# – Implementation

## Introduction

The game flow analysis deserves particular attention for serious games (ABT, 1987), which are games used for purposes other than entertainment while still providing pleasure. Serious games have been used for aiding students to learn and understand concepts taught in classrooms (BAKER; NAVARRO; VAN DER HOEK, 2003; NAVARRO, 2006; KOHWALTER; CLUA; MURTA, 2011) due to their characteristic of stimulating curiosity and for providing motivation for learning (PRENSKY, 2001). Understanding the educational results obtained in a serious game is important to assimilate the knowledge and concepts passed in the game. In addition, examining the game flow allows for the identification of good and bad attitudes made by the player or by game developers. This knowledge can be used in future game sessions to avoid making the same mistakes or even to adjust gameplay features.

In this chapter, the *provenance in games* conceptual framework is instantiated in the *Software Development Manager* (SDM) game (KOHWALTER; CLUA; MURTA, 2011) as a proof of concept. The SDM game focuses on introducing Software Engineering concepts and skills to undergraduate students. The version of SDM presented in this chapter includes the provenance gathering, allowing students to visualize their actions and identify steps that lead to successful or unsuccessful outcomes. A visualization tool for the provenance graph, named *Proof Viewer*, is also described in this chapter. *Proof Viewer* was customized to be compatible with SDM. However, it can support other games with few modifications on the interface, the filters, which use SDM nomenclature, and the vertices, which are also customized for SDM.

This chapter is organized as follows: Section 5.2 briefly describes the SDM and details of how the provenance information is gathered, generating the *game flow log*. Section 5.3 provides a simple example of game session in SDM to be used as examples. Section 5.4 describes details about Proof Viewer. Lastly, Section 5.5 presents the final considerations of this chapter.

## SDM

In SDM, which was developed using the game engine Unity3D (HIGGINS, 2010), the player has a team of employees that are used to develop software according to contracts made with customers. The gameplay and game mechanics are modeled presenting possibilities to the player to decide strategies for development and define the roles for each staff member. As in any contract, the software has requirements that must be followed during development. From a gameplay point of view, these requirements help to balance the mechanics and rules. When the software is completed and delivered to the customer, there is a quality assessment of the software and a project completion payment accordingly to the product quality.

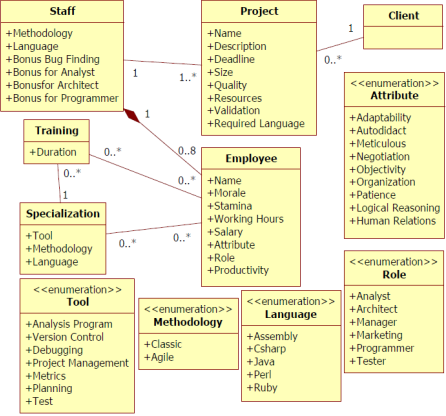


Figure 1: SDM simplified class diagram

Since SDM focuses in people management, the main elements of the game are the employees, which represent the player’s labor force. Employees can perform different roles (analyst, architect, manager, marketing, programmer, and tester.), which uses the employee’s attributes to calculate his performance depending on the role performed. Another element present in the game is specialization, used to define the employee working competence. With the specialization system, it is possible for employees to undergo training to learn new sets of skills. Also the concepts of working hours, morale, and stamina are used to modify the employee’s productivity. Figure 1 shows a simplified version of SDM’s class diagram focusing on the employee, showing his human attributes, types of specializations, the possibility of training to acquire specializations, and that the employee is affect by other employees that belong to the staff team. It also illustrates the project, its characteristics and requirement.

### Provenance Gathering

The information structure used in SDM is similar to the one explained at chapter 4. Each project contains a list of the employees involved in its development. In turn, each employee has a list of his actions executed throughout the development. If any action had an external influence during its execution, then that action will also have a pointer to the action that influenced it. Throughout the game, when actions are executed or new employees are hired, the information of the event is collected and stored for later usage. Actions will go to their respective lists while new employees are added to the project list, creating their own list of actions. Each day passed in the game also records the state of the software development.

