The Event Coder program User Manual

Wouter Spekkink

September 7, 2017

Chapter 1

Introduction

1.1 The Event Coder program

The document you are currently reading is the user manual for the open source (GPL 3.0) software tool called **Event Coder**, written by me (Wouter Spekkink) in C++ and Qt5 [1].

The program is intended to facilitate in the qualitative coding of event data. In this case, event data refers specifically to bracketed, chronologically ordered, qualitative data that captures actions and/or interactions that occurred in some type of process. My thinking about what event data is, and much of the terminology that I use, is heavily inspired by activities carried out as part of the Minnesota Innovation Research Program[2]. I have many other sources of inspiration that I will not mention in this manual. It should be easy enough to find my other sources of inspirations in the publications I have written (see my contact details in chapter 6).

Why is it useful to have this tool? Over the past years I have worked together with other people (especially Frank Boons) to develop and use various methods and tools for the study of social processes. In doing so, we have always worked with the principle that our fundamental units of analysis should be events, that is, things that "happened" or "came to pass", as opposed to, for example, variables. Based on ideas shared by the researchers from the Minnesota Innovation Research Program, we started collecting data in the form of *incidents*, which are qualitative, bracketed descriptions of actions or interactions, which consist (at least) of (1) an indication of the time at which the action or interaction occurred, (2) a (brief) qualitative description of the action or interaction, and (3) the source of data (e.g., a document, an interview, personal conversation)[3].

---

[1]The source code is available from my Github page (see chapter 6).

[2]see especially Poole, M. S., Van De Ven, A. H., Dooley, K., & Holmes, M. E. (2000). *Organizational Change and Innovation Processes: Theory and Methods for Research.* Oxford: Oxford University Press. I was not involved in this work in any way.

[3]The descriptions (point 2) are typically created by the researcher him/herself. When I

During the collection of event data, we create numerous incidents, which we store in chronologically ordered event data sets.

Again, following ideas shared by researchers from the Minnesota Innovation Research Program, we have been using qualitative coding procedures as a step in the analysis of event data. There can be multiple purposes for doing this, but at least one important purpose is (roughly) to relate the *empirical observations* recorded in incidents to *theoretical constructs*, which are not directly observable and play a role in *theories* that are developed to explain the some *empirical phenomenon* of interest to us. The **Event Coder** program can be used to facilitate such qualitative coding procedures. The program allows the user to assign attributes to incidents to express how these incidents relate to theoretically relevant constructs (see section 4.2 for different ways in which one could go about this). In addition, it also allows one to assign relationships to incidents to express what *relationships* between what *entities* are indicated by the observed incidents, which can be useful in studies that aim to answer questions about both *process* and *structure* and/or relationships between the two (see section 4.3 for different ways in which one could go about this). I and others have done some of these things using simpler tools, such as assigning attributes to incidents by simply typing their labels into Excel files. However, the **Event Coder** program makes this process more systematic and less error prone, and therefore (I hope) less painful.

This manual gives an in-depth introduction to the program and its functionality. In the remainder of the introduction, I offer some additional background to the program, and I introduce some necessary disclaimers. Consider reading these disclaimers before you send me angry emails. In chapter 2 I go into some of the assumptions that the program builds on (e.g., about the data sets that you will code), and what preparations (i.e., things you are assumed to have done before using the program) this implies. In chapter 3 I go through some of the basic features of the program that do not involve qualitative coding itself, such as importing data, and saving and loading files. In chapter 4 I explain the coding features of the program. I conclude the main body of the manual with chapter 5, discussing some of the things that one could do with the data that the program exports. My contact details can be found in chapter 6.

1.2   Part of a bigger program

It is very important for the reader to realise that, even though I wrote the **Event Coder** as a standalone program, I wrote it primarily with the intention to later integrate it, as a module, in a larger program. The idea is that, in the larger program, you would be able to do everything from entering data into your data set, coding the data in various ways, doing basic forms of analysis, creating visualisations, and exporting date files that can be easily imported

---

am using textual sources (e.g., documents, web pages, interview transcripts), I also like to add the "raw" text on which my incident description was based.

into other useful software. The current (standalone) version of the program still requires you to (1) create your data set with external spreadsheet software (e.g., LibreOffice Calc or Excel), (2) use external software to further visualise and analyse the coded data.

And here is a very important thing to consider before you start using the current version of the program: I would advice you to at least skim through the chapter 5 first. In this chapter, I offer some examples of how the data that the program exports can be imported into other software, and what possibilities this offers. Making use of this program probably only makes sense if these possibilities are actually interesting for your work.

Please, also see section 3.6 to get a good idea of what kind of data the program actually exports. The motivations for the choices that I might in this are perhaps not immediately obvious to everyone, and probably for some they will not become obvious until I actually get to integrating my work in the larger program in have in mind. However, one thing I can say is that I have a strong preference for storing, visualising, and analysing event data (and related data) in some type of graph format. This can be achieved most efficiently, if data are eventually stored either as **nodes**, **relationships between those nodes**, or **properties of nodes or relationships**. This is the reason why the data that the program exports take the form of (primarily) node lists and edge lists, which would allow you to import and visualise the data into various software tools for network visualisation [4].

1.3   Other disclaimers

Throughout the introduction, I already snuck in a few disclaimers, such as the fact that my work on this particular (standalone) version of the program will probably stop once I get to integrating it as a module into a larger program. There are several additional disclaimers I would like to make here.

It is very important that the user realises that I wrote this program, in the first place, for personal use, and that I only make it available for free in case other people may find it helpful in their work. However, you will use the program at your own risk. I do **not** take any responsibility for problems that you run into when using the program. That being said, I am always open to receiving comments, positive or negative, about problems that may occur, bugs you encounter, features you would like to be included, and so on. If you do indeed run into problems as a result of using this program, I am willing to help you think of a solution. You can find my contact details in chapter 6 of this manual.

The user should also realise that I am **not** a professional programmer. With some help from the Internet, I taught myself how to code to keep my mind

---

[4]see, for example, https://gephi.org, which is one of my favourites.

occupied during some of the lonely evening hours I spent in an campus apartment somewhere in Shenyang. I never had any formal training in writing code, and I probably make a lot of ugly mistakes, write inefficient code, and overlook simple solutions for the various problems I face while writing code. While I always try to improve my coding skills, this requires a lot of time, energy, and verbal abuse of my computers[5]. This means that I will not always have the time, energy and/or the skills required to fix certain bugs, add new features, and etcetera.

I also do **not** have a team of beta-testers to help me test the program. You are basically it. Congratulations! If I had T-shirts, I would ship you one to give you some recognition, but unfortunately I have no budget for that. The program is still fairly simple in structure, and I tried to get rid of most bugs by doing my own testing. However, the program is already complex enough for me to overlook things, so there is a good possibility to several annoying bugs remain. The only way to find these, sadly, is to encounter them while using the program. Fortunately for you, if there is one thing that I really hate, it is having bugs in my programs. If you do encounter a bug, please contact me (see chapter 6) ASAP, and I (or we) will try to figure out what is wrong, so that I can fix it.

---

[5]If you happen to be someone with more experience at writing code, feel free to go to my Github page (see chapter6) to inspect the source code of this program, and if you have the time and patience, please offer me suggestions on how to improve my skills.

Chapter 2

Preparations

2.1 It is all about the data

As mentioned in section 1.2, the **Event Coder** program is designed to be eventually integrated into a larger program. In that larger program, data would be entered into, and managed by the program directly. For now, however, the **Event Coder** program needs to import all data from external sources, that is, files created by the user before using the program. The program can be understood to make some assumptions about the nature of these files, and if these assumptions are not met while reading files the program may not work as it should.

Fortunately, creating files that can be read by the program is not difficult. Moreover, the ways these files should be created follow logically from the kind of research approach that this program is designed to fit into. In this chapter, I go into all things that the user should think of well before using this program, preferably in the stage before data collection. Most of these details have to do with the kind of data sets that the program "expects" you to work with.

2.2 Data sets

The program expects that you will import data from what I will call an *event data set*, which is nothing more then a table of chronologically ordered *incidents*. The idea of having such data sets, and the concept of incidents are both based on ideas developed during the Minnesota Innovation Research Program[1]. An incident is a bracketed, qualitative description of an observed action or interaction, which includes, at least, the following information:

1. An indication of the time at which the incident occurred.

---

[1]see especially Poole, M. S., Van De Ven, A. H., Dooley, K., & Holmes, M. E. (2000). *Organizational Change and Innovation Processes: Theory and Methods for Research.* Oxford: Oxford University Press. I was not involved in this work in any way.

2. A brief (qualitative) description of the action or interaction, including a description of the actors/entities responsible for the action.

3. A reference to the source of data.

This is still a very open description of what an incident actually can contain. As a social scientists, I am typically mostly interested in *human activity*, that is, in actions and interactions performed by human beings. That, in itself, is still a very broad category, and it leaves open the question at what level of abstraction an given action can and/or should be described[2]. We may also understand the concept of action in a much broader sense, such that we also include natural occurrences, such as volcanic eruptions, an apple that falls from a tree, or the wind that blows. In the end, what makes sense to count as an action will depend on your research questions, your theoretical orientation, and probably on who you count among your favourite philosophers.

Fortunately, the ***Event Coder*** program does not care about your definition of an action. However, whatever definition you decide to work with, the program does care about how exactly you store information on that action as incidents in your data sets. In the type of event data sets that we work with here, the very first row of data is always the **header** of the file, which contains the names of the columns in the data file (see figure 2.1). All the remaining rows of the data set represent individual incidents. The three aspects of incidents that we mentioned above (timing, description and source) should be stored in separate columns. You may label and position these columns however you like, but we expect these columns to be there.

