named by the data source, e.g., the database's name. We can have multiple data source nodes in the model, which means that the VA system should be able to query information from different data sources. Depending on the interaction, sometimes the scripts may access internal CZSaw objects. In this case the result is directly generated from the data source, which creates an edge linking the data source node to the variable containing the interaction result. For example, the search command does not explicitly involve a data source node, but since the search command requires the data source to find matching entities, this line will create a node `Entity0` in the model, with an edge linking from the data source node to this `Entity0` node. Mathematically the data source node is no different from a regular source node. But such a node is especially useful when dealing with dynamic data, i.e. streaming data. A change in one data source means the data source node's value is changed, which will propagate through its down-stream nodes and reflect the data change in visualizations.

```
entityVariable Entity0 = search(‘‘person’’, ‘‘Eve’’);
```

Some script commands create variables that rely on other variables. Such variable will have constraint expressions as their values. For example, the `entityVariable Doc` generated from the following command will create a single property successor node `Doc0` in the graph, with the script command `relatedNodes(Entity0,‘‘record’’)` as its constraint expression. Since the constraint expression involves the node `Entity0`, in the graph there is a directed edge pointing from `Entity0` to `Doc0`.

```
entityVariable Doc0 =relatedNodes(Entity0,‘‘record’’);
```

The above two lines of scripts will build the following symbolic model (Figure 3). The data source used is an XML file “bible.xml”.

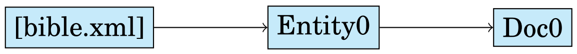


Fig. 3. A simple symbolic model

CZSaw's symbolic model is a propagation based model. Once a node's value is changed, all of its successor nodes will be automatically evaluated to update the system state. The simplest way to execute the propagation through the graph is to compute the children nodes recursively. However, this method is inefficient; if one node is a child of multiple nodes, then the node will be evaluated multiple times.

To enhance the efficiency, we use the topological order [Kahn 1962] to sort nodes in the model. With this ordering, all nodes in a DAG are sorted in a linear ordering such that, for every link `uv` that points from `u` to `v`, the node `u` comes before `v` in the ordering. After all the nodes are in a linear order, when a node changes its value, only its following nodes need to be re-evaluated. Data source nodes are always at the top of the order. Data added/removed/edited from a data source indicates that the value of the data source node has been changed, which will cause all its downstream nodes to be re-evaluated.

3.6. Interact with the Dependency Graph

CZsaw visualizes the symbolic model using a node-link graph and calls it dependency graph. For better visibility, different colors are used to distinguish the nodes' types (Figure 7). Nodes can be categorized into: data source node, source nodes with atomic values, view nodes, `entityVariable` nodes, `relationVariable` nodes, `showNodes`, and `showRelation` nodes. The analyst can easily re-arrange the graph layout by dragging and dropping nodes.

Furthermore, the analyst can directly interact within the dependency graph to:

— Read a node's value, including its constraint expression and evaluated results. Through a single right click, the user can open a pop-up dialog window with the

node's value in an input box. If the node has a constraint expression, all evaluated results will be displayed under the value box.
— Change a node's value. In the above pop-up dialog window, the user can directly edit the value resulting in the generation of a new value-assignment script statement which is added to the script. The new script will then be parsed to update the node's value and propagate it through the symbolic model.
— Execute analysis directly in the dependency graph. Available operations depend on the node's type. The analyst can right click on a node to pop up a dialog window, then select an analytic function. The selected node will become a parameter in the selected function. The user may have to select/input other parameters for the function. Depending on the task, this interaction will generate one transaction containing one or several script statements, and may create one or several new nodes in the symbolic model.

Simply viewing the value will not create a new transaction. If the interaction changes any node's value or creates new nodes, a new transaction will be created and then parsed to update the symbolic model. If a source node's value is changed, the structure of the symbolic model will not change. But if a successor node's constraint expression is changed, the model's structure may be changed since the edges are defined by the constraint expression. Old edges may be removed and new edges may be added.

3.6.1. Reuse the Model. Assigning different values to some nodes in the symbolic model will drive CZSaw to a different state. If we only change values of graph independent properties, the topological structure of the graph will be preserved, but the evaluated values of successor nodes will be updated. Here we say that the user has created a model to solve an analytics problem; additionally, the user may now use that model to solve related problems and create new visualizations by *reusing* this model with different parameter values. The user may also re-define the constraint expression of a node. This will change the edge pointing to the node.

In the symbolic model of Figure 3, we can update the value of `Entity0` by assigning it a new value (whether through direct scripting, interaction in the dependency graph, or in a data view):

```
entityVariable Entity0 = search(''person'', ''Adam'');
```

By doing so, `Entity0`'s value changes to contain all people with name "Adam". As a consequence, `Doc0` will automatically be re-evaluated to search for documents related to any person with "Adam". Once the value of an `entityVariable` is changed, then its visualization (the `showNode`) will be updated. The above operation can be easily done through the interaction of changing a node's value in the dependency graph.

3.7. Error Handling

Two types of errors may occur while dealing with the symbolic model. One occurs at the time of constructing the model, the other occurs when changes are propagated through the model.

The first error happens while constructing the symbolic model through a user's interaction. S/he can reuse an existing variable by assigning it with a new constraint expression that involves variable(s) created later. This might cause cycles in the symbolic model. A cyclic graph is possible to compute but its result is hard to predict. In some cases the cyclic graph will be convergent and the final result will reach a stable limit. But it is also possible that the result is divergent or goes into oscillation. To avoid generating cycles, after parsing the interaction and before updating the symbolic model, the system checks the scripts to see if any of the commands may cause such a problem. If it does cause cycles, currently the system generates an alert message to tell the user that such an interaction is invalid. The interaction will be discarded without recording it into the script.

The second type of error occurs during propagation. The user can change values of nodes in the model. As a result, constraint expressions in downstream nodes may become invalid (e.g. an attempt to divide by zero) with the new value of its upstream nodes. For such nodes and all of their downstream nodes, the model will not evaluate them, but simply assign them a special flag. In such a case, the model is equivalent to a sub-model by removing all of those invalid nodes. In the dependency graph, we change the appearance (add a solid read boundary) of the nodes to provide a hint to the analyst. These nodes will be ignored in data views since they do not contain valid data.

It is also possible for the evaluated value of an `entityVariable` (or `relationVariable`) node to be an empty set. For example, searching for a value that does not exist in the data. In such a case, the `entityVariable`'s value is an empty set. But it is still recognized as a valid value in the system and can be passed to its downstream nodes to compute. In data views, whether such a node may or may not be shown depends on the visualization method.

4. USE CASES

In this section we describe use of the model to analyze three IEEE VAST Challenges. The VAST challenges provided realistic datasets and problem scenarios for researchers to evaluate their VA systems and problem solving approaches [Scholtz et al. 2012]. Through solving these challenges, we were able to enrich the CZsaw script, explore the capabilities and limitations of the parametric mechanism, and suggest directions for further development. The three challenges are:

2008 mini challenge MC3:. cell phone calls (social network) problem to detect possible replacement of decommissioned phones.
2009 mini challenge MC2:. social network and geo-spatial problem to identify people in a criminal network.
2010 mini challenge MC1:. text records problem is to make sense of firearms dealing activities from a set of text documents.

At following sections of use cases, we introduce the problem scenario and dataset, plan the analysis strategy, execute the strategy either through user's interaction in CZsaw's GUI and data views or manually programming in CZsaw script, and then explore the different usages of the parametric model.

4.1. Use Case A: 2008 mini challenge MC3

An analyst may use the same strategy to solve many similar problems. This VAST challenge can be divided into serval similar small problems. The analyst at first execute a planned solution for one problem through a series of interactions, and then reuse the solution to solve other similar problems.

4.1.1. Identifying Social Network Changes From Phone Call Records. The goal here was to discover changes in the Catalano social network over a period of time. The contest provides a data set of cell phone call records over ten days. These records (rows in a spreadsheet table) reveal critical information about the Catalano social network structure. Each data set record has the following values:

— From: calling phone ID
— To: receiving phone ID
— Date&time: accurate time of the call in the format of yyyymmdd hhmm

— Duration: length of the call in seconds
— Location: location of tower originating the call

This challenge asked two questions. The first is to identify the Catalano social network. The second is to characterize the change in this Catalano social network structure over the 10 day period.

First we convert the data into CZSaw objects. Phone, date&time, and tower are CZSaw entities. A data set record defines CZSaw relation that connects phone to phone, phone to tower, and phone to date&time.

