Lappeenrannan teknillinen yliopisto
School of Energy Systems
Energiatekniikan koulutusohjelma
BH10A0200 Energiatekniikan kandidaatintyö ja seminaari

VISUALIZATION OF THERMAL-HYDRAULIC NODES

Työn ohjaaja: Teemu Turunen-Saaresti
Espoo 15.12.2015
Tatu Hovi

ABSTRACT

Tatu Hovi

Visualization of thermal-hydraulic nodes

School of Energy Systems

Bachelor's Thesis 2015.

30 pages, 16 figures and 1 table.

Key words: Visualization, Thermal-hydraulic nodes, Python, SMABRE, VVER

The purpose of this thesis was to develop a program that can illustrate thermal-hydraulic node dimensions used in SMABRE simulations. These created node illustrations are used to verify the correctness of the designed simulation model and in addition they can be included in scientific reports.

This thesis will include theory about SMABRE and relevant programs that were used to achieve the ending results. This thesis will give explanations for different modules that were created and used in the finished program, and it will present the different problems encountered and provide the solutions. The most important objective in this thesis is to display the results of generic VVER-1000 node dimensions and verify the correctness in the displayed part. The finished program was created using code language Python.

CONTENTS

LIST OF SYMBOLS

APROS	Advanced Process Simulation Software
CLI	Command-Line Interface
GUI	Graphical User Interface
GPL	General Public License
NPP	Nuclear Power Plant
SBLOCA	Small Break Loss of Coolant Accident
SMABRE	SMAll BREak
SVG	Scalable Vector Graphics
VTT	Technical Research Center of Finland
VVER	Vodo-Vodjanoi Energetitšeski Reaktor
W3C	World Wide Web Consortium
XML	Extensible Markup Language

1. INTRODUCTION

VTT (Technical research center of Finland) uses 1-Dimensional thermal-hydraulic codes such as SMABRE and APROS for modeling extensive and complex flow systems. For example these complex flow systems can be cooling circuits of nuclear power plants (NPP). The prepared models that are created by using these codes are usually very large and the designer of the model has to input immense amount of numerical input data. This input data includes lengths, heights, areas of piping, angles, etc. If the prepared model is extensive there can easily be minor or even bigger mistakes when inputting the data mentioned. These mistakes might be as simple as comma mistakes, causing invalid values for certain inputs. These mistakes are very hard to spot because neither APROS nor SMABRE has any way to illustrate specific node dimensions of the prepared model.

The task of this thesis was to develop a program that can generate these node dimensions automatically into SVG-files (Scalable Vector Graphics). Node dimensions are created by using the information found in SMABRE and APROS input/output-files. The objective for this program was to achieve fast and reliable way to ensure that the node dimensions are correct. In addition the created SVG-files could be included in reports and scientific publications for illustrating the used node dimensions.

This thesis will include theory about SMABRE and relevant programs that were used to achieve the ending results. This thesis will give explanations for different modules that were created and used in the finished program, and it will present the different problems encountered and provide the solutions. The most important objective in this thesis is to display the results of generic NPP VVER-1000 (Vodo-Vodjanoi Energetitšeski Reaktor) node dimensions and verify the correctness in the displayed part.

The finished program was created using code language Python. This thesis has been done for VTT’s nuclear safety department.

2. SMABRE

In NPP the thermal hydraulics mean the coolant's behavior in the reactor. Coolant's main purpose is to transfer energy from the core to the turbine or to heat removal systems. Simulations for thermal hydraulics have existed for over 40 years in nuclear applications. The development of such system code SMABRE, standing for SMAll BREak was initiated because there was need to simulate SBLOCA (Small break loss of coolant accident) in VVER-440 analyses. The previous attempt to simulate VVER-440 SBLOCA with system code RELAP4-Mod6 had failed and that acted as a main spark for the developing of system code SMABRE. (Miettinen 2000, 1-3)

SMABRE is very small in size and faster when comparing to the other system codes such as RELAP5 (U.S), ATHLET (Germany), CATHARE (France). SMABRE's physics contains around 150 equations covered by overall 35 000 program lines. For comparison, RELAP5 contains more than 600 equations and more or less about 200 000 program lines. (Miettinen 2000, 6)

The development of SMABRE started in 1980 at the VTT. First few years yielded no different versions because the code was not so diverse. After some diversification version indexing was defined. Since 1984, the versions exist separately; they can be compared against each other. (Miettinen 2000,12)

SMABRE belongs to the category of five-equation thermal hydraulics models with drift flux phase separation. The phase separation describes the existing gravitational phase separation very well during SBLOCA. When solving void fraction, enthalpy and pressure distributions, sparse matrix inversions are used. In addition to phase conservation equations, the dissolved boron in water and the fractions of non condensables in gas are solved. (Miettinen 2000, 22-23).