Since the information collected is used for the generation of the provenance graph, it contents are mapped to the three possible types of provenance vertex (*activities*, *agents*, and *entities*). This mapping is done according to the data model explained at chapter 4. In the following paragraphs is described the information details extracted from the game and their respective roles in the provenance graph.

Each action executed during the game is represented by an *activity* vertex. The information collected during its execution includes: who executed it, which task and role the employee was occupying, and the current morale and stamina status of the employee that executed the action. The worked hours in the day the action was generated and credits spent to execute the action are also stored. Lastly, the progress made in his task, if any. Besides those, if the action had any external influences, used or altered an artifact (prototypes or test cases), it is stored a link to the action or artifact that affected its execution.

The employees that participated in the development of the software are mapped to *agent* vertex in the provenance graph. The information collected includes the employee’s name, his current staff grade (junior, mid-level, or senior), his level, attributes, and specializations. Lastly, the *entity* vertex in the provenance graph represents one of the three possible artifacts in SDM: Prototypes produced by architects and consumed by analysts; Test Cases produced by analysts, architects, and programmers and consumed by testers; and Project, which represents the instances of the software development progress recorded each day.

The daily project information collected includes the day of its instance, the project’s deadline, how much coding was produced and the code overall quality. It also stores the clients requirements identified and modeled by analysts, how many credits the player had by the end of that day, and the state of each type of bugs in the software. For prototypes and test cases only the day they were created and their names are stored, since actions will contain the information of when they were used. Figure 2 illustrates the information collected in SDM and shown in Proof Viewer according to the vertex’s type.

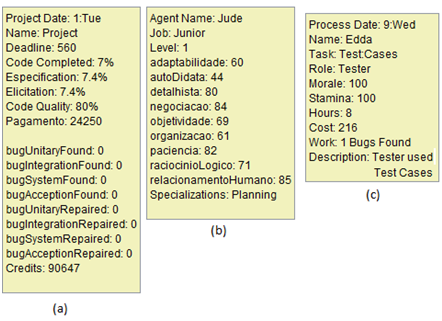


Figure 2: Information data extracted and visible at the provenance graph. The *entity* vertex representation for the project’s data (a), the employee’s *agent* vertex data (b), and the action’s *activity* vertex data (c).

At the end of a gaming session, the collected data during the game session, also known as *game flow log*, is exported to an external visualization tool, the *Proof Viewer*. *Proof Viewer* in turn analyzes the data and generates the corresponding provenance graph that represents the game session. The next section describes a game scenario in SDM that is used as an example for describing the *Proof Viewer*.

## Guiding Example

Starting the game, the player has at his disposal four employees: Yesha, Tornik, Mirax, and Emmy. The first him he does is to assign roles for each employee. Yesha is assigned as the staff’s manager and has the task of aiding analysts. Tornik is assigned as an analyst, Mirax as marketing (which aids analysts and generate cash), and lastly, Emmy is assigned as programmer. Then the player asks Yesha to hire tree new employees: Arden, which is put for training, Marke, an architect, and lastly Daniel, an analyst which will work for 14 hours a day. Almost two weeks passed before finishing Arden’s training and putting him to work as a programmer.

Starting the third week in the game, the player begins to have economic problems, running out of cash. Daniel, due to the extra hours, begins to tire and later demits itself. The game continues with a few rearrangements in task. Later on, Arden and Marke also leave the staff. At the start of the next month, and after receiving cash, the player hires another employee (Miera) as a programmer to replace Arden. At the same week, the player sets Mirax to negotiate with the client, asking to extend the project’s deadline by one more week, since it was about to end. Because of the deadline extension, the staff manages to complete the software in time, delivering it to the client.

At the end of the session, the *game flow log* is generated by using the collected information from the game (employees, actions, and the project daily progression). The examples and illustrations present at the next section are based on the *game flow log* exported from the game session described at this section (the first weeks of the development process).