While working on the problem, we noticed that some frequently used phone numbers suddenly disappeared in later days. Since the human communication network should be persistent, we suspected that phone numbers which disappeared were replaced by new numbers. The substitutes should contact a similar group of phones and be active in locations (towers) similar to the original phones. The logic of this solution can be executed by the following user interactions:

4.1.2. Step One–Search for a suspicious phone. The first step is to search for and display one phone that needs to be checked. To start, from CZSaw's dropdown menu, the analyst creates a new hybrid view window. This interaction creates one transaction with one line:

```
Transaction_1{
  view gv0 = newView( ‘‘GraphView’’ );
}
```

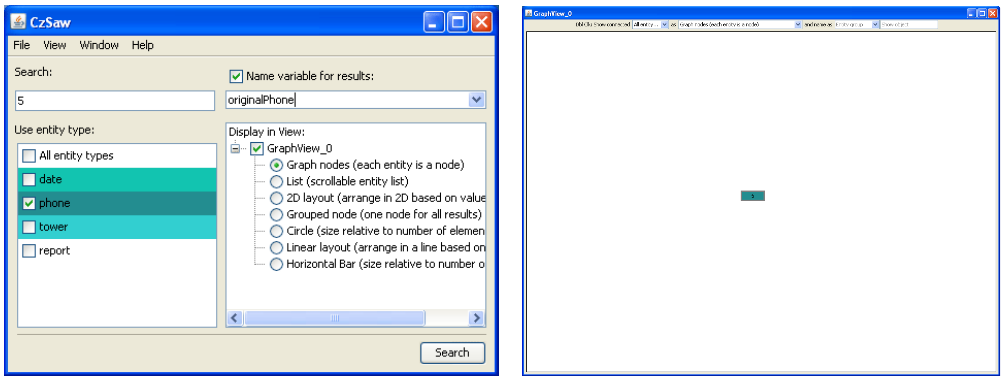
This line created a new `View` and assigned this CZSaw view object to a user-named variable `gv0`. Data is visualized in this view as a node-link graph (another option is to display data in a list–"ListView"). Later in the script, the variable `gv0` is used to refer to this view window.

After checking all Catalano(phone 200)'s connections, the analyst found that phone 5 may belong to one of the key persons in the social network. Other suspicious phones are 1, 2, 3, 97, and 137. From CZSaw's search panel, the analyst searches for phone 5 and displays it in the view `gv0`. This interaction is recorded into one transaction with two lines of scripts:

```
Transaction_2{
  entityVariable  originalPhone = search(5, ‘‘phone’’);
  showNode  showOriginalPhone = showNodes(originalPhone, gv0, ‘‘GRAPH’’);
}
```

The first line searches for an entity of type "phone" and value "5" (or several entities if there is more than one phone with value 5) and assigns this entity (entities) to a variable `originalPhone`. The analyst can use this variable later in the script to access entities in it. The second line displays entities of `originalPhone` in the graph view window `gv0`. The parameter "GRAPH" tells the view to display entities in the view as distributed nodes (one node per entity). The visualization of `originalPhone` is stored in the variable `showOriginalPhone`. The force-based algorithms [Fruchterman and Reingold 1991] are used to lay out these nodes by default. `showOriginalPhone` is a `showNode` object that represents the visualization of the entities in `originalPhone`.

4.1.3. Step Two–Find Related Phones. Next, the analyst wants to find all phones related to phone 5. S/he can interact with the showNode object `showOriginalPhone` (the phone 5 node in the view `gv0`) to discover related phones (`relatedPhone` in the following scripts) by right clicking on the phone node and accessing the popup menu (left part of Figure 5). This interaction creates four lines of script:



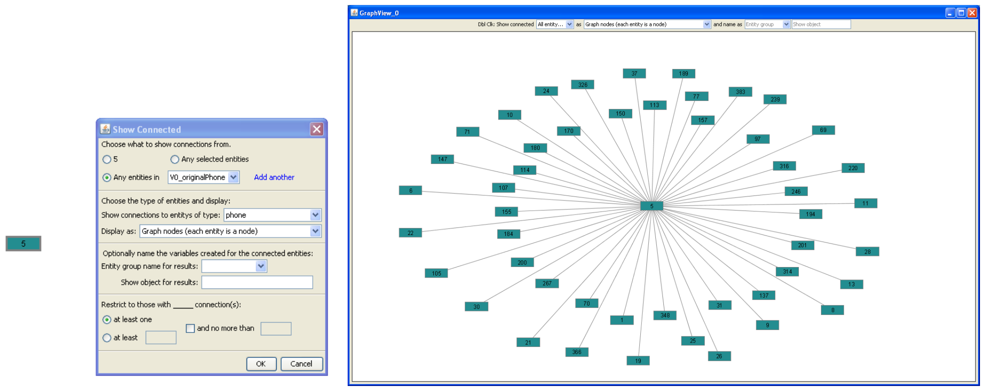
Search: 5

Name variable for results: [✓] originalPhone

Use entity type:
- [ ] All entity types
- [ ] date
- [✓] phone
- [ ] tower
- [ ] report

Display in View:
- [✓] GraphView_0
  - [✓ Graph nodes (each entity is a node)]
  - List (scrollable entity list)
  - 2D layout (arrange in 2D based on value
  - Grouped node (one node for all results)
  - Circle (size relative to number of elemen
  - Linear layout (arrange in a line based on
  - Horizontal Bar (size relative to number o

Fig. 4. Left: Search for phone 5. Right: Phone 5 is displayed in the graph view



Choose what to show connections from.
5 Any selected entities
[✓ Any entities in] V0_originalPhone

Choose the type of entities and display:
Show connections to entitys of type: phone
Display as: Graph nodes (each entity is a node)

Optionally name the variables created for the connected entities:
Entity group name for results:
Show object for results:

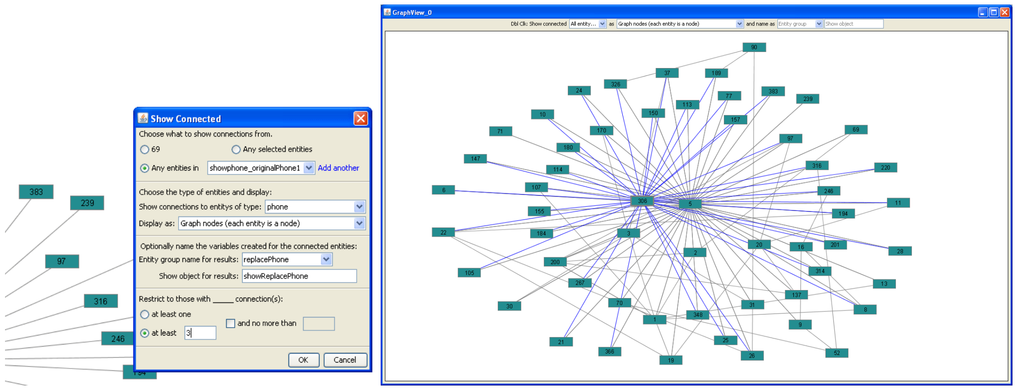
Restrict to those with _____ connection(s):
[✓ at least one]
at least
[ ] and no more than

Fig. 5. Left: Get related phones. Right: Related phones are automatically placed around phone 5

```
Transaction_3{
  relatedPhone = relatedNodes(originalPhone, ‘‘phone’’);
  showNode showRelatedPhone = showNodes(relatedPhone, gv0, ‘‘GRAPH’’);
  relationVariable relPhoneRel = relations(originalPhone, relatedPhone);
  showRelation showRelPhoneRel = showRelations(relPhoneRel,
                                    showOriginalPhone, showRelatedPhone);
}
```

The first line finds all entities related to entities in `originalPhone` and assigns them to a user-named variable `relatedPhone`. The second line visualizes (`showRelatedPhone`) these phones in the view `gv0`. The third line gets the relations of entities in `originalPhone` and `relatedPhone`. The fourth line displays the relationVariable `relPhoneRel` between the two showNode objects, which are edges among phones in the view. Phone 5 and its related phones are visualized in a node-link graph as the right part of Figure 5.

4.1.4. Step Three–Detect The Replacement. In the third step, the analyst wants to discover if there are any phones that relate to several (e.g. ≥ 3) phones in `relatedPhone` (Figure 6). Through right clicking on any related phone nodes in the view and accessing the popup menu (left part of Figure 6), the interaction generates the following transaction:



Show Connected

Choose what to show connections from.

( ) 69 ( ) Any selected entities

(•) Any entities in showphone_originalPhone1 Add another

Choose the type of entities and display:

Show connections to entitys of type: phone

Display as: Graph nodes (each entity is a node)

Optionally name the variables created for the connected entities:

Entity group name for results: replacePhone

Show object for results: showReplacePhone

Restrict to those with _____ connection(s):

( ) at least one

(•) at least 3 [ ] and no more than

Fig. 6. Left: The interaction of searching for common related phones. Right: Phones connect to three or more phones in *relatedPhone*.

```
Transaction_4{
  entityVariable replacePhone = commonRelated(relatedPhone, ‘‘phone’’, 3);
  showNode showReplacePhone = showNodes(replacePhone, gv0, ‘‘GRAPH’’);
  relationVariable relPhoneRep = relations(relatedPhone, replacePhone);
  showRelation showRelPhoneRep = showRelations(relPhoneRep,
                                   showRelatedPhone, showReplacePhone);
}
```

The first line searches for any phone that is related to at least three phones in `relatedPhone`. The rest of the lines visualize these phones and connections.

Figure 6 shows there are several phones having three or more connections. Among these, phone 306 connects to the most phones in `relatedPhone`. Continuing the investigation by checking the dates of calls (not discussed in this paper due to length limit), we can find that phone 306 was put into use right after phone 5 stopped. Thus, the analyst hypothesizes that phone 306 replaced phone 5.