SMABRE is typically connected with another system code with parallel coupling. For example, such code can be TRAB-3D. TRAB-3D is a reactor dynamics code with three-dimensional neutronics and one-dimensional thermal hydraulics. Parallel coupling makes it possible to calculate the core thermal hydraulics by both codes in parallel where TRAB-3D typically describes all the assemblies with individual channels, and the coarse code SMABRE core hydraulics with fewer channels. Development of such internal coupling was initiated in the purpose of extending the modeling of thermal hydraulics in the core

and especially for calculating the phenomenon of reversed flow. (Hämäläinen & Räty 2014, 3)

2.1 Nodes and junctions

The nodalization in SMABRE model is a discretization for the reactor system. The basic components consist of nodes, junctions and heat structures. Nodes are the most basic components that determine the dimensions. Out of these dimensions the most important are considered to be length, bottom elevation, top elevation, angle and area. The whole model consists of these nodes. Junction's main purpose is to determine connections between nodes and the heat structures while also determining the type of flow inside the specific component. In Figure 2.1, there is a nodalization of VVER-440 done manually. Figure displays node dimensions and their id numbers. Junctions are located between each node but they are not displayed because their dimensions are not important. Heat structures are present in locations where heat transfer takes place and they are not presented in the figure because the only purpose of this figure is to illustrate the node dimensions that were used in this specific model. The figure only shows one loop but the real model includes six loops. The nodalization in whole VVER-440 in this specific model includes 512 nodes, 659 junctions and 265 heat structures. (Miettinen 2000, 16)

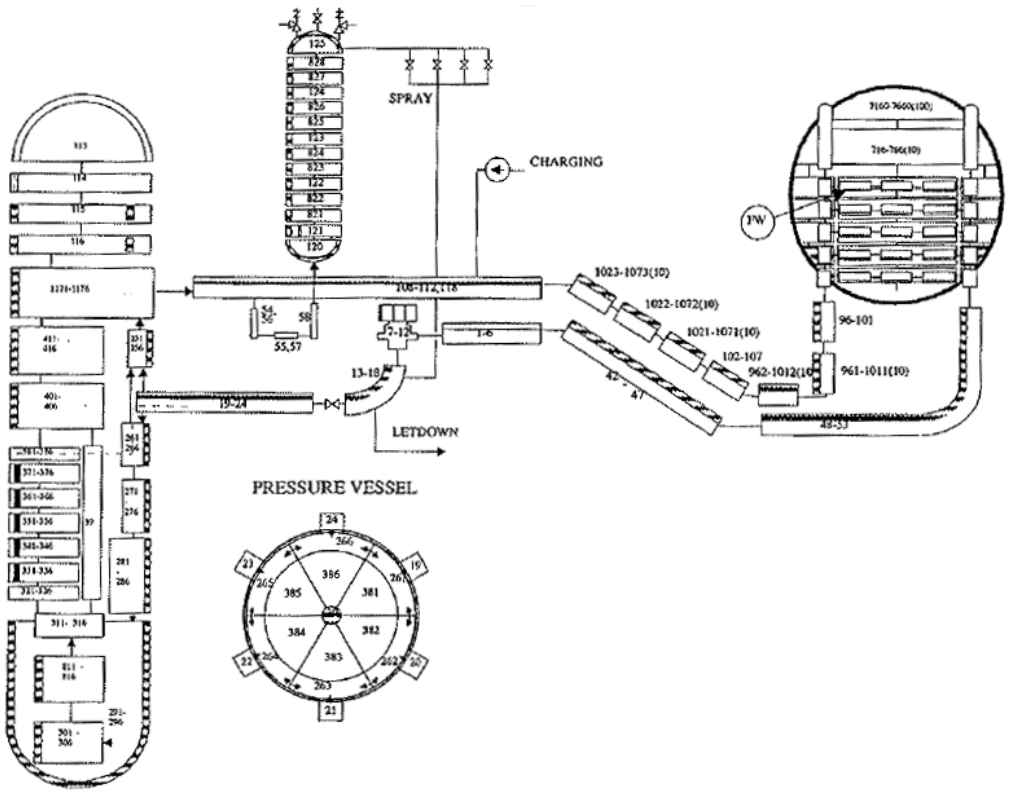

Figure 2.1 Nodalization of VVER-440 for SMABRE. (Miettinen 2000,16)

There are always more junctions than nodes since some of the nodes may consist of multiple junctions for determination of crossflows. Nodes have state parameters such as pressure, void fraction, density, enthalpy and phase mass. These will be the average values and they are located in the node's central elevation. Junctions contain flow related information: velocity, volumetric flow and mass flow. The information of interest in the junctions regarding this project is elevation, junction's type, junction's id, 'from' node and 'to' node. The elevations are very important for junctions; they need to match with the nodes they are connected to in order to get precise results. (Miettinen 2000, 38)

Figure 2.2 shows different example of nodalization: the VVER-1000 with all four loops. This generic model contains over thousand nodes and it is another typical example how the illustration of nodes is done manually. This figure will also act as a reference in section 5 where similar NPP has been created using the finished program.