## Proof Viewer

The *game flow log* is used by *Proof Viewer* to generate a provenance graph corresponding to the game session. In order to do this, *Proof Viewer* analyses the information and interpret it to generate the vertices and edges of the graph, as illustrated by Figure 3. First, the *game flow log* is processed, classifying the information to their corresponding vertex types (*activity*, *entity*, or *agent*), and then generates the edges that link each vertex.

To simplify the first step (processing input file), the information extracted from the game is arranged in pairs, where the pair represents two vertices followed by the edge that links them. As such, in a pre-processing step, *Proof Viewer* generates the vertices and edge every time it processes a pair of vertices. Each time *Proof Viewer* process a vertex, it searches the database to check if the vertex was already processed. If the vertex was previously processed, then *Proof Viewer* will use the processed vertex instead of creating a new one. Otherwise it will create the vertex. This avoids duplicates, since a single vertex can appear multiple times in the file due to the nature of how the file is structured: a vertex, another vertex, and the edge that links them. So, with the vertex had multiple edges connecting it, then it would appear multiple times in the log.

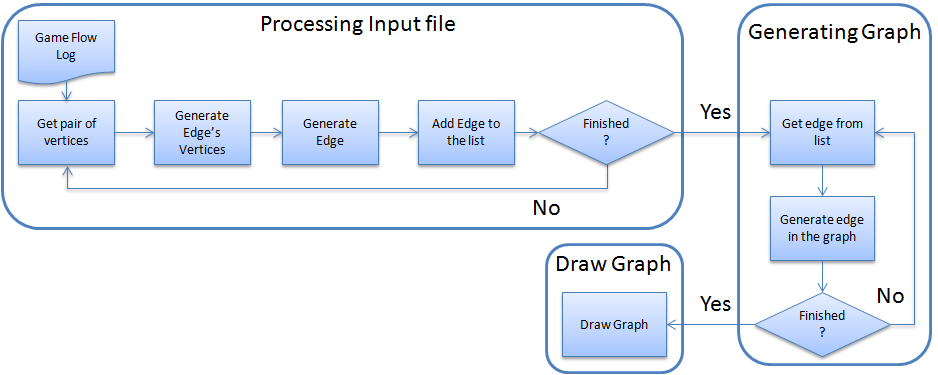


Figure 3: *Proof Viewer* processing the *game flow log* and generating the graph.

After processing both vertices, *Proof Viewer* creates the edge and stores it in a list of edges that are later on used to generate the graph. In *Proof Viewer*, an edge contains pointers to the vertex source, the vertex target, and the edge’s information (value and type). The source and target are the vertices previously processed from the pair. This is done until the entire *game flow log* was processed and all edges were placed in a list of edges that is used to generate the graph. All the information from the *game flow log* is processed in this stage, even if they don’t initially appear in the graph.

As mentioned earlier, vertices can belong to three types: *activities*, *entities*, and *agents*. When generating the file that contains the information extracted from the game, an additional tag is added to distinguish the vertices types. This distinction is used by *Proof Viewer* when generating the vertex in order to generate the correct vertex type, without the need to process the information and decide which vertex it will be according to its characteristics, saving processing time. Note however that the input format can be customized, as long as it generates a list of edges, where each element in the list has the vertex source, the vertex target, and the edge information. Table 1 shows the number of fields each element has in the generated game flow log file, which is a simple txt file, from SDM. The structure of each line in the file is composed of: Tag + Vertex + Vertex + Edge, where each field and element is separated by *tab*.

Table 1: Number of fields for each element type in the input file

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Tag | Vertex: Activity | Vertex: Agent | Vertex: Entity | Edge |
| # fields | 1 | 12 | 15 | 1 | 1 |

The next step is the generation of the graph. *Proof Viewer* uses the generated list of edges, creating each edge in the graph, and consequentially, the vertices from the edges. It is done this way because *Proof Viewer* was based on the JUNG framework (JOSHUA O’MADADHAIN; DANYEL FISHER; TOM NELSON, 2010), where an edge is created by the method *addEdge(source, target, edge)*, which adds the edge in the graph from *source* to *target* with the information *edge*. If *source* and/or *target* are not in the graph, then JUNG automatically generates the vertex. This avoids the need of creating each vertex before creating the edge in the graph, while at the same time checking for duplicates. After creating the graph, it is drawn on the screen.