The same procedure can be repeated to analyze other phones. But certainly simple repetition of the procedure will be time consuming and likely error-prone. Instead, the CZSaw symbolic model (dependency graph) built from the script allows the analyst to reuse the logic contained in this analysis process, a significant advantage of our method.

4.1.5. The Symbolic Model. The above phone search scripts generate the symbolic model shown in Figure 7 where `cellPhone.xml` is the data source node. The node `originalPhone` contains the searched result phone 5. Its visualization in view `gv0` is represented by `showOriginalPhone`.

To find the replacement of phone 1, the analyst can repeat the search interaction in Figure 4 and assign the search results into the same variable `originalPhone` from the variable list drop down menu in the search panel. S/he can also access the dependency graph to change the value of `originalPhone` or directly create a new script line to re-assign the value. As the value of the node `originalPhone` changes, the propagation mechanism evaluates its successors starting with its direct successor nodes `relatedPhone`, `showOriginalPhone`, and then continues to evaluate in a topological order sequence to update all successor nodes in this model, which will finally update the visualizations. The result is the new graph of phone 1 shown in Figure 8, which suggests phone 309 may replace phone 1. In this particular challenge question, the an-

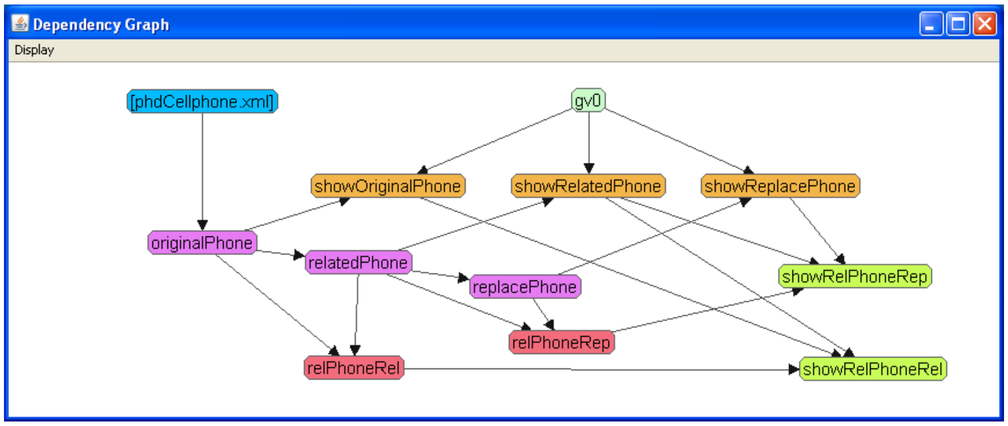


Fig. 7. The dependency graph created from the scripts of section 4.1. Colors for node types: blue - root node, purple - entityVariables, red - relationVariables, orange - showNode, green - showRelation, light green (gv0) - view

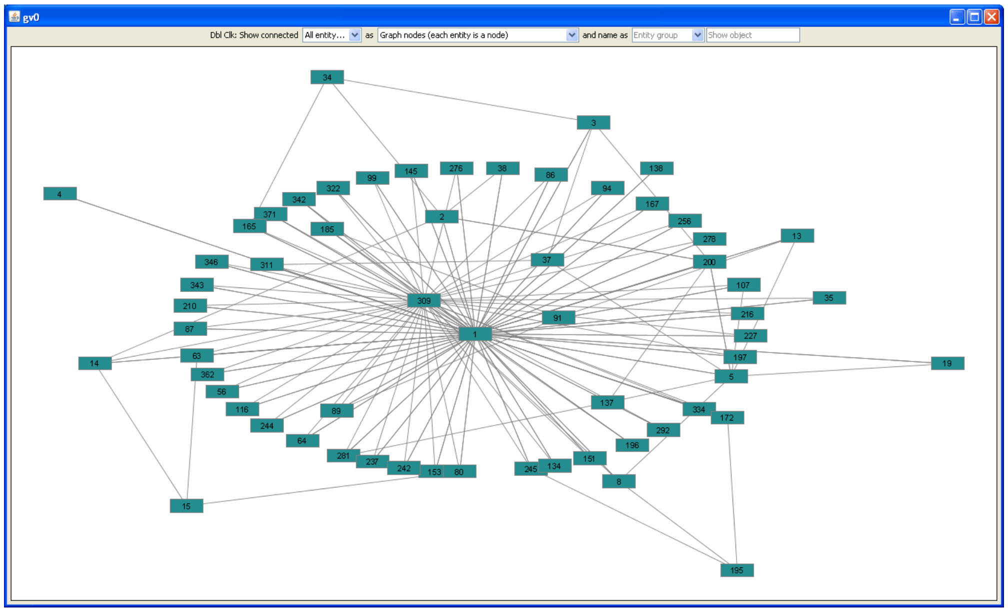


Fig. 8. Phone 309 is likely the replacement of phone 1

alyst can use the same strategy to check all the 6 phones: phones 1, 2, 3, 5, 97, and 137, a process made much simpler and more reliable via script reuse, as described above.

In Figure 5, the analyst chose to find all phones related to the current phone. Since the replacement phone should be active in the same region, s/he can also search for all towers related to the problematic phone, and then look for phones that connect to all (or most) of those towers. Through user interactions, the values of `relatedPhone` and `replacePhone` are replaced by the following:

```
entityVariable relatedPhone = relatedNodes(originalPhone, "tower");
entityVariable replacePhone = commonRelated(relatedPhone, "phone", -1);
// -1 means all entities
```

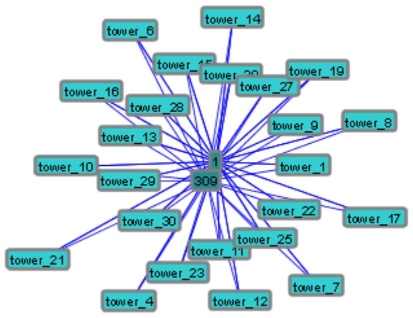


Fig. 9. Phone 309 was active within the same area as that of phone 1

With this updated model, the analyst sees that phone 309 is also active within the same area as phone 309 (Figure 9). The variable `relatedPhone` contains all of the towers that the `originalPhone` was connected to. These scripts modify the value of a non-source node, which should change the link among nodes, however, since the nodes involved in the new commands are the same as in the old commands, the graph structure of the model remains unchanged.

By modifying the scripts or re-assigning values to nodes, the analyst can reuse this symbolic model to check for other phones and verify results. Thus repetitive interactions are avoided and analysis efficiency is enhanced.

4.2. Use Case B: 2009 mini challenge MC2

In some problems, the analyst can identify rules and then convert these rules into mathematical constraints. Solving such problem become solving these constraints. In this VAST 2009 challenge[IEEE VAST 2009], we converted rules of the criminal network structure into CZsaw script statements. The symbolic model becomes the representation of the network structure. Plug into different parameters into the symbolic model, the user can explore different network structures and see different group of suspects.

4.2.1. Identifying the social network structure. The IEEE VAST 2009 mini challenge 2 was a social network and geo-spatial problem. The data provides a social network of about 6,000 people, some of whom, we are told, form a criminal ring. The structure (connections among people) of the criminal ring is known as one of two possibilities. This challenge asks us to determine the criminal ring structure (between two possibilities A and B) and identify the people (and their roles) within this criminal ring.

There are two possible criminal ring structures, A and B. Due to length limitations of this paper, we only discuss our analysis of structure A (B turned out to be a false answer). The description of this structure is:

> A. The employee has about 40 Flitter contacts. Three of these contacts are his “handlers”, people in the criminal organization assigned to obtain his cooperation. Each of the handlers probably has between 30 to 40 Flitter contacts and shares a common middle man in the organization, who we have code-named Boris. Boris maintains contact with the handlers, but does not allow them to communicate among themselves using Flitter. Boris communicates with one or two others in the organization and no one else. One of these contacts is his likely boss, who we’ve code-named the Fearless Leader.