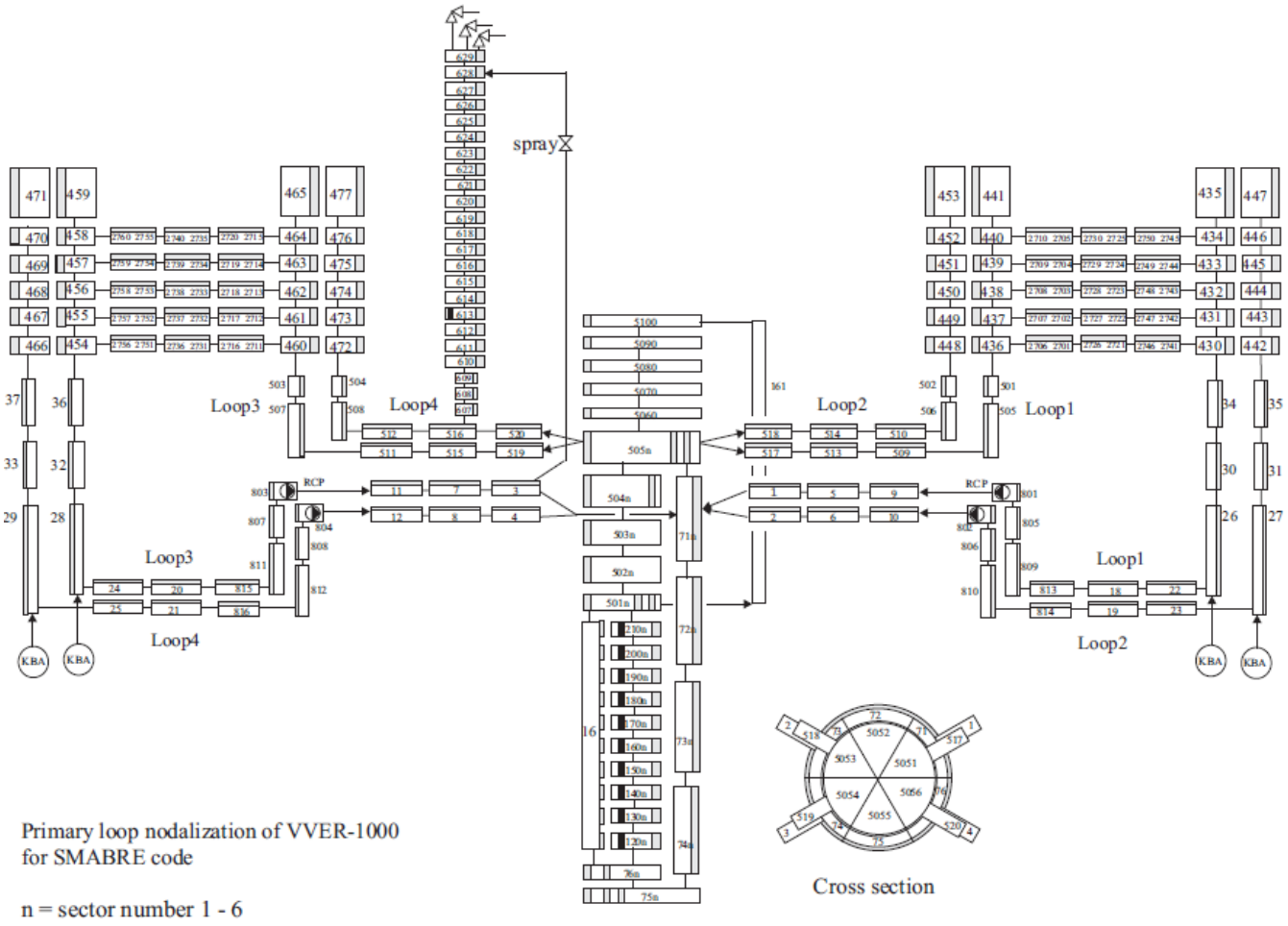

Figure 2.2 Nodalization of VVER-1000 for SMABRE. (Seppälä 2007, 9)

The previous two figures are both done manually. Normally the nodalization seen in these figures are not their real dimensions in correct scale. These figures are meant to illustrate how the nodalization should look, not how they actually are.

2.2 The input/output-files

In SMABRE, the whole plant design is described through an input file. When creating such input file the user has to input vast amount of information through various states. These states are consisted of specific order of inputting certain data. If there are any mistakes in the designed model it is direct result of faulty design or invalid inputs. The created output file includes the listing output, report type of output, plotting output and restart output. Own time interval is defined for each for writing and listing the files. The input/output-files are written with code language Fortran. (Miettinen 2000, 19-20)

The files of interest that contain the necessary information for illustrating node dimensions are found both in input and output files. Typical input/output-file can have around 20 000 lines of information. It contains all the information regarding nodes, junctions, heat structures and calculated results.

2.3 Encountered problems

All of the problems presented are the same with all SMABRE models. These problems require carefully scripted solutions. First problem is with the amount of nodes. The sizes of the input/output-files are enormous; they can easily have thousands of lines of data when storing information of 1000 nodes. The most efficient way to process all this information has to be chosen. There also has to be a solution for cherry-picking exactly the required information that is needed for drawing while at the same time ignoring the data that is not necessary.

The other problems are with certain node data. When simulations are being resolved the actual location for x-coordinates and orientations do not matter because they are not necessary in the calculations. In illustrations these are very important. The elevation stands for the y-coordinate in simulations. The x-coordinates are not used in the simulations at all because the information about where the node is connected is used instead. As for orientation, it simply does not matter if the node is angled 45° to the right or to the left. Regarding the calculations the only thing that matters is the angle. The nodes have all the required dimensional information but it is not enough for generating accurate node illustrations. The x-coordinates are unknown and the actual orientations for angles are

unknown. This will be a problem when illustrating the nodes since it is very hard to determine the correct orientation and the correct x-coordinate.