You may also add additional columns to your data set, to expand on the information about incidents that the data set records. For example, I made a habit of also including fragments of text from my original sources from which I derived my incident descriptions. In my opinion, this makes the process of coding incidents both easier, and more transparent.

Another important assumption that the ***Event Coder*** program makes about your data set is that it is chronologically ordered. That means that the incident reported in a given row is assumed to have happened after the incident preceding it (of course, with the exception of the first incident in the data set). Even if two or more incidents actually happened simultaneously, based on their position in the data set, the program will always assume that there is an order in their occurrence. In practice, this should not be a serious problem.

These are basically all the assumptions that the program makes about the overall structure of the file. However, there are some other practical considerations to be taken into account. The most important one is that the ***Event***

---

[2]One of the my other sources of inspiration is the work of Peter Abell on Comparative Narratives. Abell's work engages more or less directly with questions of how to describe activity at different levels of abstraction. See especially: Abell, P. (1987). *The syntax of social life: the theory and method of comparative narratives.* Oxford, Angleterre: Clarendon.

Figure 2.1: The expected structure of event data sets.

Coder program can only import so-called *comma-delimited files* (I will refer to them as csv files), which can be recognised by their "*.csv" extension (e.g., "My_Dataset.csv"). These files are very much like plain text files ("*.txt"), but they use so-called *delimiters* (most often commas or semicolons) to distinguish between different columns. You can open csv files in a basic text editor to see what this looks like (see figure 2.2).

Figure 2.2: What a csv file looks like in a basic text editor.

```
Timing,Description,Source
10:00,"The owners lets his dog out of the house, to play in the yard.",My imagination
10:01,The dog enters the yard at his house.,My imagination
10:03,The dog spots 5 cats that are sitting in the yard.,My imagination
10:03,One of the cats (cat 1) meows.,My imagination
10:04,The dog barks at the 5 cats.,My imagination
10:04,One of the cats (cat 2) hisses at the dog.,My imagination
10:05,The dog barks at cat 2.,My imagination
10:05,The dog starts chasing the cats.,My imagination
```

Spreadsheet software, like Excel or LibreOffice Calc, typically allows you to store tables as csv files, using the **"Save As"** option[3]. Some software, like Excel, will use a delimiter that is set "system-wide". If you are unsure what delimiter your system uses, just open your csv file with a text editor, and see what characters are used to separate columns. You will need to know what delimiter your files use to be able to import them properly. Other software, like LibreOffice Calc, allows you to choose what delimiter to use when you save the csv file.

So, you can create your data set using in any spreadsheet software that you

---

[3]I have encountered a few people who misunderstood how this works, and who simply tried to convert files by changing their extension. For example, they would rename a file called "My_Dataset.xls" to "My_Dataset.csv". This will not work, because it will not actually change the file itself. It is more likely that you will just render your file unreadable for most software, because the software will think it is reading a csv file, while it is really reading an xls file that is only disguised as a csv file.

prefer. You can also store your data set in any format that you prefer while building it. The important thing is that you should be able to store it as a simple csv file as soon as your are ready to import it into the **_Event Coder_** program. For example, while working on my data sets, I typically store my data in ods files or xls files, but as soon as I want to start coding my data, I will use the **"Save as"** feature of whatever program I am working with to store a csv file.

Other notes

There is a little bit more going on in csv files than what is shown in a regular text editor. What figure 2.2 does not show, for example, is that there are also hidden characters present in the file that tell the computer where each line of data ends (so-called newline symbols: \n). You usually do not have to worry about the presence of such characters, with some small exceptions.

The **_Event Coder_** program is not able to read csv files that have so-called newline symbols (\n) or carriage return symbols (\r) *within* their text cells. The reason for this is that the program uses a relatively simple csv file parser, which will think that a new line of data will start after encountering one of these symbols (each line of data in a csv file will end with a newline symbol by default). Fortunately, inserting such symbols into the text cells will only happen if you deliberately create them, or accidentally paste them into your file.

The program is typically able to recognise when an 'illegal' newline is encountered, and it will throw an error (see figure 3.2). Solving the error is left to the user. The problem can be solved by removing all newline symbols and carriage return symbols from the csv file. These symbols are not visible in programs like Excel or LibreOffice Calc, and in both programs you will need to use special search and replace options to get rid of the unwanted symbols. I advise you to Google for "Find and replace regular expressions with [your spreadsheet program]".

Chapter 3

Using the program: Basic features

3.1 Loading a new dataset

Importing the new data into the program works as follows. You first need to select the csv file containing your data. For this you will click the **Select File** button (see figure 3.1), which will open a file dialog that you can use to navigate to, and select the file.

Once a file has been selected, you will need to select the delimiter symbol that is used in the csv file to distinguish between different columns of the data table. For this, you can use the dropdown menu that reads **-Select delimiter-** by default. Four different symbols are allowed as delimiter, which are the comma (,), the semicolon (;), the colon (:), and the vertical bar (|). Make sure that the delimiter that you select matches the one used in the file.

Once you have selected a delimiter, you can import the data, using the **Import data** button. Once you click this button, the program will attempt to read data from selected file and, if successful, enable all other options of the program, allowing you to start coding.

Problems when importing data

If you (1) selected a valid csv file, (2) selected the correct delimiter for this file, and (3) structured your data set using instructions offered in section 2.2, you should encounter no problems when importing the data. If something goes wrong when importing data, then the problem will usually lie with one of these three points.

One possibility is that you have not selected a valid csv file. I have encountered a few people that have tried to create csv files from (for example) xls-files by simply changing the file extension. Doing this will not actually create a valid csv file that can be read by the program. The correct way for creating csv files is to use the **Save as** option in your spreadsheet editor, and select to save the file with the ***.csv** extension.

Figure 3.1: Options to import data.
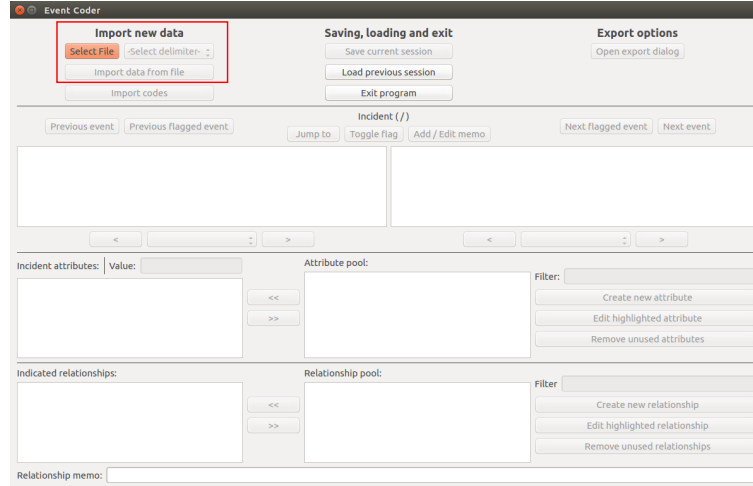
If you selected the wrong delimiter, the program will usually import the data, but it will fail to distinguish between different columns of the data set, and possibly assume that the entire dataset only contains one column. This should be obvious from the texts displayed by the program. In this case, simply import the data again, using the correct delimiter.

If you see the error message displayed in figure 3.2, this means that some cells of your data set probably contain newline symbols and/or carriage return symbols that need to be removed before importing data (see section 2.2).

Figure 3.2: Data import error report.
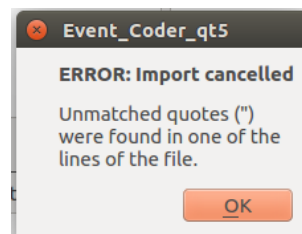
If you are certain that you made no mistakes in one of these points, and you still encounter problems when importing your data set, then you may have encountered a bug in the program that needs to be fixed. In that case, please get in touch with me (see chapter 6).

3.2 Saving and loading data

Coding a data set typically will take a long time, which is why the program allows you to save your progress, and to load the saved session at another moment. Saving data can be done by clicking the **Save current session** button (see figure 3.3). A file dialog will appear, asking you to select a location to store the file, as well as a name for the file. The files will always be saved with the ".sav" extension.

If you want to load a previously stored session, click the **Load previous session** button (see figure 3.3). A file dialog will appear, allowing you to navigate to, and select the file that you wish to load.

Figure 3.3: Saving and loading files.
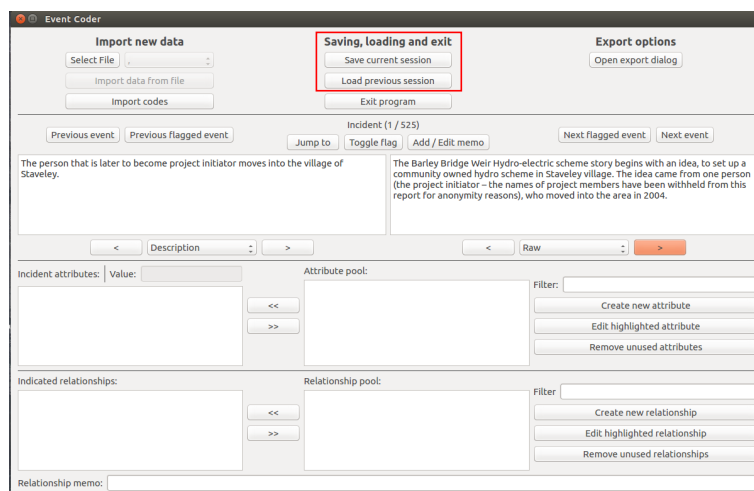
3.3 Importing existing codes

Coding data is typically an iterative process, and it is possible that, during the coding process, the user makes changes to the data set being coded, for example, by adding new rows of data, by adding new columns of data, or by changing the contents of data cells. The program therefore allows the user to import existing codes from an old version of a given data set into a new version of the same data set.