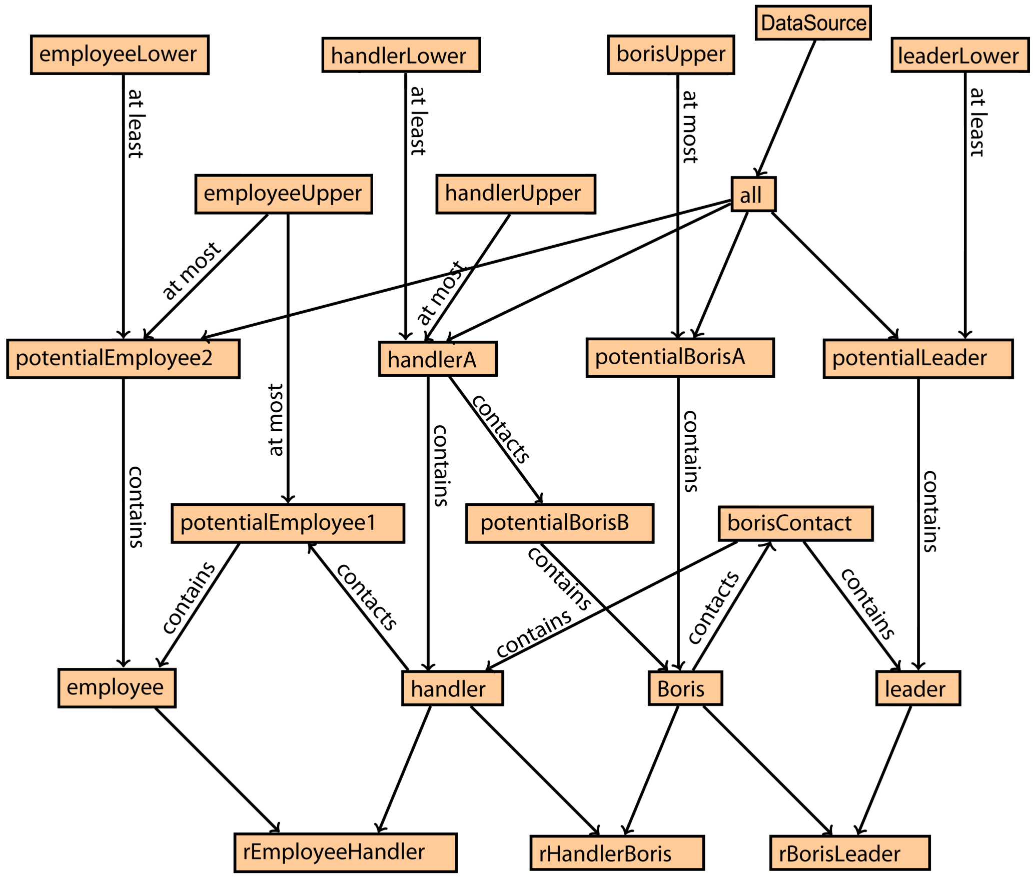


Fig. 10. The symbolic model for the 2009 social network challenge

The Fearless Leader probably has a broad Flitter network (well over 100 links), including international contacts [IEEE VAST 2009].

4.2.2. The Solution. To solve the problem, the analyst needs to check the linkage data to see if it matches the hypothesized network structure. A Flitter member is a CZSaw entity with type “person”. CZSaw relations are simply the Flitter links. The command `commonRelated` is essential here to search for a person who is related to a number of “at least” (lower boundary) and “at most” (upper boundary) Flitter members.

Given the facts of the criminal ring structure A (as below), the analyst can construct a symbolic model like Figure 10. Constraints among nodes are all derived from the network structure statements.

— At first the analyst puts all Flitter members into a node `all`:

```
    entityVariable all = search(‘‘%’’, ‘‘person’’);
            //% is a wildcard to search for matching entities
```

— *Each of the handlers probably has between 30 to 40 Flitter contacts*: the analyst sets up two source nodes, `handlerLower` and `handlerUpper`, with the initial values of 30 and 40. The child node `handlerA` contains Flitter members who have `handlerLower` to `handlerUpper` contacts.

```
entityVariable handlerA = commonRelated(all,
                                        ``person'',
                                        handlerLower,
                                        handlerUpper);
```

—*Handlers share a common middle man in the organization, whom the analyst has code-named Boris*: `handlerA` has a child node `potentialBorisB`. `potentialBorisB` contains members who contact at least three members in the node `handlerA`:

```
entityVariable potentialBorisB = commonRelated(handlerA,
                                               ``person'',
                                               3);
```

—*Boris communicates with one or two others in the organization and no one else*: the node `potentialBorisA` contains members that have at most `borisUpper` contacts. `borisUpper` is a small number like 4 or 5–three handlers plus one or two others.

```
entityVariable potentialBorisA = commonRelated(all,
                                               ``person'',
                                               1,
                                               borisUpper);
```

—*Boris* node contains members that are both in `potentialBorisA` and `potentialBorisB`:

```
entityVariable Boris = intersection(potentialBorisA, potentialBorisB)
\\ find entities in both potentialBorisA and potentialBorisB
```

—All *Boris*'s contacts are in the node `borisContact`, in which should contain the handler and the leader.

```
entityVariable borisContact = relatedNodes(``person'', Boris);
```

The intersection of `borisContact` and `handlerA` makes the node `handler`:

```
entityVariable handler = intersection(borisContact, handlerA);
```

—*Employee ... Three of these contacts are his handlers*: Members in `potentialEmployee1` contact at least three members in `handler`:

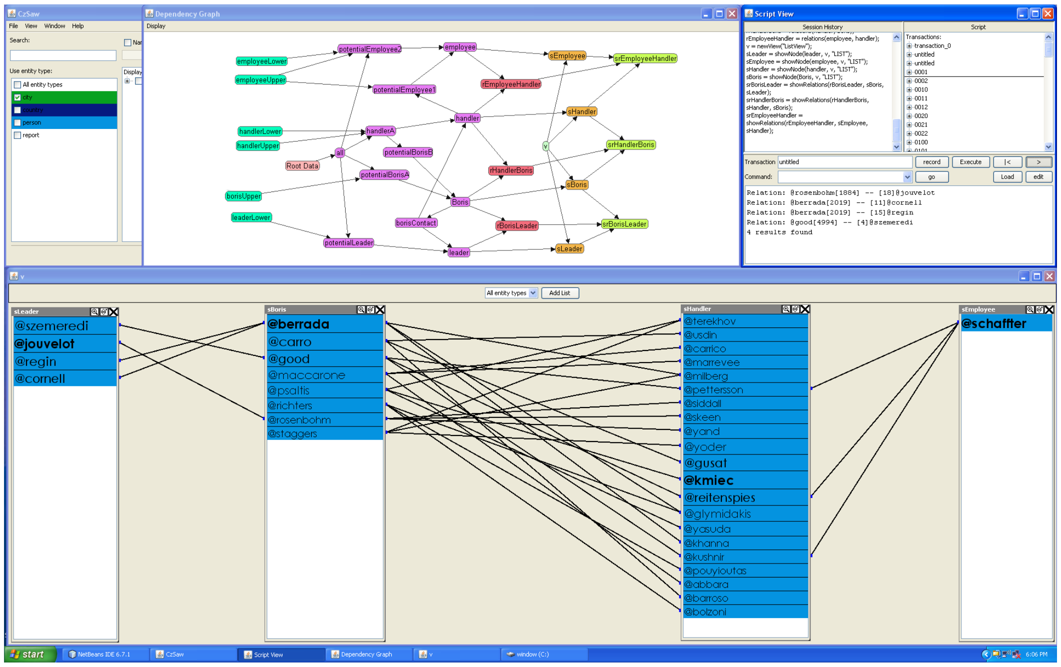
```
entityVariable potentialEmployee1 = commonRelated(handlers,
                                                  ``person'',
                                                  3);
```

—*The employee has about 40 Flitter contacts*: `potentialEmployee2` contains members who have about 40 contacts (could be a range, e.g., 35 - 45, which can be assigned with two nodes `employeeUpper` and `emplyeeLower`):

```
entityVariable potentialEmployee2 = commonRelated(all,
                                                  ``person'',
                                                  employeeLower,
                                                  employeeUpper);
```

Members in both `potentialEmployee1` and `potentialEmployee2` make the node `employee`:

```
 entityVariable employee = intersection(potentialEmployee1, potentialEmployee2)
```



```
rEmployeeHandler = relations(employee, handler);
v = newView("ListView");
sLeader = showNode(leader, v, "LIST");
sEmployee = showNode(employee, v, "LIST");
sHandler = showNode(handler, v, "LIST");
sBoris = showNode(Boris, v, "LIST");
srBorisLeader = showRelations(rBorisLeader, sBoris,
sLeader);
srHandlerBoris = showRelations(rHandlerBoris,
sHandler, sBoris);
srEmployeeHandler =
showRelations(rEmployeeHandler, sEmployee,
sHandler);
```

```
Relation: @rosenbohm[1884] -- [18]@jouvelot
Relation: @berrada[2019] -- [11]@cornell
Relation: @berrada[2019] -- [15]@regin
Relation: @good[4994] -- [4]@szemeredi
4 results found
```

{employeeLower:40, employeeUpper:40, handlerLower:30, handlerUpper:40, borisUpper:5, leaderLower:100}

Fig. 11. Using CZSaw to solve the VAST 2009 social network challenge

— *Fearless Leader probably has a broad Flitter network (well over 100 links)*: The node `potentialLeader` contains members that have at least 100 (`leadLower`) contacts:

```
entityVariable potentialLeader = commonRelated(all,
                                               ‘‘person’’,
                                               leadLower);
```

— *One of these contacts (of Boris) is his likely boss*: The leader must be in both `potentialLeader` and `borisContact`:

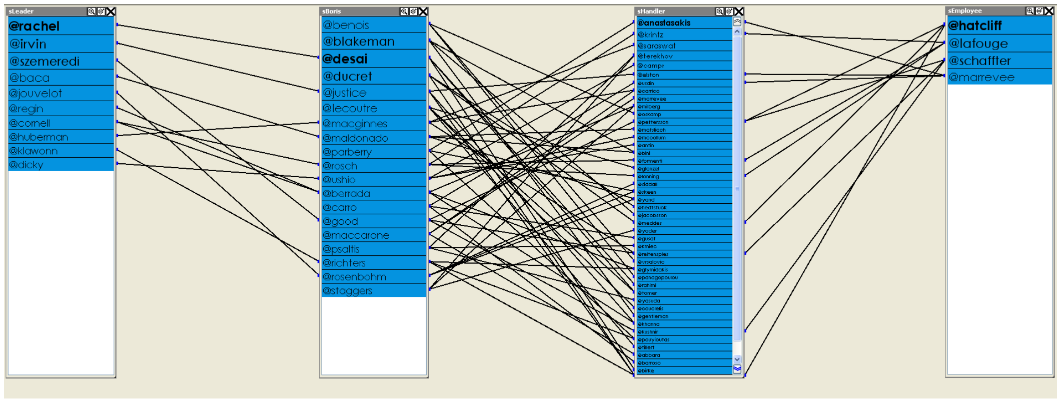
```
entityVariable leader = intersection(potentialLeader, borisContact);
```

— To visually examine connections, we need to identify relations (nodes `rEmployeeHandler`, `rHandlerBoris`, `rBorisLeader`) among nodes `leader`, `Boris`, `handler` and `employee`:

```
relationVariable rEmployeeHandler = relations(employee, handler);
```

Based on this symbolic model (Figure 10), the analyst manually develops a script to generate a solution. Running this script in CZSaw yields the Flitter network shown in Figure 11. The dependency graph in Figure 11 has several additional nodes for visualizing the network. Nodes like `sEmployee`, `srEmployeeHandler` are the visualizations of suspects and their relations.