Final known problem is with the junctions. Junctions determine the connections and relations between the nodes. For gaining this important information all of the junctions have to be browsed through. There are more junctions than there are nodes because some of the nodes might have up to four junctions they are connected to. The problem is how to resolve these connections and how to determine which junctions are needed to ignore for solving the correct connections and relations between the nodes.

3. PROGRAMS USED

The software that was used is explained shortly in following chapters. Note that all the used software in this project was free and open-source.

3.1 Python

For summarizing python the next quote is from the official site. "Python is an interpreted, object-oriented, high-level programming language with dynamic semantics." (Available: https://www.python.org/doc/essays/blurb/) Python was chosen to this project due to it being free and open source programming language and due to its vast capabilities of handling large text files. The input/output-files have a problem as mentioned in section 2.3; they contain immense amount of information and dealing with this the most efficient way was the most influencing reason for choosing Python as language to use. Portable Python was used because it contained useful modules and software. One of these software was PyScripter: A good editor that was very useful when pinpointing mistakes in the coding progress. Portable Python is very deft since it can also be run easily from USB drive.

Python saved time and effort when making the program because it contains many useful modules and packages. Next sections will explain shortly some of the most important features of python that were used in the finished program.

3.1.1 Dictionaries and Lists

Lists are used to store a collection of data in a specific order. Python can utilize this even further in dictionaries that are a mapped data type. In dictionaries one can store data in key-value pairs. This can include specific keys that correspond even into lists of data. Python has amazing functioning using both dictionaries and lists but the most important thing is that one can iterate over lists and dictionaries. (Lie Hetland 2008, 10)

Iterating over lists and dictionaries means that they can be looped over by fixed number of iterations. For example, it is possible to loop through all the indexes in a list or all the keys in dictionaries. This method can be used to temporarily pull specific values from their stored locations and utilize this information somewhere else. Iterating will keep on going until all the elements have been processed or the loop hits in a break command. (Asadi et al 2015, 18) The ability to iterate over dictionaries plays very crucial part in this project

because all of the data regarding nodes and junctions will be saved in their own dictionaries.

3.1.2 Shelve

Dictionaries and lists will get deleted once the program is closed. The shelve module is used in order to keep the data persistent. Shelve module in Python is a very simple storage solution; it is a persistent, dictionary-like object and very easy to use. The function 'open' in shelve is the most important one because it only needs a name for the creation of database. After creation of shelve the data that has been stored in temporary dictionaries can be stored for good and this shelved data can easily be imported anywhere it is needed.

By default shelve cannot know when a mutable persistent-dictionary entry is modified. This means that once shelf is created and the data is written for the first time they become unchangeable because of Python semantics. This does not cause a problem because the stored data only needs to be stored for later use in other modules. The stored data does not need any modifications. (Lie Hetland 2008, 238-239).

The shelve module is used to create persistent databases out of dictionaries and lists that contain important data regarding nodes and junctions. These databases can then be imported to different modules where this data is needed. Using shelve makes the data handling is very efficient.

3.1.3 Regular expressions

Regular expression is a pattern or an order of words which can be used to match similar piece of context. (Lie Hetland 2008, 242) The ‘re’ module makes it very simple to use regular expression based searches. For example, if regular expression was to be assigned as 'length' this could be used to find all precise lines in the input/output-files that contains 'length' respectively. Regular expression can also be assigned as partial word or a slice of word with all possible endings. This is why there has to be careful planning which kind of expression to use since if not defined precisely the ‘re’ module will find all the expressions, even ones that are not necessary.

Regular expression is a very powerful tool and it has been used in this project to solve the problem mentioned in section 2.3 regarding the collection of the specific data in nodes and junctions.

3.2 InkScape

Next from official site: "InkScape is an open-source vector graphics editor similar to Adobe Illustrator, Corel Draw, Freehand or Xara X. What sets InkScape apart is its use of scalable vector graphics, and open XML-based W3C standard, as the native format". (Available: https://inkscape.org/en/about/overview/) InkScape can be used to edit or create vector graphics, for example: logos, diagrams, charts, etc. Most importantly they are scalable. Typical image consists of bitmap; a fixed set of pixels while an image that is composed of vectors is more like fixed set of shapes. When enlarging normal bitmap image the pixels will become visible and the image will eventually become fuzzy. This will not happen in vector images since the image will rescale itself accordingly.

The SVG-files in InkScape are based on XML (Extensible Markup Language). Later when referring to drawing nodes; it will actually mean creating of SVG-files by writing XML. Created by W3C (World Wide Web Consortium), The XML is very structured way for storing data. This Structured data can mean things like pictures, messages, configuration parameters, technical drawings spreadsheets and many more. XML is not a programming language; it is more consisted by a set of rules that define classes of languages. With XML it is very easy to generate data, read data, and ensure that the data structure is unequivocal. (W3, 2014)

4. CREATED MODULES

In this section, the program's behavior and all the created modules that were necessary in illustration of SMABRE nodes are explained. The finished program contained about 3500 program lines and it was split into 5 different modules: Main.py, Arrangement_SMABRE.py, Arrangement_APROS.py, draw_SVG.py and Junction_checker_smabre.py. Some of these modules are explained in the following chapters.