The procedure for importing codes involves the following steps:

1. The user should first make sure that the codes assigned to the **old version** of the data set are stored, by using the **Save session option** (see section 3.2). At a later step, we will import the codes from this save file. For this example, we refer to this files as **Saved_Codes.sav**.

Figure 3.4: Importing codes.
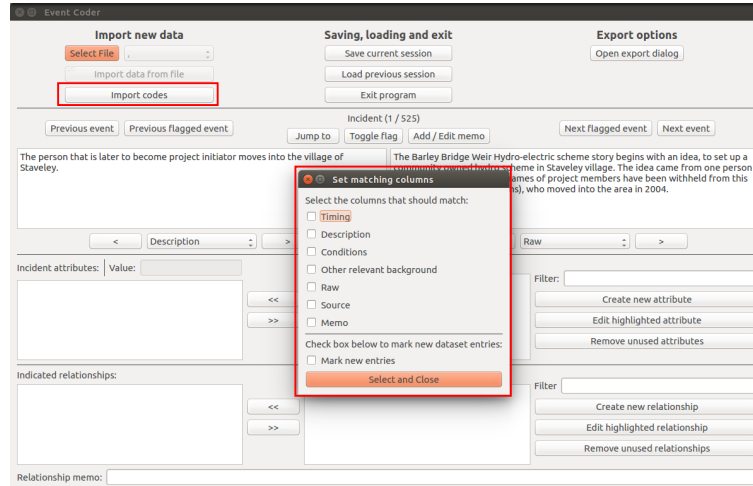
2. After saving the codes assigned to the **old version** of the data set, we can import the **new version** of the data set, using the procedure described in section 3.1. Thus, the steps taken here are the same as when you would start coding a completely new data set.

3. After the **new version** of the dataset has been loaded, you should click the **Import codes** button (see figure 3.4). You will first be shown a warning dialog, just to make sure that you can double check what you are doing. If you select **Ok** in the warning dialog, you will be shown a file dialog. Use this dialog to find the file with your saved codes (**Saved_Codes.sav** in this example). Select and open this file.

4. A new dialog will appear, the specific contents of which will depend on the contents of your data sets. An example is shown in figure 3.4, but yours will probably look different. The dialog will show the columns that the old version of your data set and the new version of your data set have in common. Here, you need to select those columns that the program can use to match entries in the **old version** of the data set with entries in the **new version** of the data set. The program will look at the data in the selected columns of both files. If it finds a row of data in the **new version** of the data set and a row of data in the **old version** of the data set that have the exact same contents in the selected columns, then the program will decide that these rows of data are the same. The program will then ensure that all the codes that are somehow associated with the row of data in the **old version** of the data set, are also associated with the corresponding row of data in the **new version** of the data set. For this to work correctly, the rows of data in both data sets need to be unique. For example, if you select only one column, which

has the same contents in multiple rows in one or both of the data sets, then the program will not be able to decide which codes are associated with which row of data. **In this case the codes will probably be assigned to the new data set erroneously, without warning![1] It is therefore very important that you always select a combination of columns that allows the program to identify each row of data individually.** Usually, it should be enough to select the column that indicates the timing of the incident, and the column that describes what happened in that incident, because having two incidents with the exact same timing, and the exact same contents (description) would probably never make sense, because they would describe the same (inter)action, making one of them redundant. See figure 3.5 for a schematic overview that may help to understand the process.

5. You also have the option to have the program mark any new entries in the **new version** of the data set. This means that, while checking the columns you have selected, the program will also remember any rows of data in the **new version** of the data set that it could not match to any rows of data in the **old version** of the data set. These entries will be marked (see 3.4), so that you can easily find them while coding the data.

6. If you happened to have changed data in one of the rows of the data set, that is, the row was already present in the **old version** of the data set, but you changed some of the contents of that row of data in the **new version** of the data set (in one of the selected columns), that row of data will be treated as new entry, and any codes assigned to that entry will be lost.

7. The program should now return to the main screen, and any attribute codes and relationship codes that were present in the save file that you loaded should now also be available in the current session. Moreover, any incidents that already had codes assigned to them in that save file should now also have those codes assigned to them in the current session.

Problems when importing codes

There are several things that could go wrong when importing codes from an old data set into a new one. One problem would occur if you do not select any columns to be imported in the appropriate dialog. In this case the program will simply report that no columns were selected, and take no further action. Another similar problem would occur if you try to import codes from a data set that has now columns in common with the data set that is loaded into your current session. In this case the dialog where columns can be selected will be empty (except for the option to mark new entries). If you try to proceed, the

---

[1]In a later version of the program I will attempt to prevent this from happening altogether.

Figure 3.5: Schematic overview of matching of data set entries.
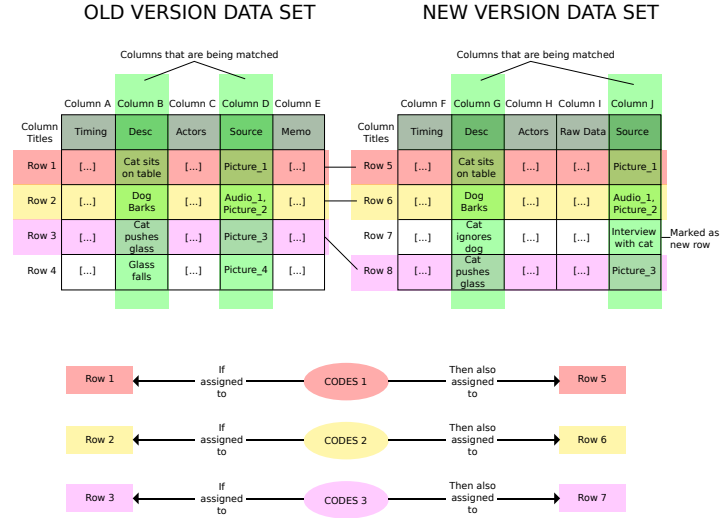
program will behave as if no columns were selected (see above), report an error, and take no further action.

Another problem may be that some of the codes assigned to entries in the **old version** of the data set do not get imported into the **new version** of the data set, even though these entries appear in both. This should only happen if something changed in the contents of these entries (and only in the contents of those columns that the program tries to match). Even if you made only small changes in the contents of the selected columns, the program will treat the corresponding entry as a new, uncoded entry.

A more difficult problem will occur if you do not select columns that allow the program to identify each row of data in the **old version** and/or the **new version** of your data set individually (see the box below for an explanation).


Example to illustrate problem when importing non-unique rows

First, imagine that you select two columns (C_1 and C_2), the contents of which need to be matched by the program. Second, imagine that in the **old version** of the data set, there exist two incidents (two rows in the data set; I_1^{old} and I_2^{old}) that have identical data in those columns. In addition, imagine that the sets of codes (S_1 and S_2) that you assigned to these two incidents are different ($S_1 \rightarrow I_1^{old}$ and $S_2 \rightarrow I_2^{old}$).

When comparing the two versions of the data set, the program encounters one of these rows of data I_1^{old}. If it also encounters a row of data in the

---

new version of the data set that has the exact same data in C_1 and C_2 (let us call this row I_x^{new}), it will think it has found a match, and assign the corresponding codes to the corresponding entry in the **new version** of the data set ($S_1 \rightarrow I_1^{old}$ becomes $S_1 \rightarrow I_x^{new}$). However, the program then proceeds, and it will also encounter the other row of data I_2^{old}, the contents of which (in columns C_1 and C_2) also match with those of I_x^{new}. As a result, the program will simply overwrite the codes that already existed ($S_1 \rightarrow I_x^{new}$ gets overwritten by $S_2 \rightarrow I_x^{new}$). More importantly, the program will not recognise that something has gone wrong, so it will do this work silently, without reporting any error (also see figure 3.6).

Figure 3.6: Codes that are silently overwritten due to multiple matches.
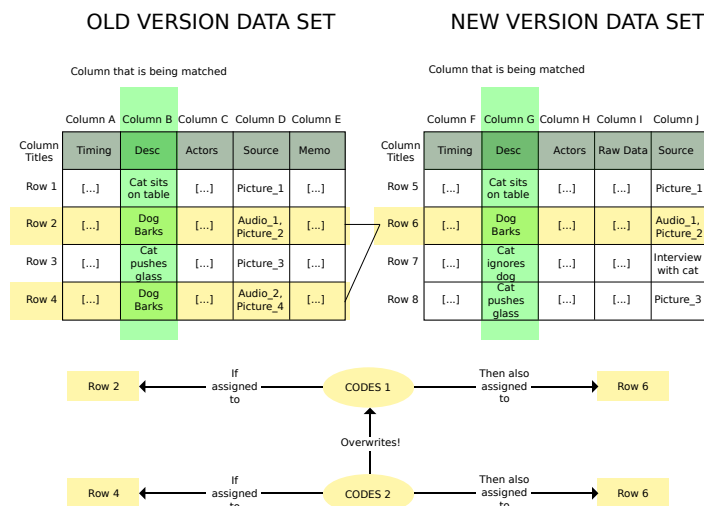
To prevent this problem from occurring, you must always make sure that, based on the columns that you selected in the import dialog, the program will never find two rows of data that are exactly the same. As explained in section 3.3, it should typically be enough to select (1) a column that indicates the timing of the incident, and (2) a column that contains the qualitative description of the incident itself (see section 2.2 for the assumptions that the program makes about how your data set is structured). It is unlikely that two incidents have the exact same contents in these two columns (because then they would simply refer to the exact same incident).