### Graph Visualization and Representations

After the game flow log is processed and the graph is generated, it is drawn on the screen so the user can see and analyze it. Figure 4 illustrates the graphical user interface (GUI) from *Proof Viewer*, using the generated provenance graph from the Guiding Example. The provenance graph is displayed at the center of the screen but due to the graph size, only a part of it is displayed. However it is possible to change the zoom and navigate the graph. The graph layout is set to be similar to a table, were each “line” represents the activities for each agent and each “column” represents a day in the game. The layout can be customized by creating new layouts or using existing ones available from JUNG. The filters, which are customized to SDM, and other possible functions are located at the lower region of the interface. Starting with the buttons, the first one is “Granularity: 7 days”. This button is only an example of grouping vertices together from the same agent. In this case, it groups vertices from the same week.

The “CollapseAgent” button collapses all the agent’s vertices into him. It can be useful to detect if that agent had any influence throughout the game, instead of looking vertex by vertex. The “Collapse” button allows the user to collapse the selected vertices, while the “Extend” button will remove the last collapse made to generate the selected vertex. The “Reset” button removes all modifications made in the graph, returning it to the original state. The “MainGraph” and “TutorialGraph” buttons change the displayed graph, where the main graph is the graph being analyzed and the tutorial graph is a smaller graph used as example. The “Edge Style” is used to change the edge’s arrow curvature from a quad-curve to straight lines. Lastly, the “Mouse Mode” is for changing the function that the mouse will perform. There are two functions: Transform, which is used to navigate the graph, and Picking, which is used to select vertices in the graph.