4.1 Main loop and data collection

Module 'Main.py' serves as simple CLI (Command-line interface). All the other modules are built around it but also the data collection is handled by 'Main.py'. User can interact with all of the modules by entering certain commands as seen in Figure 4.1. GUI (Graphical User Interface) was considered but it was never finished due to lack of time.

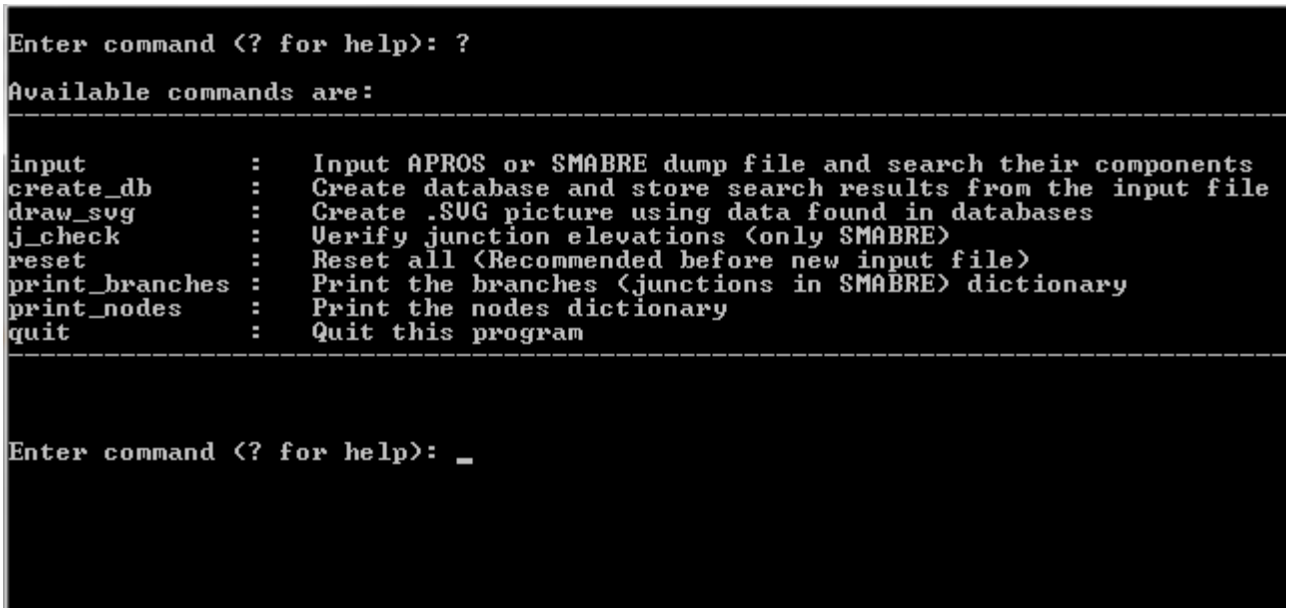

Figure 4.1 The main loop that is acting as simple CLI.

Nothing can be done as long as the needed data is not defined precisely. As mentioned in section 2.3, the dump file contains enormous amount of information. Collecting the needed data manually is incredibly slow; therefore it should never be an option. Python has amazing solution for this: Regular expressions (section 3.1.3).

Module 'Main.py' uses three different regular expressions for three different purposes. The first regular expression matches to a spot in the input/output-file that contains the information about how many nodes and junctions are in the model. The amount of nodes and junctions influences the next two regular expressions that are used to find the dimensional information in nodes, connection types and elevations in junctions. Note that one can easily add more regular expressions, for example finding other desired data such as temperatures, flow rates, etc. SMABRE part of the program does not support illustrations of these mentioned values but they can be added in future versions.

After the user has typed in command 'input' and inserted the input/output-file it will be recognized automatically by the ending of the file. Certain procedures are done for finding desired data and they are stored temporarily under the dictionaries "NodesD" and "BranchesD". Giving command 'create_db' starts shelve module (section3.1.2). One should use this command and store the information for later use in other modules. Figure 4.2 Illustrates the moment after the searched data has been stored by using shelve function under the name "test_data".