3.4   Navigating through the data

The program, to some extent, enforces a specific way to walk through your data set. As explained in section 2.2, the program assumes that your data are chronologically ordered, with the incidents that occurred earliest at the top, and the incidents that occurred latest at the bottom. When you start coding a new data set, the program will always assume that you wish to start the coding process with the earliest incident. The program will present to you one incident at a time, showing you details on the currently selected incident in up to two fields (see figure 3.7).

Figure 3.7: Overview of incident navigation section.
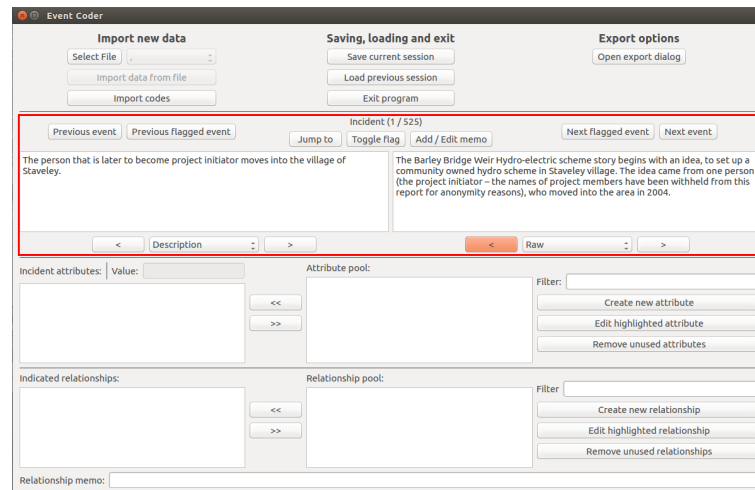
In the two fields, the program can display the contents of any of the columns of your original data set. Below each field, you will find buttons that you can use to choose which column of the original data set to display in the field (the first column of data is selected by default for both fields). The arrow buttons will go to the previous or next column, and the drop-down menu can be used to simply jump to a specific column.

Typically, you will want to assign codes to the incidents based on the information provided in one or more columns of data, such as the description of the (inter)action that the incident captures, and possibly the raw text (from the sources of data) on which this description is based. The program can show up to two columns of data, which, in my own experience, is a nice balance between having a good overview, and not having to process too much information at a time.

Typically, you will create new codes on the fly (see chapter 4), assign appropriate codes to the current incident, and then move on to the next incident. Above the fields with the information on the current incident, you will see an indica-

tor of which incident is currently selected ("**Incident ([current]/[total])**"). To navigate the incidents, use the buttons above the fields with the texts (see figure 3.7). These buttons are pretty straightforward. The **Previous incident** button navigates to the previous incident in the data set. If the currently selected incident is the first in the data set, clicking this button will let you jump to the last incident in the data set. The **Next incident** button lets you navigate to the next incident in the data set. If the currently selected incident is the last incident in the data set, clicking this button will let you jump to the first incident in the data set.

You will also see a button called **Jump to**. Clicking this button will open a small dialog that allows you to type the index number of the incident that you want to jump to. If you type an 'illegal' number (e.g., a number below 1, or a number that exceeds that total number of incidents in the data set) nothing will happen.

Marking incidents

In the area demarcated in figure 3.7, you will also find three buttons that refer to flagged incidents (**Previous flagged incident**, **Next flagged incident**, and **Toggle flag**). Flags can be used to mark incidents that you would like to return to later. If you click the **Toggle flag** button, an exclamation mark will appear next to the index to show that this incident is currently flagged (see figure 3.8).

Figure 3.8: Indication of flagged incident.
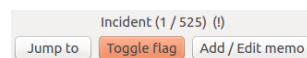
If you later want to return to your flagged incident, you can use the **Previous flagged incident** or the **Next flagged incident** button. These buttons work similar to the **Previous incident** and the **Next incident** buttons, but they will skip all incidents that are not flagged. This should allow you to relatively easily find incidents that you flagged earlier.

Automatically flagging new incidents when importing codes

As described in section 3.3, when you import codes from a previously stored session into a new version of a data set, you will also have the option to mark any new incidents. This means that the program will automatically flag all incidents that are in the **new version** of the data set that you are currently coding, but that were not in the **old version** of the data set from which you are importing the codes. This allows you to always easily find the incidents that have not received any codes before.

Adding memos to incidents

In any coding process it usually helps to write down your thoughts while coding, so that other people can, for example, read your motivations for assigning certain codes to a certain incident. Moreover, when coding larger data sets, you will often forget yourself why you made certain coding decisions. Typically, you would write down any relevant thoughts in the form of memos. The program allows you to associate memos with particular incidents. The memo associated with the current incident can be accessed by clicking the **Add / Edit memo** button. Clicking this button will open a small dialog, which consists out of a text field, where you can write your memo, and one button that allows you to save and close the memo dialog. If you want to add a memo, or edit an existing one, simply write the text you want to add, and then click the **Save and close** button. If you want to remove a memo you wrote earlier, simply delete all the text in that memo, and click the **Save and close** button. Each incident will have its own memo.

3.5   Filtering

In various dialogs of the program, you will find **filter fields**. As you are using the program, and you are enthusiastically adding new attributes, categories, entities, relationships, and so on, it will soon become a nuisance to find your way through all the objects you have created. That is why I created the **filter fields**.

Filter fields work very simple. As soon as you type letters in such a field, the program will filter out objects from the corresponding lists that do not contain that letter in their label. If you add more letters to the filter, creating a string, then the program will filter out all objects that do not contain that string in their label. For an example of a filter field in action, see figure 3.9. The filters are case sensitive (i.e., 'e' and 'E' are treated as two different cases).
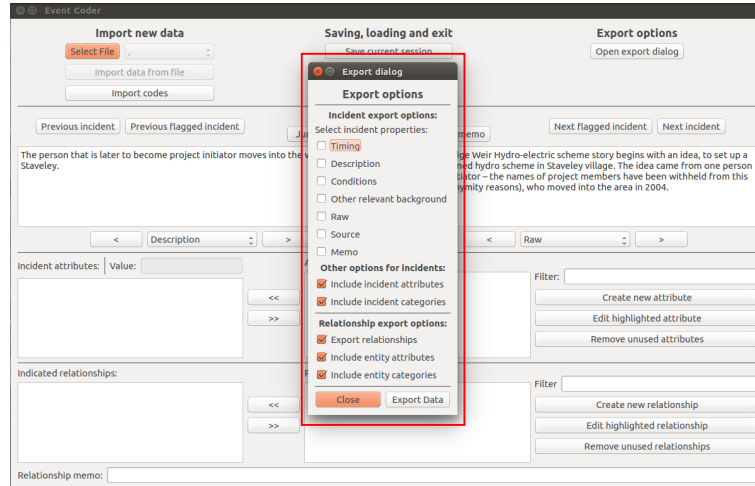
Figure 3.9: Filtering attributes.
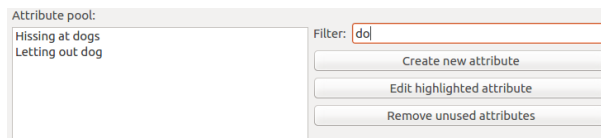
3.6   Exporting data

In chapter 4 I explain in depth how the coding program itself works. Once you have coded a data set, and you want to export the results, you can click the **Open export dialog** button. This will open a new dialog, where you can select several export options (see figure 3.10).

Figure 3.10: The export dialog.

The first set of options, under the header **Incident export options**, allows you to assign properties to your incidents. These properties are simply the contents of the various columns of the data set that you imported. In fact, if you selected all the properties, using the corresponding tick boxes, one of the files exported will simply be a copy of the data set that you imported, with the only difference that every incident will be assigned a unique ID, numbered from 1 to N, where N is the total number of incidents in your data set. These incidents, and any properties that you select form them, will be written to a file that is called **"Incidents_Nodes.csv"**. As the name of the file suggests, this is a nodes file, which in this case is structured such that it can be immediately imported into Gephi, my favourite network visualisation program[2]. Why it is useful to export the incidents and their properties in a node list will be clarified further below.

By default, the program will also export (1) all attributes that you have assigned to incidents, (2) all categories that you have assigned to incident attributes, (3) all relationships that you have assigned to incidents, (4) all attributes that you have assigned to entities in relationships, and (5) all categories that you have assigned to entity attributes. You can deselect any of these options, although categories can only be exported if their corresponding attributes are also exported. Depending on what options you select, the following will be exported:

1. *If you select to **Include incident attributes**, then three additional files will be created*: **"Incident_Attributes_Nodes.csv"**, a node list with

---

[2]See https://gephi.org. I offer no instructions on how to import data into Gephi here. Various guides for that are available elsewhere. Indeed, the data can also be imported into other software, although you might have to make small changes for that to work.

the details (labels and descriptions) of all attributes that have been assigned to incidents, **"Incident_Attributes_to_Incidents_Edges.csv"**, an edge list that records which attributes have been assigned to which incidents (if values have been assigned, these will be recorded in the edge list as well), and **"Incident_Attributes_Matrix.csv"**, a matrix with incidents in the rows and attributes in the columns, showing a 0 in cells corresponding with incidents and attributes that are not associated, and showing a 1 in cells corresponding with incidents and attributes that have been associated. Alternatively, if a value has been assigned to an attribute, then that value will be reported in the cell, instead of a 1.

2. *If you also select to **Include incident categories**, then two additional files will be created*: **"Incident_Categories_Nodes.csv"**, a node list with the details (labels and descriptions) of all categories that have been assigned to incident attributes, **"Incident_Attributes_to_Categories_Edges.csv"**, an edge list that records which incident attributes belong to which categories.