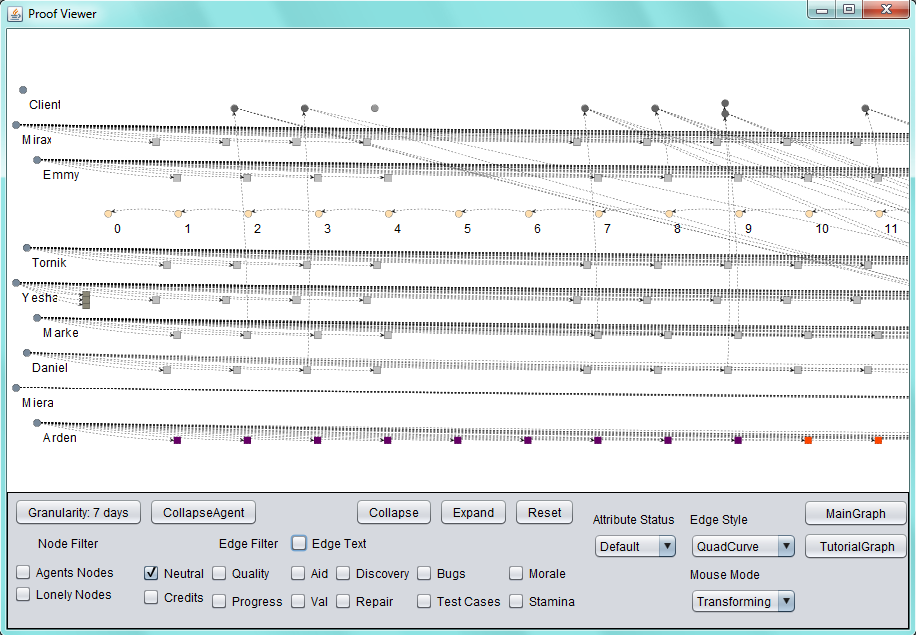


Figure 4: Proof Viewer’s GUI

The next items in the interface are the checkboxes used for filters. Starting with Node filters, the “Agents Nodes” hides all agents in the graph. This is just to illustrate the possibility of hiding vertices by type. In this case, it hides *agent* type vertices. The “Lonely Nodes” hides all vertices that have no edge linking it in the current display. This is useful to clean the graph from vertices that have no edges from the displayed types. The edge filters set the display to only show the selected types of edges. This is done changing the display status of each edge in the graph, displaying only the types selected while hiding the rest. An edge is composed of a value plus type. For example, and edge labeled as *342 validation* has a value of 342 and the type *validation*. From SDM, the edge’s types can be: Credits, Quality, Progress, Aid, Val (short for Validation), Discovery, Repair, Bugs, Test Cases, Morale, and Stamina. Neutral edges are edges that represent association between vertices, while the others represent influences. Black edges are neutral edges, which are also dotted, or edges with a value equal to zero. Note that only the “Neutral” type is selected in the displayed graph from Figure 4. This means that the graph is only showing neutral edges. The graph is set to always start the visualization with only the neutral type selected, pre-filtering all other edges. This is just an example of possible pre-filtering. Any type of filter can be used during the initialization of the graph. This is useful to reduce the graph granularity, hiding information from the user to avoid overwhelming him. The full graph can be seen if all edge’s types are selected, resulting in the section of the graph illustrated by Figure 5. The “Edge Text”, when selected, displays the edge label, containing its value and type. This information can also be seen by moving the cursor to the edge, displaying a tooltip. Vertices details are also available by moving the cursor over it.



Figure 5: Same graph from but with all edges

The edge filter is important because it allows for the identification of types of influences in the graph, filtering other influences that are not relevant for the analysis. For example, at days 10 and 11, employee Daniel had drastic changes in his performance, dropping from *342 validation* to *34 validation*, as shown by FIGURE. This was detected by activating only the “Val” edge filter. The reason for this sudden drop can be traced to Mirax and Yesha by changing the filter to “Aid”, where Yesha provided an aid of 298% in day 10 and a negative aid of 248 at day 11 to Daniel. Mirax provided 227% and 136%. By combining the factors, at day 10 Daniel had 525% bonus in his task from aids, while at day 11 he had -112%. Thoroughly, Daniel productivity without any bonus was 65 at day 10 and 53 at day 11, which is within his productivity margin.

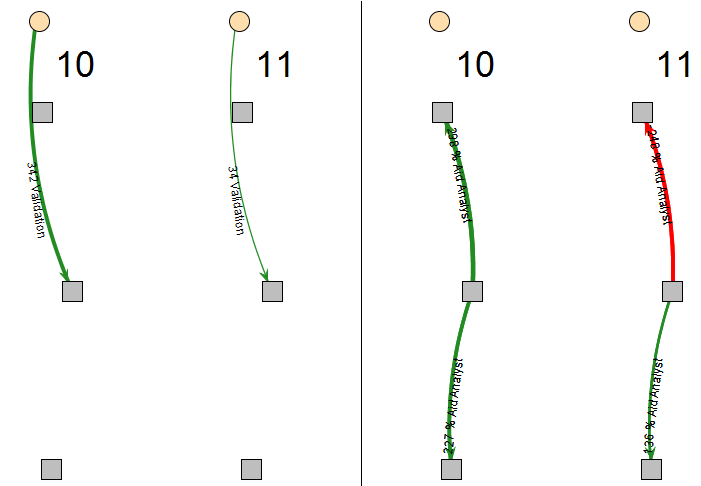


Figure 6: Analyzing Daniel's productivity. Left picture has only "Val" edge filter on. The right picture has with only "Aid" edge filter on. Employees are: Yesha (upper tasks), Daniel (Middle), and Mirax(bottom).

The “Attribute Status” changes the vertex color according to their values from the selected attribute. In SDM they can be: Morale, Stamina, Hours (Short from Working Hours), Weekend (marks vertices from days saturday and sunday), Credits, and Role. The vertex color will not change if it doesn’t have the selected attribute. The default mode colors common activities with a shade of gray and uncommon activities with different colors. Possible uncommon activities, and the color it will have, must be specified. For example, in SDM the uncommon activities are: Idle (red color), Training (purple color), Fired (brown color), Promotion (green color), Hired (Cornsilk color), and Negotiation (Honeydrew color). This color difference between vertices is useful to quickly identity non-ordinary events. For example, by looking at the graph from Figure 7 it is possible to quickly identify that an employee trained for one week and was idle for four days after that. Also that employee Daniel was fired while Mirax was promoted, and that Yesha hired three new employees at the beginning of the game.