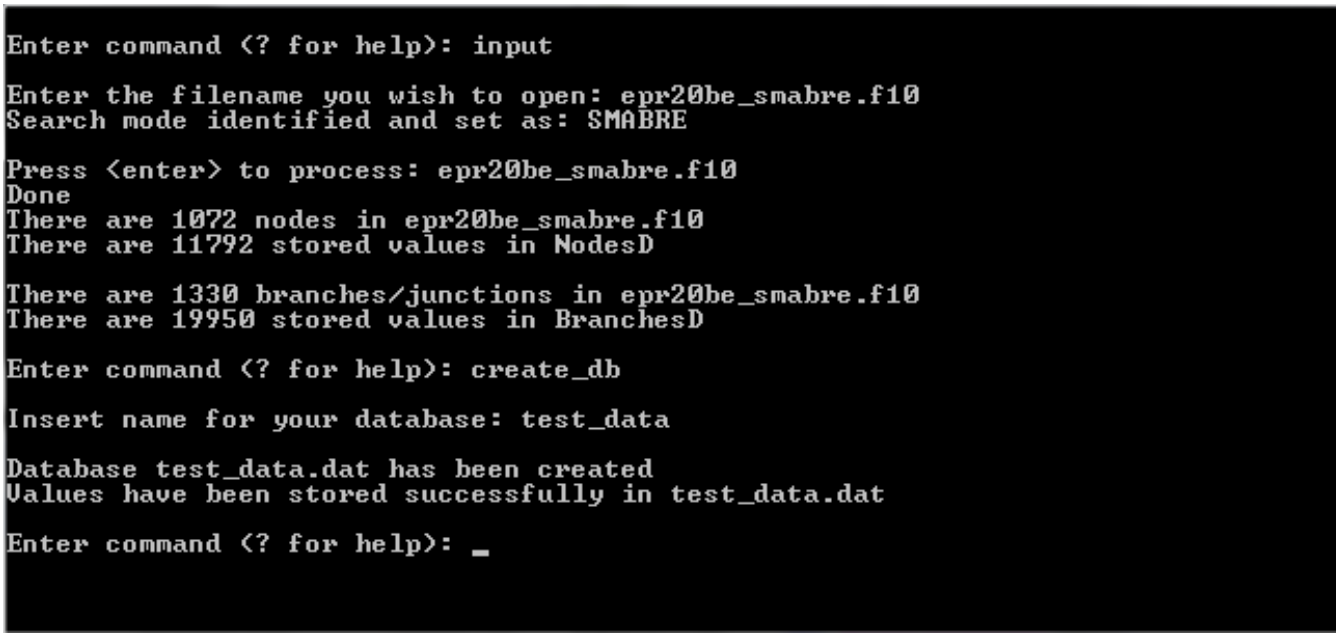

Figure 4.2 The moment after the creation of database.

4.2 Arrangement

All the required data has to be searched before moving onwards. Drawing still cannot be commenced since the data alone is not enough. There needs to be some sort of arrangement or an order how to draw and when to draw the different nodes. The 'Arrangement_SMABRE.py' module was created to handle this task. It is very crucial part of the program since it goes systematically through all of the nodes and solves their connections by using information found both in junctions and nodes. Without arrangement procedure there would be no information about how the nodes would be connected to each other. It is easy to see the difference when comparing Figure 2.2 and the following Figure 4.3 that illustrates how chaotic the nodalization of VVER-1000 would look without the arrangement; if only the elevation and node dimensions were taken into account.

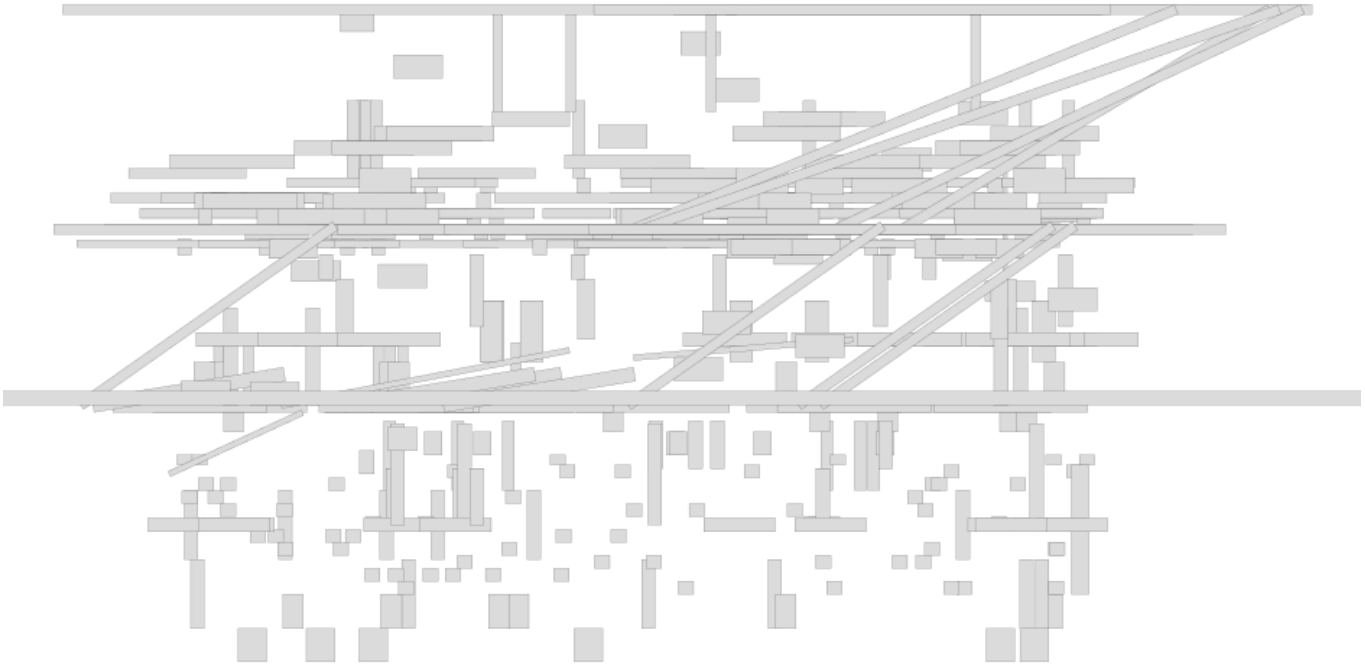

Figure 4.3 The VVER-1000 nodes drawn without arrangement.