3. *If you select to **Export relationships**, then five additional files will be created*: **"Relationships_Nodes.csv"**, a node list with the details (labels and memos) of all relationships that have been assigned to incidents, **"Incidents_to_Relationships_Edges.csv"**, an edge list that records which incidents are indicators for which relationships, **"Entities_Nodes.csv"**, a node list with the details (labels and descriptions) of all entities that are in one or more relationships, **"Entities_Edges.csv"**, and edge list that records which entities are in which type of relationship with each other, and **"Entities_to_Relationships_Edges.csv"**, an edge list that records which entities are present in which relationships.

4. *If you also select to **Include entity attributes**, then three additional files will be created*: **"Entity_Attributes_Nodes.csv"**, a node list with the details (labels and descriptions) of all attributes that have been assigned to entities, **"Entity_Attributes_to_Entities_Edges.csv"**, an edge list that records which attributes have been assigned to which entities (if values have been assigned, these will be recorded as well), and **"Entity_Attributes_Matrix.csv"**, a matrix with entities in the rows and attributes in the columns, showing a 0 in cells corresponding with entities and attributes that are not associated, and showing a 1 in cells corresponding with entities and attributes that have been associated. Alternatively, if a value has been assigned to an attribute, then that value will be reported in the cell, instead of a 1.

5. *If you also select to **Include entity categories**, then two additional files will be created*: **"Entity_Categories_Nodes.csv"**, a node list with the details (labels and descriptions) of all categories that have been assigned to entity attributes, and **"Entity_Attributes_to_Categories_**

Edges.csv", an edge list that records which entity attributes belong to which categories.

Thus, in total, up to 16 files will thus be exported. These files will all be automatically saved in the **"export/"** folder, which can be found in the folder from which you run the program (if the folder was not already present, a new folder will be created). **Every time you export data, existing files in this folder will be overwritten.** Thus, if you want to keep old files, make sure that you copy them to another folder before exporting data again.

To understand more about what you could possibly do with all these files, please read chapter 5.

3.7 Log files and exiting the program

It will probably come as no surprise to you that the default way to close the program is to click the **Exit program** button. Alternatively, you could close the dialog of the program in the same way you would close any dialog in your OS.

Just before the program closes, it will export a log file to the **"logs"** folder that is located in the folder from which you run the program (if this folder does not exist yet, the program will create it). The log file will be time stamped with the date and time at which it was created. In the log file, you will find a lot of details of various operations that you have performed while using the program, such as navigating to other incidents, assigning or unassigning attributes and/or relationships to incidents, and several other things. These logs are created silently and automatically.

The logs are designed to help you make the coding process more transparent. For example, they may help you retrace your steps if at some point in the coding process you run into some kind of problem, and do not remember how you got there. The log also records which columns of your data set you were expecting when assigning certain attributes or relationships, helping you to remember on what information you based your decision to assign these codes. I do not expect that anyone will be terribly interested in every expecting these log files, but it may be reassuring to know that they are there if you need them.

Chapter 4

Using the program: Coding features

4.1   Introduction to coding

Now we get to the core "stuff" of what the ***Event Coder*** program is all about: Coding the data. In some ways, coding data in this program is similar to what you would do using software packages such as NVivo, Atlas.ti, or MaxQDA. Say, for example, that you would code an interview transcript in Atlas.ti. What you would typically do, is to select a fragment of text in the interview transcript, and then assign a label to this fragment. In the ***Event Coder*** tool we do more or less the same thing, but our fragments of text have already been identified beforehand, in the form of incidents. Another way of looking at this is that creating your event data set (especially its incident descriptions), is already an important part of the coding process[1].

In the ***Event Coder*** program the coding options are divided into two main modules, **Attributes** and **Relationships**, which can be used in various ways, and which do not necessarily have to be used together. In this chapter, I will explain both coding modules in depth, and offer some basic suggestions for how they could be used.

4.2   The Attribute module

We will first focus on the module that can be used to associate attributes with incidents. I decided to use the term attributes in this case, because these entities will generally be used to identify relevant properties of the incidents.

---

[1]And I believe this is a good way of looking at it, because the difficulties you face while deciding where an incident starts and where it ends, and what is a relevant incident in the first place, are quite similar to the difficulties you face while trying to identify the relevant fragments of text in an interview transcript.

Some example of different sorts of attributes that I thought of myself, are as follows:

- **(Inter)action types**: Given that incidents are assumed to capture (inter)actions (see section 2.2), one thing we possibly want to do is to use attributes to identify different types of action. For example, imagine that we have a data set that captures interactions between cats and dogs. Examples of attributes that capture different types of such (inter)actions are *barking*, *meowing*, *chasing cats*, *hissing at dogs*, and so on. We would have a list of such attributes, each describing an individual type of (inter)action, and assign these to incidents where appropriate. The point here is to treat each individual incident is the occurrence of a particular type of (inter)action (see figure 4.1.A).

- **Colligated events**: On some occasions, the researcher using the program might not necessarily be interested in the (inter)actions described by his/her incidents per se, and he/she might instead be more interested in some more abstract event for which the incidents serve as indicators. For example, imagine someone who made a data set that records the interactions between two primitive clans that inhabit roughly the same geographical area. The incidents could in this case be empirical descriptions of the interactions that involve members from both clans. Some of these interactions may refer to occasions where members of the clans exchanged certain objects or services, which one might see as indicative of *trading* between the clans. Other interactions may refer to occasions where members of the clans attack each other, which one might see as indicative of *conflict* between the clans. One could of course assign such interaction types as attributes (see above), but one would thereby overlook that some of the exchanges (described by more than one incident) and some of the attacks (described by more than one incident) refer to the same instances of *trading* or *conflict*. The point here is thus to use attributes to indicate which incidents "group" together. We could, for example, distinguish between different instances of trading by numbering them (see figure 4.1.B). If desirable, we could even make explicit that the different versions of the same attribute (e.g., *Trading_1* and *Trading_2*) belong together by assigning a category called *Trading* to them (see section 4.2).

- **Value changes**: There are approaches to studying social processes that treat each incident inherently as a change. In some cases, one could even want to use values to express by how much something changes. Imagine that you check how much money you currently have on your bank account, and you find (to your surprise) that you are in debt. In a flurry of irrational behaviour, you decide to reconstruct all your incomes and expenses in an event data set, to reconstruct how you got to this point (Maybe it works therapeutic? Who knows?). To capture your

incomes and expenses, you could simply create two attributes: *Money increase* and *Money decrease*, and assign these to incidents in your data set accordingly. As will be explained further on in this section, you have the possibility to assign a value to any attribute that is also assigned to an incident. This value will always be unique to that particular "incident-attribute pair". This allows you to track, over time, how much money went in or out (although values do not necessarily have to be numerical), and by how much (see figure 4.1.B).

Figure 4.1: Three possible uses of attributes.
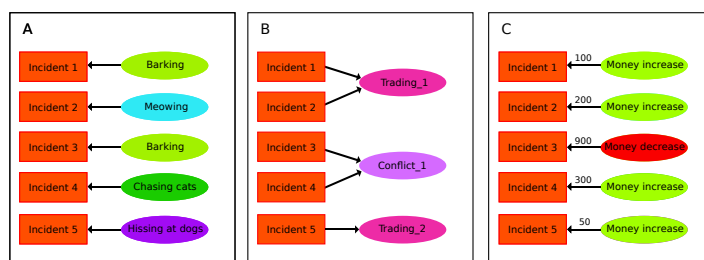
Of course, other uses of attributes are thinkable. One could for example also use attributes to identify the actors that are involved in the activities described in incidents, or to identify the places where the activities described in the incident take place (but also see section 4.3 for an alternative approach to coding for this). I tried to design this module in a way that accommodates different approaches. In reality there are no great technical differences in the way that attributes are associated with incidents in the three different scenarios visualised in figure 4.1. What the program does, in the background, is always more or less the same. Taking different approaches to using attributes is therefore primarily a matter of how the user thinks about them, and how this thinking fits in the overall research design of the user. Of course, the possibilities are not limitless. A lot will depend on how you are actually able to use the attributes after you export them. To read more details about this, please see section 3.6 and chapter 5.

Creating new attributes

If you have just started a new session, then the attributes module will look more or less as illustrated in figure 4.2.

No attributes have been created yet, so the list of (assigned) **Incident attributes** (left) and the (unassigned) **Attribute pool** (right) will be empty. As I will explain in more detail below, attributes that are assigned to the incident will always appear in the left field. The "pool" of attributes from which one may select attributes to assign is shown in the right field.

Figure 4.2: The attribute module.
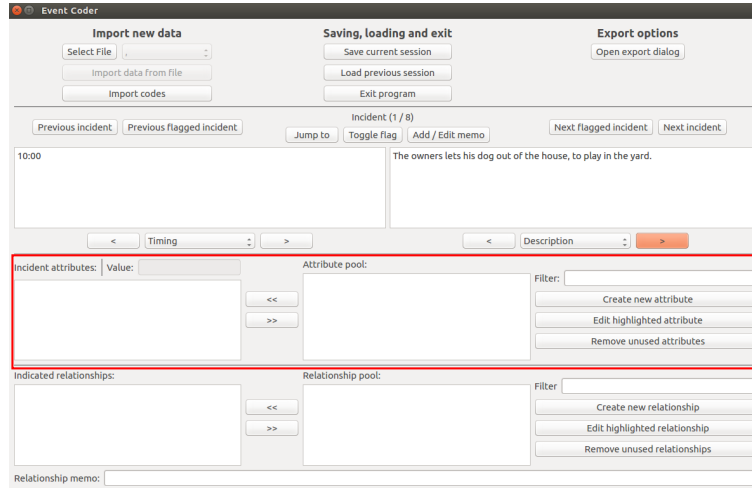
If you wish to assign attributes, you will first need to create them. For this, you have to click the **Create new attribute button** on the right side of the module (see figure 4.2). This will open a new dialog, which looks like the one shown in figure 4.3.