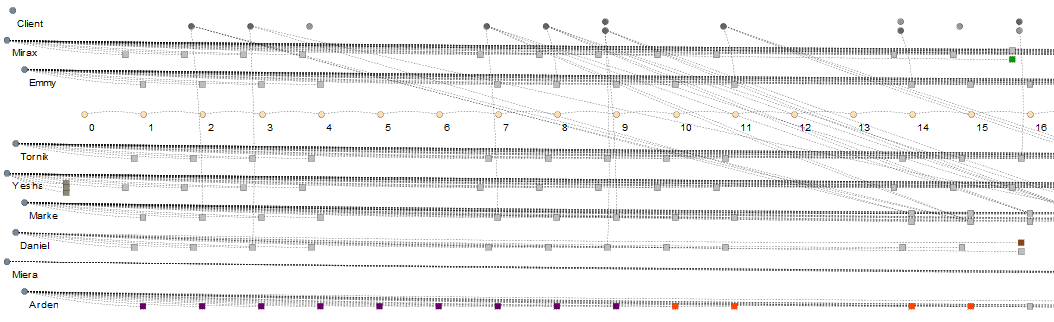


Figure 7: Same graph from with Attribute Status set to default mode.

This type of visualization by evaluating attributes is useful to quickly identify particular sections from the graph. Another example in the same scenario is to check the player’s economy. By changing the visualization to evaluate Credits instead of the default mode, the vertices that have the player’s credits value changes their color according to its status. By looking at Figure 8, it is possible to see that the player’s ran out after day 10. It is also possible to identify the source of this problem by activating the “Credits” edge filter, which is illustrated at Figure 9.

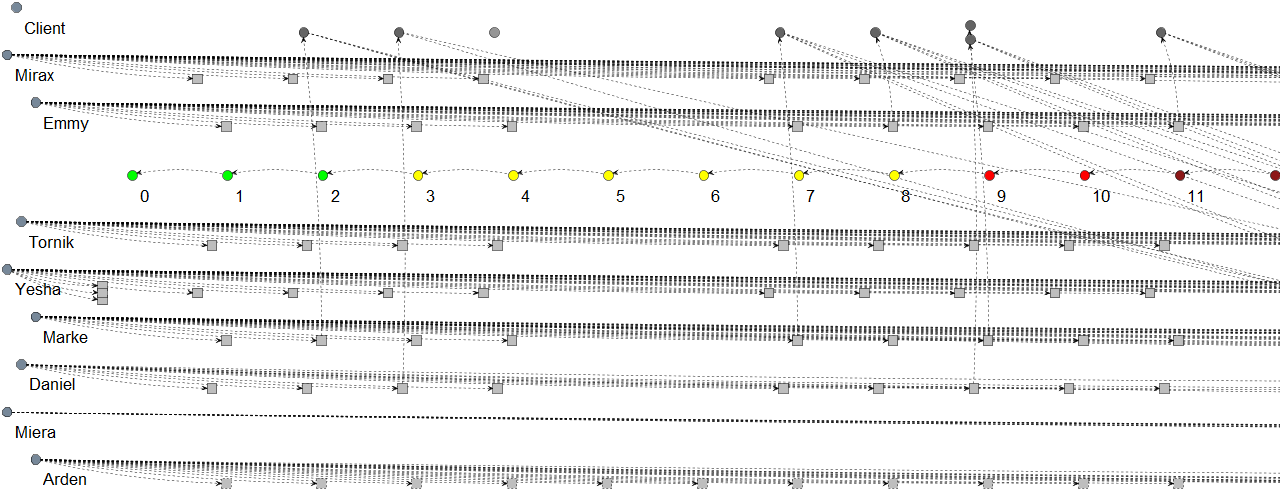


Figure 8: Same graph from Figure 4 with Attribute Status set to Credits mode

As can be seen by Figure 9, the player had many expenses with his employees, more so by hiring three new employees and training another. It is possible to group the vertices together to increase the visibility of the expenses, as illustrated by Figure 10. In total it was spent 24.170 with hiring, 8036 with Arden, 0 with Miera because she was not in the game yet, 3240 with Daniel, 1971 with Marke, 1899 with Yesha, 1809 with Tornick, and 2007 with Emmy. Mirax actually gained 6840 credits for the player by exerting her role as marketing, which generates cash and aid analysts.