From the question description, the analyst sets initial values for the 6 source nodes in the model, `employeeLower:40`, `employeeUpper:40`, `handlerLower:30`, `handlerUpper:40`, `borisUpper:5`, `leaderLower:100`. With this model and the given values, CZSaw displays groups of people in its list view (bottom part of Figure 11). The



{employeeLower:30, employeeUpper:50, handlerLower:30, handlerUpper:40, borisUpper:6, leaderLower:70}

Fig. 12. Other possible suspects with a larger range of possibilities

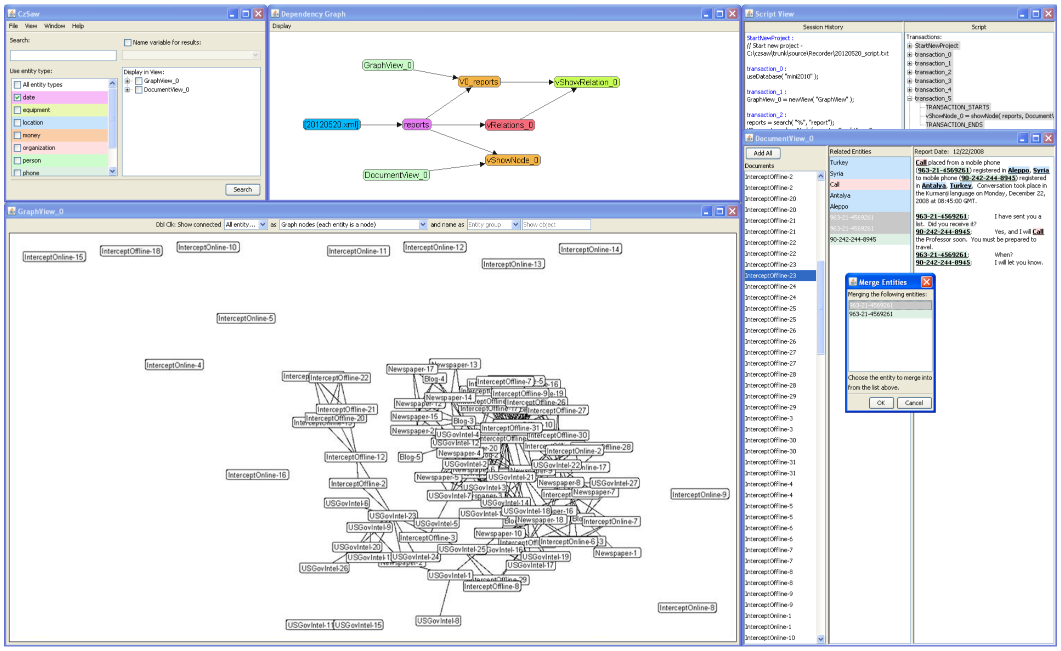
bottom half of the screenshot is a graph showing connections among people. This graph shows four groups of people, from left to right, where each contains the name(s) of suspected leaders, Boris (middle man), handler, and employee. Edges among these people show the Flitter connections. People with no-connections can be filtered out visually. The analyst can see under the Employee group that there is only one person `@schaffter`, who can be determined to be the suspect embassy employee. By following its edges, s/he sees three people (`@pattersson`, `@reitenspies`, and `@kushnir`), who should be the three handlers. By following the edge to the group which contains Boris, s/he sees that Boris' ID is `@good` and the leader's ID is `@szemeredi`. This result perfectly matches the published answer. There is no suitable Flitter member who can match the network structure B.

However, in the real world, these numbers may keep on changing: members in Flitter may grow or a member may change his/her contact. Also, some information (e.g., the numbers of contacts) may not be accurate enough. Therefore, analysts should be able to investigate alternative suspects by using different values. For example, the analyst may want to expand the range of suspects of employees, wondering: What if the employee has 35 or 45 contacts? Or, what if Boris has more than 2 other contacts (allowing him at most to have 6 or maybe 10 contacts)? By applying a different set of values {`employeeLower:30, employeeUpper:50, handlerLower:30, handlerUpper:40, borisUpper:6, leaderLower:70`}, the model updates the list view as shown in Figure 12, which suggest a larger range of suspects.

4.3. Use Case C: 2010 mini challenge MC1

Some VA system (e.g. Jigsaw [Stasko et al. 2007] use entity-relation approach to analyze text documents. The analyze outcome highly rely on the accuracy of extracted entity. However, automated entity extraction is often imperfect. Solely base on automated entity extraction may create misleading visualization results. Human aided entity extraction could be an essential touch up to make the analysis more accurate. In this use case, we demonstrate how the propagation mechanism automatically adjust visualizations while the analyst manually refine entities.

VAST 2010's mini challenge 1 was designed to investigate firearms dealing activities from 103 text records, which come from different resources and describe different countries, regions, and people [IEEE VAST 2010]. Within the encompassing story of



```
StartNewProject :
// Start new project -
C:\czsaw\trunk\source\Recorder\20120520_script.txt

transaction_0 :
useDatabase( "mini2010" );

transaction_1 :
GraphView_0 = newView( "GraphView" );

transaction_2 :
reports = search( "%", "report");
```

```
Transactions:
  StartNewProject
  transaction_0
  transaction_1
  transaction_2
  transaction_3
  transaction_4
  transaction_5
    TRANSACTION_STARTS
    vShowNode_0 = showNode( reports, Document\
    TRANSACTION_ENDS
```

Report Date: 12/22/2008

Call placed from a mobile phone (963-21-4569261) registered in Aleppo, Syria to mobile phone (90-242-244-8945) registered in Antalya, Turkey. Conversation took place in the Kurmanji language on Monday, December 22, 2008 at 08:45:00 GMT.

963-21-4569261: I have sent you a list. Did you receive it?
90-242-244-8945: Yes, and I will Call the Professor soon. You must be prepared to travel.
963-21-4569261: When?
90-242-244-8945: I will let you know.

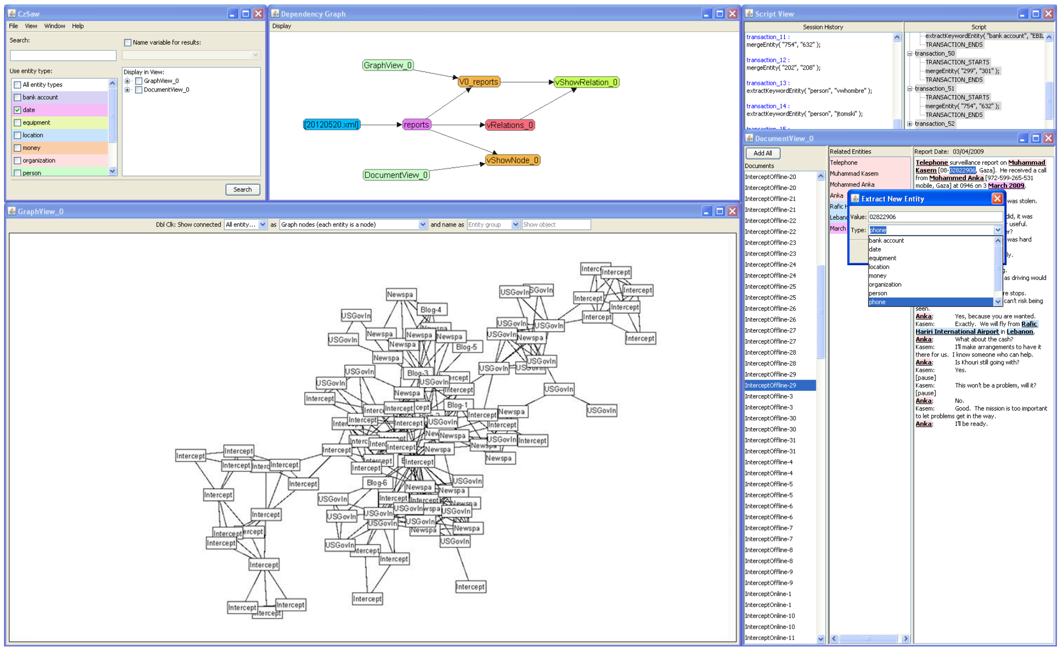
Fig. 13. Originally records are either isolated or condensly clustered

illegal firearms dealing activities, there are also several sub-threads. Many errors or inconsistencies exist in these documents, such as misspelled names in surveillance reports. It is basically impossible for an analyst (or several analysts) to keep track of the whole scenario through reading these documents one by one even if they had enough time to read them all. Before entering the data into CZSaw, we pre-processed the text records for named entity extraction using Alias-i's Lingpipe system [Alias-i 2012] and then imported into CZSaw [Chen et al. 2010]. Entities extracted include: person, organization, city, country, date, etc.