Figure 4.3: The attribute dialog.
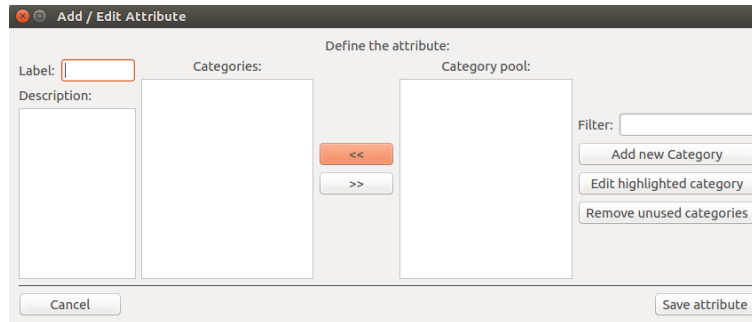
If you want to create a new attribute, there are always at least two types of information you need to provide: (1) each attribute needs to be assigned a *unique* label, and (2) each attribute needs to be assigned a description. If one of these types of information is missing, the program will now allow you to save the new attribute. The **label** is basically the name of the attribute that we will use to refer to it. The label has to be unique, that is, no attributes with identical names are allowed. This is mostly a "fail-safe" that I built in to prevent the user from confusing him/herself. However, it also greatly simplified

much of the processes going on in the background for handling attribute data[2].

In the **description** field you can put anything that you like, but it should generally be used to clearly describe what the attributes stands for, in such a way that other people understand what it is supposed to capture (i.e., a clear definition).

Once you have created a label and a description of the attribute, you can save the new attribute. You can always change any aspects of the attribute later (see below). Optionally, you could also assign categories to attributes, which first have to be created, using a similar procedure (also see below).

If you change your mind about creating the attribute, simply click the cancel button. You will return to the main dialog, without making any changes to the list of available attributes. Any categories that you may have created in the meantime will still exist.

After creating an attribute, you will return to the main dialog, and the new attribute will appear in the **Attribute pool**. This pool contains all available attributes that may be assigned to the current incident. Thus, after creating a new attribute, they will not be assigned to any incident yet. In figure 4.4 we have already created a few attributes. There is another feature that is illustrated in the figure, which is that, if you hover your mouse over any attribute, its description will be shown as a tool tip. I implemented this feature to ensure that you do not have to open the attribute dialog every time that you are wondering what the description of that attribute was again.

Figure 4.4: Some new attributes in the pool.
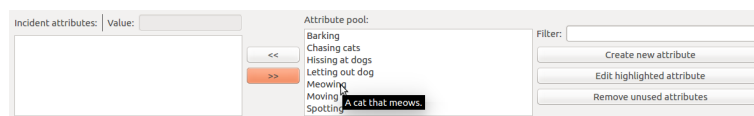
Editing attributes

You may occasionally want to edit the label, description, or categories of an attribute that you created earlier. This can be done easily by selecting the attribute in one of the lists (**Incident attributes** or **Attribute pool**) and then clicking the **Edit highlighted attribute** button. Alternatively, you can simply double click any attribute in one of the lists.

This will open the attribute dialog with the details of the existing attribute already shown. Simply edit the details as you like, and save the changes.

---

[2]So, just in case you are, for some reason, hoping that creating attributes with identical names will ever be possible: It is not going to happen, unless you decide to alter the code yourself, of course.

Removing unused attributes

Removing attributes may not be entirely intuitive. I decided not to allow the user to simply click an attribute, and remove it from the **Attribute pool**. The reason is that the user might have assigned that attribute to another incident, forgotten about this, and thus unintentionally remove the attribute from that incident as well. I therefore assume that one would only ever want to remove attributes that have not been assigned to any incident in the data set. Thus, this is exactly what clicking the **Remove unused attributes button** will achieve. The program will identify all attributes in the **Attribute pool** that have not been assigned to some incident in the data set, and then remove these from the program.

Removing unused attributes is an irreversible action, so if you regret removing some attribute, there is no other option than to create it again, from scratch.

Indeed, a drawback of this approach is that it becomes relatively hard to get rid of attributes that you previously assigned to incidents, but do not want to use anymore. Your only option is to find all the incidents to which the attribute has been assigned, unassign the attribute from all of these incidents, and then use the **Remove unused attributes** option. Indeed, you can make this process less painful by using the following approach:

You could export the data that you have coded so far (see section 3.6), and use the **"Incident_Attributes_to_Incident_Edges.csv"** file to identify the incidents to which the attribute that you want to remove has been assigned. You can then enter the IDs of these incidents in the **Jump to** dialog to quickly navigate to these incidents, where you can unassign the attribute.

Incident attribute categories

By now, I have mentioned several times that it is possible to assign categories to attributes (actually, I like to think of this as assigning attributes to categories, but technically there is not really a great difference). In figure 4.3 you may have noticed that the attribute dialog itself as a list of assigned **Categories**, a **Category pool**, and options to add categories, edit categories, and remove unused categories.

The principles here are the same as with the attributes themselves. Initially, the list of assigned **Categories** and the **Category pool** will both be empty. To create a new category, click the **Add new Category** button. This will open the dialog illustrated in figure 4.5.

As with attributes, you have to give each category a *unique* name, as well as a description in order to be able to save them. If you change your mind about creating a new category, just click the **Cancel button**. Otherwise, after assigning a label and offering a description, click the **Save category** button. To edit a category, select a category in one of the lists (**Categories** or **Category pool**) and click **Edit highlighted category**.

Figure 4.5: The category dialog.
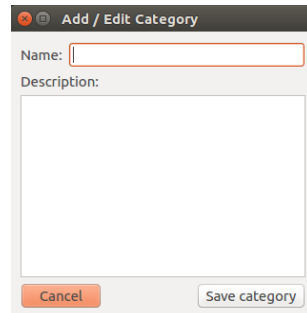
After creating new categories, they will appear in the **Category pool** in the attribute dialog (see figure 4.6). Whenever you are creating a new attribute, or editing an existing attribute, you can assign categories from the **Category pool** by clicking the assign button («). To unassign a category, click the unassign button (»). In figure 4.6, we assigned one category *Dog activities* to the attribute *Barking*. As with attributes, you can hover over the label of any category you created to see the description of that category in a tool tip.

Figure 4.6: Attribute with assigned category.
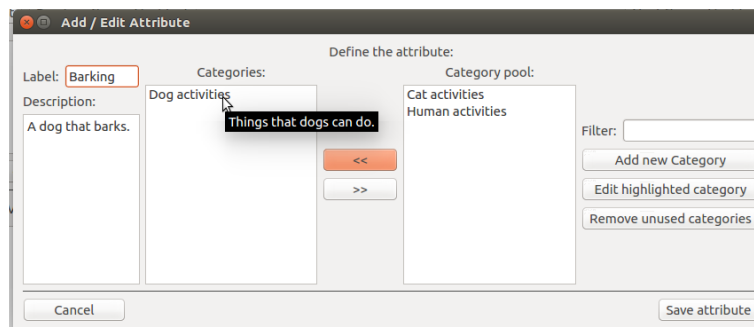
Removing categories also works similar to removing attributes, which means that you can only remove all categories that are not currently in use by clicking the **Removed unused categories** button. If you want to remove a category that you have already assigned to various attributes, you might want to consider the following approach:

Export the data that you have coded so far (see section 3.6), and check the file **"Entity_Attributes_to_Categories_Edges.csv"** to identify the attributes to which the category has been assigned. Then, in the main dialog of the program, use the filter to quickly find these attributes (they could appear in the list of **Incident attributes** or the **Attribute pool**, depending on whether they were assigned or not). Then edit these attributes (e.g., double click their label) and unassign the category you intend to remove from them. After unas-

signing the category from all attributes in this way, remove the category by clicking the **Remove unused categories** button.

Assigning and unassigning attributes

Once you have created attributes, you can start assigning them to incidents. Assigning an attribute to an incident can be done by selecting the attribute in the **Attribute pool**, and then clicking the assign button («). The attribute will move from the **Attribute pool** to the list of **Incident attributes**, and the attribute is now associated with this incident. In figure 4.7 we have assigned one attribute to the current incident, based on the description of the incident to us.

Figure 4.7: Attributes assigned to incident.
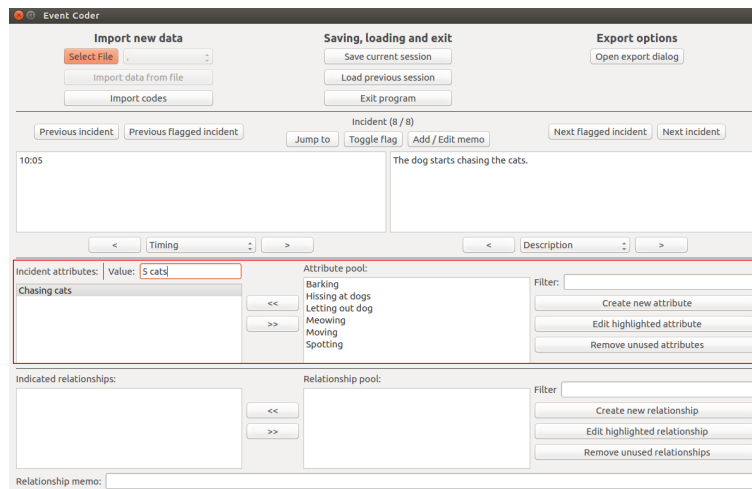
In this case, we have also done something else: we have added a value to the attribute, by clicking on the assigned variable, and then typing "5 cats" in the value field, which is located above the list of **Incident attributes**, to indicate that the dog is chasing 5 cats in total.