Figure 9: Same graph from Figure 4 with Attribute Status set to default mode and Credits edge filter on

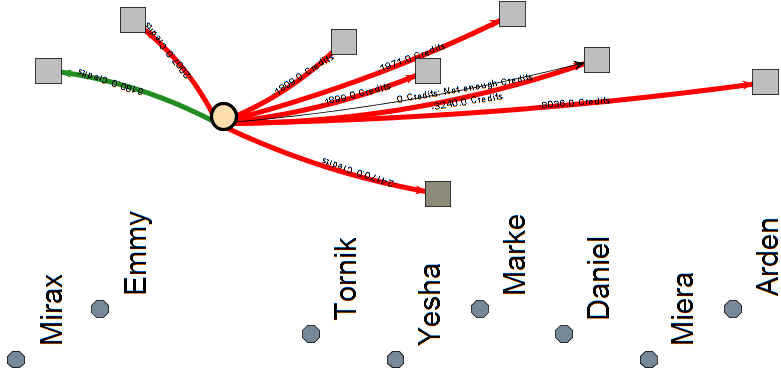


Figure 10: Same graph from Figure 4 with Attribute Status set to default mode, Credit edge filter on, and vertices collapsed (from day 0 to day 11). Figure was rotated by 90º to the left and the edge’s labels were enabled.

## Final Considerations

In this chapter was discussed THIS AND THAT.

In summary, the process from reading the *game flow log* and generating the graph is: Firstly, *Proof Viewer* reads the *game flow log* and gets the first pair of vertices and the edge information, if any. Then it classifies the vertices according to their types and generates them. The next stage it generates the edge, with the source and target being the vertices previously generated and the information contained that came along with the pair. Then it adds the generated edge to the list and checks if there are any more vertices pairs to be processed. Repeat until the entire *game flow log* is processed. After creating all edges and storing them in the list, it proceeds to the next stage, which is the graph generation. For each edge in the list, create the edge in the graph. After all edges, and consequentially vertices, were created in the graph, *Proof Viewer* draws the graph.

Even though Proof Viewer was customized for SDM, it can be modified to work with other games. Most of the resources present in it were designed to work independently from the game by using templates and abstract classes.

Input file to XML.

# References

ABT, Clark C. *Serious Games*. 1. ed. Abt Books: University Press of America, 1987.

BAKER, Alex; NAVARRO, Emily; VAN DER HOEK, André. Problems and Programmers: An Educational Software Engineering Card Game. *International Conference on Software Engineering(ICSE)*, p. 614–621, 2003.

HIGGINS, T. *Unity - 3D Game Engine*. Disponível em: <http://unity3d.com/>. Acesso em: 5 maio 2011.

JOSHUA O’MADADHAIN; DANYEL FISHER; TOM NELSON. *JUNG: Java Universal Network/Graph Framework*. sourceforge: Open-source, 2010. Disponível em: <http://jung.sourceforge.net/>.

KOHWALTER, Troy; CLUA, Esteban; MURTA, Leonardo. SDM – An Educational Game for Software Engineering. In: 2011 X BRAZILIAN SYMPOSIUM ON GAMES AND DIGITAL ENTERTAINMENT (SBGAMES), nov. 2011, Salvador. *Anais*... Salvador: In: X SBGames, nov. 2011.

NAVARRO, Emily. *Simse: a software engineering simulation environment for software process education*. 2006. California State University at Long Beach, Long Beach, CA, USA, 2006.

PRENSKY, Marc. Fun, Play and Games: What Makes Games Engaging. *Digital Game-Based Learning*, p. 1–31, 2001.