To make sense from these records, we visualized the record network (Figure 13). Two records are connected if they contain the same entity. It takes only 6 interactions to build the required CZSaw model:

(1) Use the source data `20120520.xml`, which creates the data source node `[20120520.xml]` in the graph.
(2) Create a graph view. CZSaw automatically gave the view a name `GraphView_0`
(3) Search for all records, save the result into a variable `reports`, and show this in the view `GraphView_0`. The visualization of `reports` (a ShowNode CZSaw object) is called `V0_reports`.
(4) Right click on the ShowNode object `V0_reports`, from the drop down menu, selecting the command to show relations of the entities in the entityVariable `reports` to the entities in `reports` itself. Two reports are related if they contain the same entity (entities). This step creates two variables `Relations_0` and `vShowRelation_0`.
(5) Open a document view. CZSaw assigned it the name `DocumentView_0`.
(6) Right click on the showEntities `V0_reports` and put all reports in `reports` into the `DocumentView_0`

While reading reports shown in `DocumentView_0`, the analyst sees that the *machine-extracted* entities contain many errors. They are neither fully accurate nor complete.



```
transaction_11 :
mergeEntity( "754", "632" );

transaction_12 :
mergeEntity( "202", "208" );

transaction_13 :
extractKeywordEntity( "person", "vwhombre" );

transaction_14 :
extractKeywordEntity( "person", "jtomski" );
```

```
    extractKeywordEntity( "bank account", "EBIL
    TRANSACTION_ENDS
transaction_50
    TRANSACTION_STARTS
    mergeEntity( "299", "301" );
    TRANSACTION_ENDS
transaction_51
    TRANSACTION_STARTS
    mergeEntity( "754", "632" );
    TRANSACTION_ENDS
transaction_52
```

Report Date: 03/04/2009

Telephone surveillance report on **Muhammad Kasem** [08-02822906, Gaza]. He received a call from **Mohammed Anka** [972-599-265-531 mobile, Gaza] at 0946 on 3 **March 2009**.

Value: 02822906
Type: phone

Anka: Yes, because you are wanted.
Kasem: Exactly. We will fly from **Rafic Hariri International Airport** in **Lebanon**.
Anka: What about the cash?
Kasem: I'll make arrangements to have it there for us. I know someone who can help.
Anka: Is Khouri still going with?
Kasem: Yes.
[pause]
Kasem: This won't be a problem, will it?
[pause]
Anka: No.
Kasem: Good. The mission is too important to let problems get in the way.
Anka: I'll be ready.

Fig. 14. After many entity refinement operations, documents start to build clusters

CZSaw provides a set of functions (e.g. extracting entities, merge entities, link/unlink entity with reports) to allow the analyst to correct errors while managing the entities [Chen et al. 2010]. With these functions, the analyst can manually refine entities while reading a report or working within other views (e.g. similar nodes in Hybrid View's entity network). These entity refinement operations create/modify/remove entities and the relations of reports to entities, which changes values in the data source node `[20120520.xml]`. The propagation mechanism automatically re-evaluates all nodes in the graph, which automatically updates the visualization in all views. Entity refinement does not create new reports. Since two reports are related if they contain the same entity, the entity refinement operation might also change the relations among reports. Extracting entities may create new relations among reports, while unlinking entities will remove relations among reports. Each entity refinement interaction will update the report network in the view. After many entity refinement operations, these reports build up different clusters (Figure 14), which show us the organization of the collection of reports and suggest an appropriate order for reading the reports. Reports within a cluster usually focus on one topic. Reports connecting different clusters *may be* worth reading precisely because they tend to reveal connections between groups in the real world.

5. CONCLUSIONS

Provenance tells us how conclusion were drawn or decisions made. Capturing and reusing the reasoning is particularly apt in the visual analytics domain where the concept "analytic provenance" has become a major research focus [Benzaken et al. 2011; Gotz and Zhou 2008; Silva et al. 2007; Eccles et al. 2007; Kreuseler et al. 2004]. Based on propagation-based parametric design systems, we present a script and symbolic model approach to represent analytic provenance. We see similarities between how a designer creates designs and how an analyst plans and executes an analysis process, and then adopts the symbolic model to represent the structure of the analytical reasoning process. In CZSaw, the analyst can recall the chronological states of the analysis process with CZSaw scripts, and may interpret the rationale of the analysis with the symbolic model underlying the analysis process.

Our development of the scripting capability and symbolic model in CZSaw matches North et al. [2011]'s five stages to support analytics provenance: perceive, capture, encode, recover, and reuse. Within CZSaw's data visualization environment, user interactions are captured and encoded into CZSaw scripts, which contain blocks of commands. The user can replay these interactions, which are basically a history record of the analysis process. To encourage the analyst to recover and reuse phases of the analytics process, the symbolic model "recovers" by visualizing the relationship between the results and interactions. It is vital for the analyst to understand how other analysts (or in fact the analyst him/herself) drove the visual analytics system to obtain the current results. Then, it is possible for the analyst to reuse parts of the CZSaw script and reapply an analytical strategy to explore different parameter settings or solve similar problems.

We employed data from three VAST mini challenges to demonstrate usage of our model and its propagation mechanism. While working on the three VAST challenges, the problem solving strategy are well planed and the analysis process are "structured and disciplined" [Thomas and Cook 2005]. In the first use case, the analyst can execute a planned solution through a series of interactions and later reuse the strategy to solve similar problems. In the second case, the symbolic model is a representation of the criminal network structure, which makes the solution obvious. Solving the problem is like solving a series of mathematical equations. With different parameters, the user can explore different sets of suspects. In the third case, the propagation mechanism of the model allows the analyst to work effectively when updating data in a WYSIWYG way. The data changes are reflected in the visualizations immediately.

In this paper, we proposed a feasible mechanism to support visual analytics provenance by recording and reusing. It maybe hard for a novice non-programming expert to fully utilize the CZsaw script's programming capability. This problem actually existed in may systems with end-user programming capabilities. We studied architecture learning scripting to build complex parametric models [**?**]. While the learning curve is steep, architects are willing to spend time and effort to learn scripting because it gives them a powerful tool to create complex 3D architectural models. This study tracking architects' work processes in multi-day sessions helped us understand both the change in the nature of attempted tasks and in user development of new idioms of use. We believe that in certain complex situation, analyst may find that the programming skill may enhance much on their analysis outcome.

For future development, there are two directions in which we may proceed:

Enhance the interaction design to be more "intelligent". We can aim to develop an interactive intelligent visual analytic system. An interactive intelligent system can be viewed from two aspects: intelligent technology and human interaction [Jameson and Riedl 2011]. The current propagation-based symbolic model is a simple form of constraint solver. It is a good starting point. We will improve upon the design both at the lower-level to support information foraging loops and at the higher-level to support sensing making loops. The goal is to allow the analyst to interactively build a more "clever" model to solve more complex analytics problems.

Compare and communicate analytical strategies. For a given analytics problem, different users may have different solutions. The interaction history recorded in the script and the corresponding symbolic model can be varied. Ziemkiewicz et al. [2012] suggest that a visualization should be studied in the context of differences among its users. With both the history recording scripts and the symbolic model, CZSaw provides us with the possibility to study the individual's analysis process. We believe that analytics provenance is not only about recalling the processes performed by the analyst himself, but also about communicating the provenance among different analysts in order to let others better understand how the decision was made.

REFERENCES

AISH, R. AND WOODBURY, R. 2005. Multi-level interaction in parametric design. *Proceedings of Smart Graphics 2005*, 151–162.

ALIAS-I. 2012. Lingpipe 4.1.0. [http://alias-i.com/lingpipe (accessed May 1st, 2010)].

BENTLEY SYSTEMS, INC. 2010. Generativecomponents v8i. [http://www.bentley.com/en-us/products/generativecomponents/ (accessed May 5th, 2011)].

BENZAKEN, V., FEKETE, J., HÉMERY, P., KHEMIRI, W., AND MANOLESCU, I. 2011. Ediflow: data-intensive interactive workflows for visual analytics. In International conference on Data Engineering (ICDE 2011), Hannover, Germany, 04 2011. IEEE.

CALLAHAN, S. P., FREIRE, J., SANTOS, E., SCHEIDEGGER, C. E., SILVA, C. T., AND VO, H. T. 2006. Managing the evolution of dataflows with vistrails. In *ICDEW '06: Proceedings of the 22nd International Conference on Data Engineering Workshops*. IEEE Computer Society, Washington, DC, USA, 71.