Arrangement has been divided into two separate modules. One module is for APROS and one module for SMABRE. Arrangement modules are not run by giving commands in CLI; they are automatic and are run before drawing the SVG-files. Arrangement module finds location and connection for every node by using the junction's type and junction's id and then comparing them to corresponding ‘to’ and ‘from’ nodes. There are some exceptions, because some of the nodes may have two or more junctions they are connected to.

Normally this would cause the arrangement module to make duplicates of these certain nodes. Duplicates should be avoided in order to generate accurate illustrations of node dimensions. There are some precautions to avoid this problem from happening. Duplicate problem is solved by giving the arrangement only once for every node. Once node has its connection locked it is immediately removed from the arrangement progress. The resulting effect is that the rest of the junctions connected to this specific node will get ignored.

The simulated models of NPPs are usually divided into different sections, for example there might be separate loops that are not connected. This would mean that all of the nodes are not connected in one large chain either. Arrangement goes systematically through all of the nodes creating specific chains of them. When encountering node that has no connection for the next node, these nodes are treated as border conditions. Node that acts as border condition in the real model will also behave as an end in arrangement. Arrangement module simply moves on by creating new chain for the next chain of nodes.

As mentioned in section 2.3, the problem with SMABRE is with the enormous amount of nodes. The direct result of this is that the arrangement for SMABRE takes longer (1000+ nodes ≈ 60+ minutes) the program needed slight changes to speed up the arrangement procedure. Arrangement has been made faster by ignoring certain junction types such as cross flow junctions and by removing the already found nodes from arrangement loop, as mentioned before. User can and should create specific database after the arrangement for SMABRE nodes are done. Normally there would not be an option to create shelve database out of arrangement. This option was added, because creating database out of arrangement means that the arrangement is necessary to be run only once on every model. The database is called as "chains" database, because of the nature that arrangement turns all the nodes into certain chains of nodes. Figure 4.4 illustrates the moment after creation of "test_data_chains".

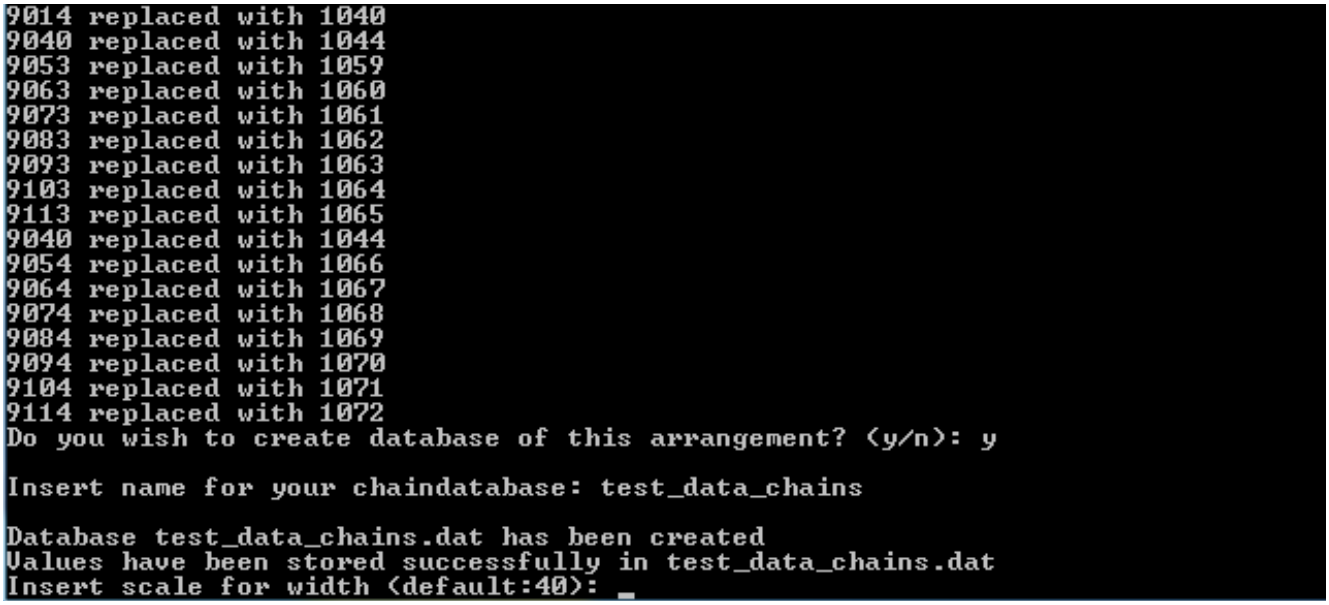

Figure 4.4 Creation of "test_data_chains" database.

Few problems emerged in the final version of the arrangement module. Removing certain junction types indeed made the arrangement module work faster but it also created another problem. The connection became unknown when a certain node was the last node and it also had one of these removed junction types. These nodes could not be connected to anywhere. The decision was made to keep unknown nodes like this, because faster arrangement was more beneficial. These unknown nodes will be drawn in the end of the SVG-file. This will make it clear for the user that the node they are supposed to be connected to is unknown, because of the changes in the arrangement module. Locations for these nodes are found manually by looking through the input/output-files or the written node information in the SVG-file. Typically the amount of these unknown nodes varies between zero and ten. The changes made arrangement procedure to drop down from 60+ minutes to around 20 minutes.