If we decide that this value is not important after all, we could just select the attribute again, and delete the value that we entered.

Unassigning attributes is achieved by clicking the attribute in the list of **Incident attributes**, and clicking the unassign button (»). The attribute will move from the list of **Incident attributes** to the **Attribute pool**, and the attribute will no longer be associated with the current incident. Also, if a value was assigned to the attribute, this value will now have been removed. **This action is irreversible**.

And with this we have covered everything important that there is to know about the attribute module of the program!

4.3   The relationship module

The other module that can be used for coding purposes is the relationship module. This module can be found at the bottom of the main dialog of the program (see figure 4.8). Its primary purpose is to identify relationships between entities that are indicated by the incidents under consideration.

Figure 4.8: The relationship module.
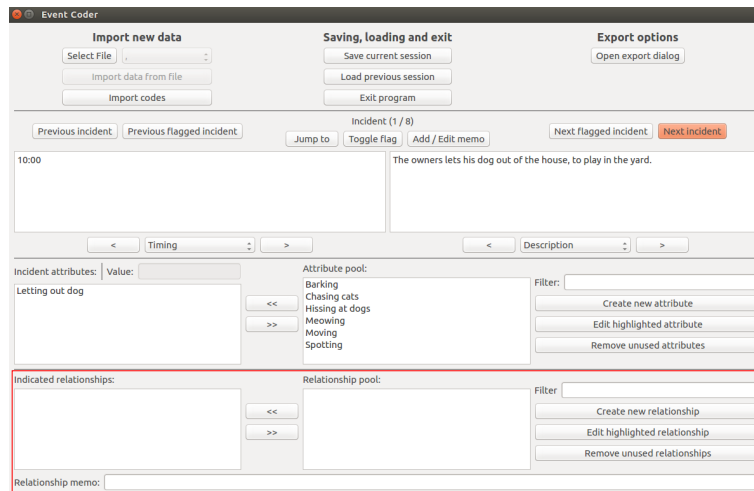
For example, say that we have a set of incidents that describe how a dog encounters a cat in his yard, which it then proceeds to chase out of the yard. There are several relationships that we could possibly refer from this (although for most we would perhaps need a bit more context). For example, we could infer that the dog considers the yard as his territory ($Yard-[is\ territory\ of] \rightarrow Dog$), we could infer that the dog hates the cat ($Dog-[hates] \rightarrow Cat$), and we could infer that the cat is afraid of the dog ($Cat-[is\ afraid\ of] \rightarrow Dog$). These are examples of relationships that could be identified with the relationship module of the program, and the idea is that the incidents are used as evidence for the existence of the relationships.

Creating new relationships - part 1

If you have not yet created relationships, the list of **Indicated relationships** and the **Relationship pool** will both be empty. If you want to assign relationships to the current incident (i.e., identify a relationship that is indicated by the incident), you will first need to create one. This takes a number of steps. The first step is to click the **Create new relationship** button. This will open the relationship dialog (see figure 4.9).

This dialog indeed looks quite different from the attribute dialog. In this case, we have three fields, which can be used to select (from left to right) the **source**

Figure 4.9: The relationship dialog.

node of the relationships, the **type** of the relationship, and the **target node** of the relationship. The **source node** and **target node** are always entities. **Relationship types** have a label, a description, and also a direction, which can be **directed** or **undirected**. In order to be able to create relationships, you first need to create entities and relationship types. We have to discuss this in detail before we can finish our discussion on creating relationships.

Entities

A new entity can be create by clicking the **Define new entity** button. This will open a new dialog (see figure 4.10).

Figure 4.10: The entity dialog.

Similar to attributes, entities always require (1) a *unique* name, and (2) a description. The name will be used throughout the program to refer to the entity. The description should be used to offer a more detailed description of what or who this entity is. Optionally, you can also assign attributes to entities. **Assigning attributes to entities works the exact same as assigning attributes to incidents. I therefore do not explain this procedure here, but refer you to section 4.2**. The program does, of course, make a distinction between attributes (and their categories) associated with incidents, and attributes (and their categories) associated with entities, but they are created, edited, removed, assigned and unassigned in the exact same way.

Once you have created a new entity, it will appear in the **source node** selection field, and the **target node** selection field (see figure 4.12).

As with the attributes and categories, editing entities in these lists can be done by selecting them, and clicking the **Edit highlighted entity** button (also see section 4.3). This will open an entity dialog with the corresponding information already included.

As with attributes and categories, it is only possible to remove entities that are not currently in use, which in this case means that you can only remove entities that are not participating in one of the relationships that you have already defined (whether that relationship itself has been assigned to an incident or not).

To remove unused entities, click the **Remove unused entities** button. Before doing this, you might first want to remove unused relationships (see section 4.3). This should also free up any entities that were assigned to unused relationships.

Relationship types

To create a new relationship type, click the **Define new relationship type** button. This will open the relationship type dialog (see figure 4.11).

Figure 4.11: Relationship type dialog.

As with attributes, entities and categories, new relationship types can only be saved if you have assigned both a label and a description to them. The label is used to refer to the relationship types throughout the program, and the description should be used to offer a clear definition of the relationship type. In the case of relationship types you also need to indicate the direction of the relationship, which is set to **Directed** by default. In a direction relationship, the relationship is always directed from the **source node** to the **target node** (e.g., $source - [likes] \rightarrow target$). In an **undirected** relationship, there is no direction, and the **source node** and the **target node** are interchangeable (e.g., $source \leftarrow [talks\ with] \rightarrow target$)[3].

Once you have created a new relationship type, it will appear in the **Relationship type** selection field (see figure 4.12).

As with the attributes, categories, and entities, editing relationship types in these lists can be done by selecting them, and clicking the **Edit highlighted relationship type** button (also see section 4.3). This will open a relationship type dialog with the corresponding information already included.

As with attributes, categories an entities, it is only possible to remove relationship types that are not currently in use, which in this case means that you can only remove relationship types that are not participating in one of the relationships that you have defined earlier (whether that relationship itself has assigned to an incident or not).

To remove unused relationship types, click the **Remove unused relationship types** button. Before doing this, you might first want to remove unused relationships (see section 4.3). This should also free up any relationship types that were assigned to unused relationships.

Creating new relationships - part 2

Now that we have created a few entities and relationship types (see figure 4.12), we have the basic ingredients required to define a relationship.

To create a relationship, we need to select (1) a **source node**, (2) a **relationship type**, and (3) a **target node**. Selecting a node or relationship type can be done by clicking the corresponding label in the the appropriate list, and then clicking the **use selected** button (this button will only activate if you have selected an appropriate item). Once you have selected a entity or relationship type, it will disappear from the list. This also means that entities can never have a relationship to themselves. In the middle of the dialog, a schematic overview is offered of the nodes and relationship type that the user has selected (see figure 4.13). Note that the schematic overview also indicates whether the relationship type is a directed or not. This schematic form is also used as the label for relationships, as we will soon see.

---

[3]For this reason, the program will treat two undirected relationships as identical even if they mirror each other. For example, the program will think that $A \leftarrow [talks\ with] \rightarrow B$ is identical to $B \leftarrow [talks\ with] \rightarrow A$.

Figure 4.12: The relationship dialog with entities and relationship types available for selection.

Figure 4.13: The relationship dialog with entities and a relationship type selected.

Once you have created one or more relationships, they will appear in the **Relationship pool** in the main dialog (see figure 4.14).

Editing relationships

Editing a relationship after it is created works similar to editing attributes, categories, entities or relationship types. Simply select the relationship in the list, and then click the **Edit highlighted relationship** button. This will open the relationship dialog, but with the corresponding **source node**, **relationship type** and **target node** already selected.

It is possible that you also want to edit properties of one of the selected entities, or of the relationship type after you have selected them. To edit one of the selected entities, click the **Edit left assigned entity** button, or the **Edit**

Figure 4.14: Some new relationships in the pool.

right assigned entity button. To edit the selected relationship type, click the **Edit assigned relationship type** button.

Removing relationships

As with all other objects we have discussed so far (and for the same reasons), it is only possible to remove relationships that are not currently in use. Unused relationships can be removed by clicking the **Remove unused relationships** button.

If there is a particular relationship that you want to remove, but it has already been assigned to one or more incidents, then I advice you to use the following approach: Export the data that you have coded so far, and open the **"Incidents_to_Relationships_Edges.csv"** to see to which incidents this relationship has been assigned. Then use the **Jump to** dialog to jump to the indexes of the corresponding incidents, such that you can unassign the relationship you want to remove from these incidents (use the filter field to find the relationship more quickly).

4.4   Assigning and unassigning relationships

Once you have created relationships, you can start assigning them to incidents. This can be done by selecting the relationship in the **Relationship pool**, and then clicking the assign button («). The relationship will move from the **Relationship pool** to the list of **Indicated relationships**, which means that we now consider the current incident as an indicator of the relationship. In figure 4.15 we have assigned two relationships to the current incident, based on the description of the incident.

In this case, we have also done something else: we have written a memo about the relationship. These memos are created using the bottom-most field in the main dialog. I chose to allow the user to edit these memos here, because I assumed that memos about relationships would be created primarily in the context of how they relate to the data. All you have to do to create a memo for a relationship is to select the relationship in the list **Indicated relationships** or in the **Relationship pool**, and then type the memo in the field. There is no need to save the memo in any way. Everything you type is recorded instantly. If you want to remove a memo, simply delete the text.