CHEN, Y. V., DUNSMUIR, D., KADIVAR, N., LEE, E., GUENTHER, J., JANI, S. A., DILL, J., SHAW, C., WOODBURY, R., STONE, M., AND QIAN, C. 2010. Czsaw: Model based interactive analysis of interwoven, imprecise narratives, vast 2010 mini challenge 1 award: Outstanding interaction mode. In *Proceedings of IEEE Visual Analytics Science and Technology 2010*. IEEE, Salt Lake City, Utah.

COX, P. T., P. T. 1990. Using a pictorial representation to combine dataflow and object-orientation in a language-independent programming mechanism. *Glinert, E. P., editor, Visual Programming Environments: Paradigms and Systems*.

DASSAULT SYSTÈMES SOLIDWORKS CORP. 2011. SolidWorks 2011. [http://www.solidworks.com/ (accessed August 10th, 2011)].

DENNIS, J. B. 1974. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*. Springer-Verlag, London, UK, UK, 362–376.

DERTHICK, M. AND ROTH, S. F. 2001. Enhancing data exploration with a branching history of user operations. *Knowledge Based Systems 14,* 1-2, 65–74.

DUNSMUIR, D., LEE, E., SHAW, C. D., STONE, M., WOODBURY, R., AND DILL, J. 2012. A focus + context technique for visualizing a document collection. In *Hawaii International Conference on System Sciences*. IEEE Computer Society, Los Alamitos, CA, USA, 1835–1844.

DUNSMUIR, D., Z., B. M., CHEN, Y. V., JOORABCHI, M. E., JOORABCHI, M. E., ALIMADADI, S., LEE, E., DILL, J., QIAN, C., SHAW, C., AND WOODBURY, R. 2010. Czsaw, imas & tableau: Collaboration among teams, vast 2010 mgrand challenge award: Excellent student team analysis. In *Proceedings of IEEE Visual Analytics Science and Technology 2010*. IEEE, Salt Lake City, Utah.

ECCLES, R., KAPLER, T., HARPER, R., AND WRIGHT, W. 2007. Stories in geotime. *IEEE Symposium on VAST 2007*, 3–17.

FRUCHTERMAN, T. M. J. AND REINGOLD, E. M. 1991. Graph drawing by force - directed placement. *Software -Practice and Experience (Wiley) 21,* 11.

GARG, S.; NAM, J. R. I. M. K. 2008. Model-driven visual analytics. In *IEEE Symposium on Visual Analytics Science and Technology*. 19–26.

GOTZ, D. AND ZHOU, M. X. 2008. Characterizing users' visual analytic activity for insight provenance. *Information Visualization 8,* 1, 42–55.

HEER, J., MACKINLAY, J. D., STOLTE, C., AND AGRAWALA, M. 2008. Graphical histories for visualization: Supporting analysis, communication, and evaluation. *IEEE Transactions on Visualization and Computer Graphics 14,* 6, 1189–1196.

HILS, D. D. 1992. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing 3,* 1, 69 – 101.

HOFFMAN, C. M. AND JOAN-ARINYO, R. 2005. A brief on constraint solving. *Computer-Aided Design and Applications 2,* 5, 655–664.

IEEE VAST. 2008. Vast 2008 challenge mc3: Cell phone calls. [http://www.cs.umd.edu/hcil/VASTchallenge08 (accessed May 31st, 2012)].

IEEE VAST. 2009. Vast 2009 challenge mc2: Social network and geospatial. [http://hcil.cs.umd.edu/localphp/hcil/vast/index.php (accessed May 31st, 2012)].

IEEE VAST. 2010. Vast 2010 challenge mc1: Text records - investigations into arms dealing. [http://hcil.cs.umd.edu/localphp/hcil/vast10/index.php (accessed May 31st, 2012)].

JAMESON, A. AND RIEDL, J. 2011. Introduction to the transactions on interactive intelligent systems. *ACM Trans. Interact. Intell. Syst. 1,* 1, 1:1–1:6.

JANKUN-KELLY, T.J., M. K. G. M. 2007. A model and framework for visualization exploration. *IEEE Trans. on Visualization and Computer Graphics 13,* 2, 357–369.

JOHNSTON, W. M., HANNA, J. R. P., AND MILLAR, R. J. 2004. Advances in dataflow programming languages. *ACM Comput. Surv. 36,* 1, 1–34.

KADIVAR, N. 2011. Visualizing the analysis process: Czsaw's history view. M.S. thesis, Simon Fraser University.

KADIVAR, N., CHEN, Y. V., DUNSMUIR, D., LEE, E., QIAN, C., DILL, J., SHAW, C., AND WOODBURY, R. 2009. Capturing and supporting the analysis process. In *Proceedings of IEEE Visual Analytics Science and Technology*. Atlantic City, NJ, 131–138.

KAHN, A. B. 1962. Topological sorting of large networks. *Commun. ACM 5,* 11, 558–562.

KESSLER, M. 1963. Bibliographic coupling between scientific papers. *American Documentation* 12.

KREUSELER, M., NOCKE, T., AND SCHUMANN, H. 2004. A history mechanism for visual data mining. In *IEEE Symposium on Information Visualization 2004*. IEEE Computer Society, Austin, Texas, USA, 49–56.

MCNEEL, R. 2010. Grasshopper - generative modeling for rhino. [http://http://www.grasshopper3d.com (accessed May 25th, 2012)].

NIEMEYER, P. 2005. Beanshell - lightweight scripting for java. [Online; accessed May 5th, 2011].

NOOY, W. D., MRVAR, A., AND BATAGELJ, V. 2005. *Exploratory Social Network Analysis with Pajek*. Cambridge University Press.

NORTH, C., CHANG, R., ENDERT, A., DOU, W., MAY, R., PIKE, B., AND FINK, G. 2011. Analytic provenance: Process+interaction+insight. The 2011 Annual Conference Extended Abstracts on Human Factors in Computing Systems, Vancouver, BC, Canada. 33-36.

PIKE, W. A., S. J. C. R. AND O'CONNELL, T. A. 2009. The science of interaction. *Information Visualization 8,* 4, 263–274.

PIROLLI, P. AND CARD, S. 1999. Information foraging. *Psychological Review 106,* 4, 643–675.

RUSSELL, D.M., S. M. P. P. AND CARD, S. 1993. The cost structure of sensemaking. In *Proceedings of INTERACT'93 and CHI'93 conference on Human Factors in Computing Systems*. ACM Press., 269–276.

SCHOLTZ, J., WHITING, M. A., PLAISANT, C., AND GRINSTEIN, G. 2012. A reflection on seven years of the vast challenge. In *Proceedings of the 2012 BELIV Workshop: Beyond Time and Errors - Novel Evaluation Methods for Visualization*. BELIV '12. ACM, New York, NY, USA, 13:1–13:8.

SHIPMAN, F.M., H. H. 2000. Navigable history: A reader's view of writer's time. *The New Review of Hypermedia and Multimedia 6,* 1, 147–167.

SHNEIDERMAN, B. 1996. The eyes have it: A task by data type taxonomy for information visualization. *Proceedings of IEEE Visual Language*, 336–343.

SHRINIVASAN, Y. B. AND VAN WIJK, J. J. 2008. Supporting the analytical reasoning process in information visualization. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*. ACM, New York, NY, USA, 1237–1246.

SILVA, C., FREIRE, J., AND CALLAHAN, S. 2007. Provenance for visualizations: Reproducibility and beyond. *IEEE Computing in Science & Engineering 9,* 5, 82–89.

SMALL, H. 1973. Co-citation in the scientific literature: A new measure of the relationship between two documents. *Journal of the American Society of Information Science* 24, 265–269.

STASKO, J., GORG, C., LIU, Z., AND SINGHAL, K. 2007. Jigsaw: Supporting investigative analysis through interactive visualization. *IEEE Symposium on VAST 2007*, 131–138.

SUTHERLAND, I. E. 1963. Sketchpad, a man-machine graphical communication system. Ph.D. thesis, Massachusetts Institute of Technology.

THOMAS, J. J. AND COOK, K. A. 2005. *Illuminating the Path: The Research and Development Agenda for Visual Analytics*. National Visualization and Analytics Center, 11662 Los Vaqueros Circle, Los Alamitos, CA.

WOODBURY, R. 2010. *Elements Of Parametric Design*. Routledge.

WRIGHT, W., SCHROH, D., PROULX, P., SKABURSKIS, A., AND CORT, B. 2006. The sandbox for analysis – concepts and methods. *ACM CHI 2*, 801–810.

XIAO, L., GERTH, J., AND HANRAHAN, P. 2006. Enhancing visual analysis of network traffic using a knowledge representation. In *Proceedings of IEEE Visual Analytics Science and Technology*. 107–114.

YI, J., KANG, Y., STASKO, J., AND JACKO, J. 2007. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Trans. on Visualization and Computer Graphics 13,* 6, 1224–1231.

ZIEMKIEWICZ, C., OTTLEY, A. R., CROUSER, J., CHAUNCEY, K., SU, S. L., AND CHANG, R. 2012. Understanding visualization by understanding individual users. *IEEE Computer Graphics and Applications 32,* 6, 88 – 94.