The relationship memos thus are similar to attribute values. However, unlike attribute values, relationship memos can also be written for relationships that are not currently assigned, because the memo is for the relationship in general, and not in relation to the specific incident that is currently selected[4].

Figure 4.15: Relationships assigned to incident.
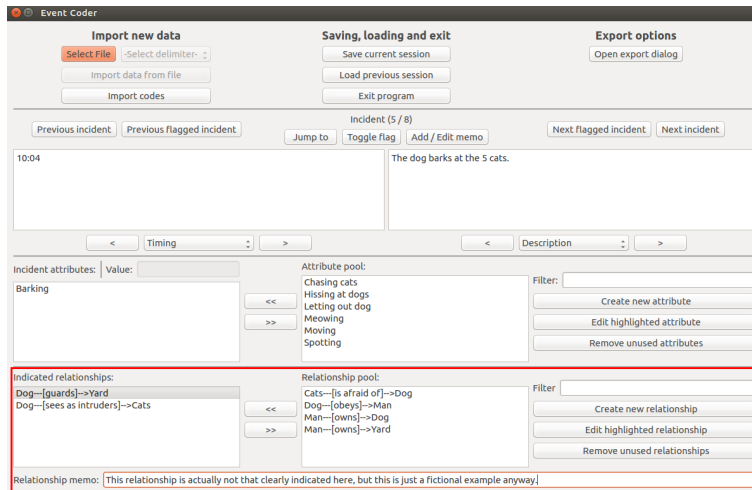
You can unassign a relationship by selecting it in the list of **Indicated relationships**, and then clicking the unassign button (»). The relationship will move from the list of **Indicated relationships** to the **Relationship pool**, and the current incident will no longer serve as an indicator for this relationship. The memo that was assigned to the relationship will remain intact.

And that covers all important things you need to know about the relationship module. We have now covered everything you need to know about using the program itself. In the next chapter I discuss a few things that you could potentially do with the data that you have coded.

---

[4]This is just a design choice that I made, because I think the memos will be more useful this way.

Chapter 5

What is next?

So, you have coded a data set (see chapter 4), and you have exported your coded data (see section 3.6), resulting in a bunch of files in the **"export/"** folder. What can we actually do with those files now? I think there are many possibilities, and I will not explore all of these here, but let me mention at least a few[1].

5.1   Graph databases

Lately, I have been experimenting a lot with the use of graph databases, which are, in very simple terms, databases in which all data are either stored as **nodes**, as **relationships between nodes**, or as **properties** of nodes or relationships. For me, storing data this way is ideal, because I usually like to present event data in some "networked form", and with graph databases there are very few steps required between storing data and presenting data in such a form. As discussed in section 1.2, I intend to integrate the *Event Coder* program in a larger program at some point. My intention is for this program to interface directly with a graph database, without the user ever having to worry about how this works. For now, however, the data that the *Event Coder* program exports have to be imported into a graph database manually. I will not offer a detailed example of how to do this here, because it is well beyond the scope of this manual[2].

Figure 5.1 shows a screenshot of a some of our coded data after I have imported it into a Neo4j graph data base[3]). This screenshot actually gives a pretty good picture of the kind of data structures that my programs will eventually aim for. Some of you may wonder why I do not simply use a tabular structure for our data. One good reason to use graph structures to structure your data, is that it is relatively easy to convert graph structured data in tabular form, while

---

[1]If you have ideas that you would like to share, please contact me (see chapter 6)!

[2]Read as: "I really do not feel like working out such an example right now."

[3]See https://neo4j.com/.

the graph form allows you to store objects and their relationships in a much more efficient way. For example, say that I want to create a simple table of our entities and their attributes. This is can be achieved relatively easily (see figure 5.2 for an example of such a table produced with Neo4j).

Figure 5.1: Some of our coded data stored in a graph database.

Figure 5.2: Our table of entities and their attributes.

This does not actually tell you anything that you could do with these data in terms of analysis, but I hope it does make clear that it creates a lot of flexibility in how you store, present, and structure your data. Once you manage to put your data into a graph structure, it is easy to convert your data in the various forms required by different software packages.

5.2 Visualising relationships

We do not actually have to first import our coded data into a graph database before we can make interesting visualisations. Some people may only be interested in creating an overview of which attributes are associated with which incidents, or an overview of all the relationships that are indicated by the incidents that we coded. Such overviews can be easily created with network visualisation software. I have mentioned several times that my favourite network

visualisation tool is Gephi[4]. I have therefore structured many of the exported files such that you can immediately import them into Gephi. The visualisation illustrated in figure 5.3 was created in a few minutes. I simply imported the file **"Entity_Nodes.csv"** as a node list, and the file **"Entity_Edges.csv"** as an edge list from Gephi's data laboratory, created a simple layout with the Gephi's Force Atlas 2 layout, changed the colour of the nodes, made sure that the labels of the edges show up, and then exported the result, after fine-tuning the visualisation in Gephi's preview window.

Figure 5.3: Visualising the relationships between entities.
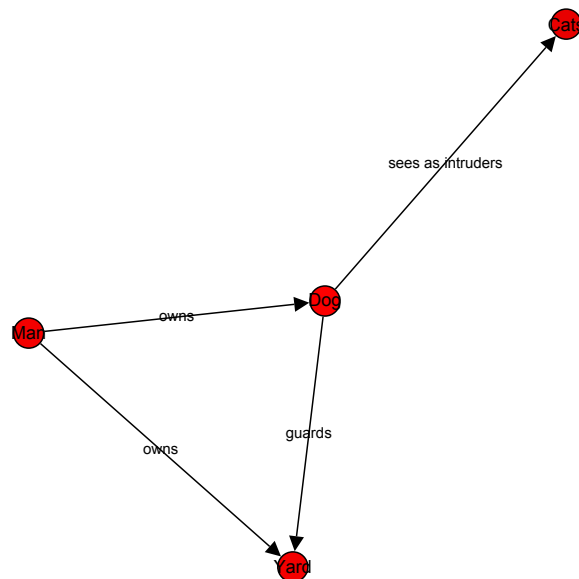
You can use this procedure with most of the files that the ***Event Coder*** program exports. Eventually, you could just reproduce the type of graphs that we saw in figure 5.1, although Gephi was originally not intended to be used as a graph database[5].

---

[4]See https://gephi.org/.

[5]There is a plugin for Gephi that allows you to use Gephi as an interface for a Neo4j graph database. At the time I was writing this manual, this plugin was not yet updated for

5.3 Creating event graphs

Another program that I have recently created is the ***Linkage Coder*** program. This program imports the exact same types of data sets as the ***Event Coder*** program, but its purpose is different: Rather than being a tool for qualifyign incidents (with attributes and relationships), the ***Linkage Coder*** program can be used to identify relationships *between* incidents. For example, certain actions in a given data set may have occurred *in response to* other actions in that data set. The ***Linkage Coder*** program is designed to identify such linkages and make them explicit (through qualitative coding).

For me, the ***Event Coder*** program and the ***Linkage Coder*** program are intimately related. In fact, I plan to integrate both of them in a larger program in the future (see section 1.2). I use the joint output of the programs to create *event graphs*. These are graphs in which the nodes represent events, and the edges represent relationships between events[6]. The ***Linkage Coder*** program helps me to create the basic structure of event graphs, and the ***Event Coder*** program helps me to qualify the events, for example, by identifying what types of actions these events represent.

Figure 5.4 shows a visualisation of an event graph that I recently used in a presentation. In this graph, the events from left to right based on the order of their occurrence. The colours represent different types of interactions (something we could identify with the ***Event Coder*** program), and the edges represent which interactions occurred in response to which other interactions (something we could identify with the ***Linkage Coder*** program).

Figure 5.4: Example of event graph.
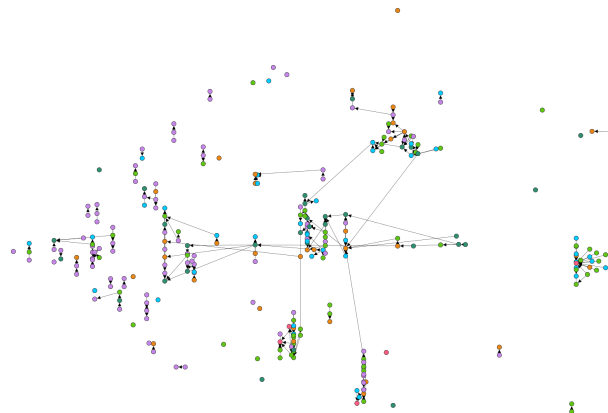
the latest version of Gephi.

[6]For an example of a publication in which event graphs are used, see Spekkink, W. A. H., & Boons, F. A. A. (2016). The Emergence of Collaborations. *Journal of Public Administration Research and Theory*, 26(4), 613âĂŞ630.

5.4 Future examples

This is it for now. I plan to add more examples in the future, as my own work (using this program) progresses.

Chapter 6

Contact details

At the time of writing this manual, I work as a Research Associate at the Sustainable Consumption Institute (SCI) of the University of Manchester.

Wouter Spekkink
The Sustainable Consumption Institute
The University of Manchester
188 Waterloo Place, Oxford Road
M13 9PL, Manchester
The United Kingdom

Email address (office): wouter.spekkink@manchester.ac.uk
Email address (personal): wouterspekkink@gmail.com

Website: http://www.wouterspekkink.org
Github page (where I upload source code): https://github.com/WouterSpekkink