APPENDIX

A.1. CZsaw commands

A.1.1. CZsaw objects. CZsaw objects and commands are built based on

```
Entity(int id, String value, String type, (optional)String name)
```

An entity has unique ID, type, and value. For user's convenience, it could have an optional name for easier display and recognize. If the name is omit, the entity's value will be used for showing in the visualization.

```
Record( int id, String name, ArrayList Entities, String text) extends Entity
```

In our implementation, the record is a special extension of the entity by adding a list of entities in its attributes.

```
Relation(Entity leftEntity, Entity rightEntity, (optional)Number strength=1 )
```

A relation is a pair of entities (or records). If not explicitly defined, the default strength is 1.

```
EntityVariable(ArrayList entities)
```

An entity variable contains a list of entities (records).

```
RelationVariable(ArrayList entities)
```

An relation variable contains a list of relations.

A.1.2. Data Query Commands

```
EntityVariable search(String Keyword, String EntityType,
                          (optional)String operator=‘‘==’’ )
```

Get all entities with the value matches the `Keyword` and type is `EntityType`. If omitted, the default value of the operator is "$=$" (equal to). Other possible values are "$<$", "$\leq$","$>$", and "$\geq$", which can let the user search for values by comparison. Wildcard "%" can be used as part in the `keyword` to substitute for zero or more characters. For example, `search(‘‘Eve’’,‘‘person’’)` will return all people with name exactly as "Eve". `search(‘‘Eve%’’,‘‘person’’)` will return all people with name started with "Eve", such as "Everil", "Everilda","Evelina", "Evelyn" etc. In all commands, the value of `EntityType` could be "record" to retrieve records with their text content matching the keyword. If the EntityType is "%", then all kinds of entities include reports will be searched. Due to BeanShell's support of loose types, if the keyword contain only digits, the value will be stored as numbers and can be compare or compute mathematically.

```
EntityVariable relatedNodes(String EntityType, entityVariable EV):
```

Return an EntityVariable with all entities related to entities in `EV` with type `EntityType`. If `entityType` is "%", get related entities in all kinds. The `EntityType` could be "record" to get all related records.

```
EntityVariable commonRelatedNodes(String entityType, entityVariable EV,
                                  int lowerLimit)
```

Get all entities that connect to at least `lowerLimit` number of entities in `EV`.

```
EntityVariable commonRelatedNodes(String entityType, EntityVariable EV,
                                  int lowerLimit, int upperLimit)
```

Get all entities that connect to at least `lowerLimit` but no more than `upperLimit` of entities in `EV`.

```
EntityVariable relatedRecords(EntityVariable EV)
```

Get all related records of `EV`. Same as `relatedNodes(‘‘record’’, EntityVariable EV)`.

```
RelationVariable relations(EntityVariable Ev1, EntityVariable Ev2)
```

Get all relations between entities in `Ev1` and entities in `Ev2`.

```
EntityVariable getGlobalEntity(int entityID)
```

Get one entity by its ID `entityID`

```
EntityVariable getGlobalEntity(int[] entityIDs[])
```

Get several of entities by the array `entityIDs`

```
Entity getEntity(int entityID)
```

Get one entity by its ID.

A.1.3. Visualization Commands

```
View newView(string ViewType)
```

Create and display a new data view. Possible `ViewType` values: “GraphView”, “ListView”, “DocumentView”, “SemanticZoomView”

```
void hide(View ViewWindow)
```

Hide the `ViewWindow`.

```
void show(View ViewWindow)
```

Make the `ViewWindow` visible.

```
ShowNode showNode(EntityVariable EV, view VW, string ShowType):
```

Visualize the `EV` in view `VW` with the type of `ShowType`. Possible show types values include: “graph”, “list”, “group”, “disc”.

```
void hide(ShowNode SNode)
```

Hide the `SNode` from its view.

```
show(ShowNode SNode)
```

Make the `SNode` visible.

```
ShowNode object methods:
```

```
   Public void changeContent(EntityVariable EV)
```

Replace the current visualized EntityVariable of the `ShowNode` by another EntityVariable `EV`.

`Public void setVisible(Boolean ShowHide)`

Set the visible status of the `ShowNode`

`Public void changeVisType(String VisualizationType)`

Change the visualization type of `ShowNode`.

`ShowRelation showRelation(RelationVariable RV, ShowNode SNode1, ShowNode SNode2):`

Visualize the RelationVariable `RV` in between the ShowNodes SNode1 and SNode2. Possible show types values include: “graph”, “list”, “group”, “disc”.

`hide(ShowRelation SRel)`

Hide the `SRel`.

`show(ShowRelation SRel)`

Make `SRel` visible

`ShowRelation object methods:`

`Public void changeContent(RelationVariable RV)`

Replace the visualized RelationVariable of the `ShowRelation` by another relation-Variable `rV`.

`Public void setVisible(Boolean ShowHide)`

Change the show/hide status of the `ShowRelation`.

A.1.4. Data Manipulation Commands

`EntityVariable union(EntityVariable EV1, EntityVariable EV2)`

Creates an new EntityVariable with entities either in `EV1` or `EV2`.

`EntityVariable intersection(EntityVariable EV1, EntityVariable EV2)`

Creates an new EntityVariable with entities both in `EV1` and `EV2`.

`EntityVariable difference(EntityVariable EV1, EntityVariable EV2)`

Creates an new EntityVariable with entities only in `EV1` and not in `eV2`.

`EntityVariable methods:`

`Public EntityVariable(Entity[] Entities)`

Create a new `EntityVariable` with an array of Entities.

`Public void setEntities(Entity[] Entities)`

Set the value of `EntityVariable` as the array of `Entities`. Duplicated entities will be removed.

`Public void removeEntities(EntityVariable EV)`

Remove `EV`’s entities from the `EntityVariable`.

`Public void removeEntity(Entity E)`

Remove the entity `E` from the `EntityVariable`.

`Public void removeEntities(Entity[] Entities)`

Remove many entities from the `EntityVariable`.

`Public void addEntities(EntityVariable EV)`

Add `EV`'s entities into `EntityVariable`. Duplicated entities will be removed.

`Public void addEntity(Entity E)`

Add the entity `E` into `EntityVariable`.

`Public void addEntities(Entity[] Entities)`

Add many entities into the `EntityVariable`. Duplicated entities will be removed.

`Relation newRelation{object A, object B}`

Create a new relation between object `A` and `B`.

`EntityVariable union(RelationVariable EV1, RelationVariable EV2)`

Creates an new RelationVariable with entities either in `RV1` or `RV2`.

`EntityVariable intersection(RelationVariable RV1, RelationVariable RV2)`

Creates an new relationVariable with entities both in `RV1` and `RV2`.

`EntityVariable difference(RelationVariable RV1, RelationVariable RV2)`

Creates an new RelationVariable with relations that are only in `RV1` and not in `RV2`.

`RelationVariable methods:`

`Public RelationVariable(Relation[] relations)`

Create a new `RelationVariable` with relations.

`Public void setRelations(Relation[] relations)`

Set the value of `RelationVariable` as the relations.

`Public void removeRelations(RelationVariable RV)`

Remove `RV`'s relations from the `RelationVariable`.

`Public void removeRelation(Relation R)`

Remove the relation `R` from the `RelationVariable`.

`Public void removeRelations(Relation[] relations)`

Remove many relations from the `RelationVariable`.

`Public void addRelations(EntityVariable RV)`

Add `RV`'s relations into `RelationVariable`. Duplicated relations will be removed.

`Public void addRelation(Relation R)`

Add the relation `R` into `RelationVariable`.

`Public void addRelations(Relation[] Relations)`

Add an array of relations into the `RelationVariable`. Duplicated relations will be removed.

A.1.5. Entity Refinement Commands

`Void linkEntity(int EntityID, int RecordID)`

link one entity (by ID) to a record (by ID). Many new relations will be built for this entity to existing entities in the record.

`Void unlinkEntity(int EntityID, int RecordID):`

remove the entity (by ID `EntityID`) from the record (by ID `RecordID`).

`Void mergeEntity(int EntityID1, int EntityID2)`

Replace the first entities (by ID EntityID1) by the second entity (by ID EntityID2) in all records. The first entity will be removed from the database.

`Void aliasEntity(int EntityID1, int EntityID2):`

Make the first entity (by ID EntityID1) an alias of the second entity (by ID EntityID2). Any records connects to the first entity will be linked to the second entity, and vise versa.

`Void removeEntity(int EntityID)`

Remove the entity (by ID EntityID) and all its associated relations from the data.

`Entity extractEntity(String EntityValue, String EntityType)`

Create an new entity with type `EntityType` and value `EntityValue`. Its unique ID is automatically generated. If a record contains a string value and the string contains the word `entityValue`, the new entity will be added to the record's entity set.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

The authors would like to acknowledge other members in the CZSaw research group, Dr. Chris Shaw, Maureen Stone, Dustin Dunsmuir, Eric Lee, Nazanin Kadivar, Jeffrey Guenther, Saba Alimadadi, for their work in other research directions on Visual Analytics to make CZSaw a fully functional VA application.

Received May 2012; revised Jan 2013; accepted