4.3 Coordinate system

The coordinate system is not separate module, it is part of the 'draw_SVG.py' but it is important enough to have a separate chapter. At first it was considered to use heuristic methods for achieving satisfactory results when concerning coordinates of the nodes. All of the nodes would receive random coordinates and then they would be spread out like a shotgun shell on a ceiling. All the connections and relations between nodes are known due

to arrangement module. It was considered creating a method where all the nodes would approach to their connected counterparts. The resulting picture would not be perfect but it would be accurate enough for verifying the correctness of the specific model. This was actually created in one version but the process was overwhelmingly slower than anticipated. Alternative methods had to be taken into account. From heuristic methods, this was switched into more effective use of the already available data. Since knowing arrangement and node elevations the Cartesian coordinate system was chosen. In the Cartesian coordinate system, there is a fixed and shared origin for all of the nodes. Fixed origin was chosen to be at point (0,0). All of the coordinate values that the nodes get are the distance from this fixed origin.

Nodes are defined to have six coordinate points from zero to five. Figure 4.5 shows the coordinates in a single node when flow direction is vertical and upwards. First coordinate is defined by the central elevation of the node and it is separately calculated on each node by using node's top and bottom elevation. It naturally acts as a starting point and rest of the coordinates are calculated around it by using the values that the node possesses. Fourth coordinate (coordinate number 3) is actually useless in SMABRE part of the coordinate system; it is included because it is inherited from the APROS coordinates where the fourth coordinate value has a meaning. Remaining coordinates mark the corners of node.

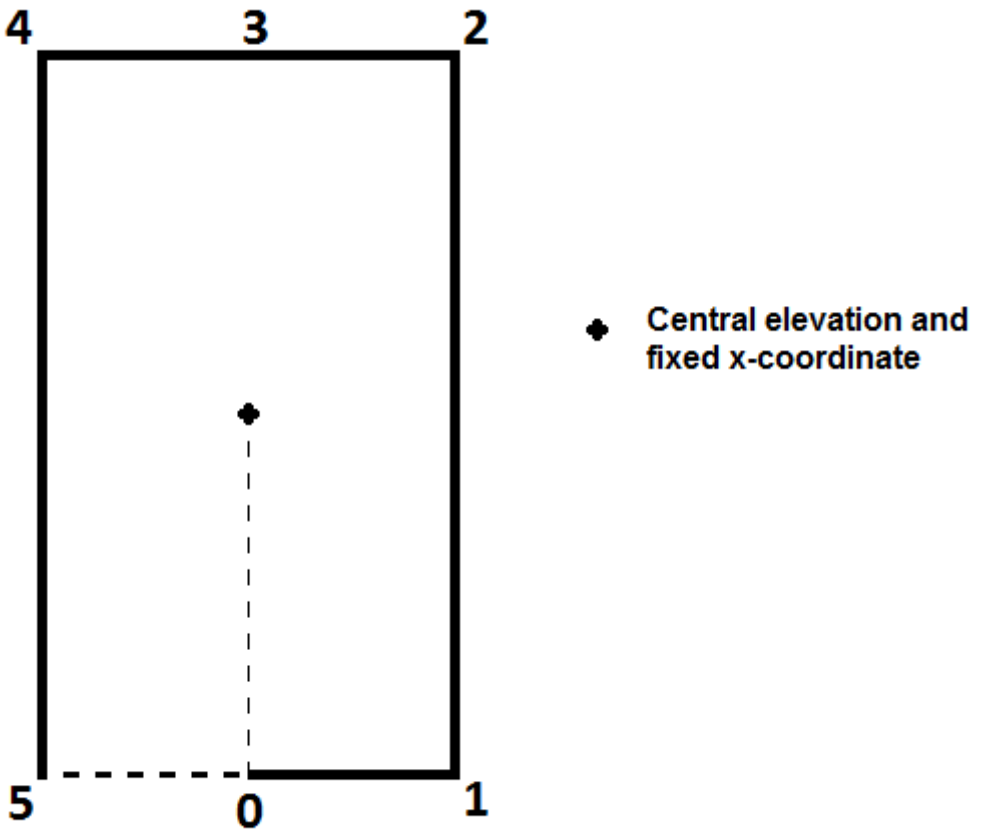

Figure 4.5 Node coordinates points, central elevation and fixed x-coordinate.

As discussed in section 2.3, there are problems with the x-coordinate and angles. Since x-coordinates of the nodes are not available, the starting coordinate has only one true value; the elevation. X-coordinates are generated by the program and the chains of nodes have their own fixed x-values. This means that the resulting picture relies heavily on manual interaction. Manual interaction is also necessary when dealing with some of the angles. There are no easy ways to ensure if the angles are right-handed or left-handed. The program will always assume all angles to be right-handed. The overall result is that the node elevations are correct, the dimensions are correct, the order of the nodes are correct, most of the angles are correct, but the x-coordinates are not since they are simply unknown. Through manual interaction one can easily manipulate x-coordinates of the nodes and build up a picture quite easily on InkScape. Left-handed angles can be achieved by flipping the necessary nodes. Figure 4.6 is showing the result before and after manual interaction.

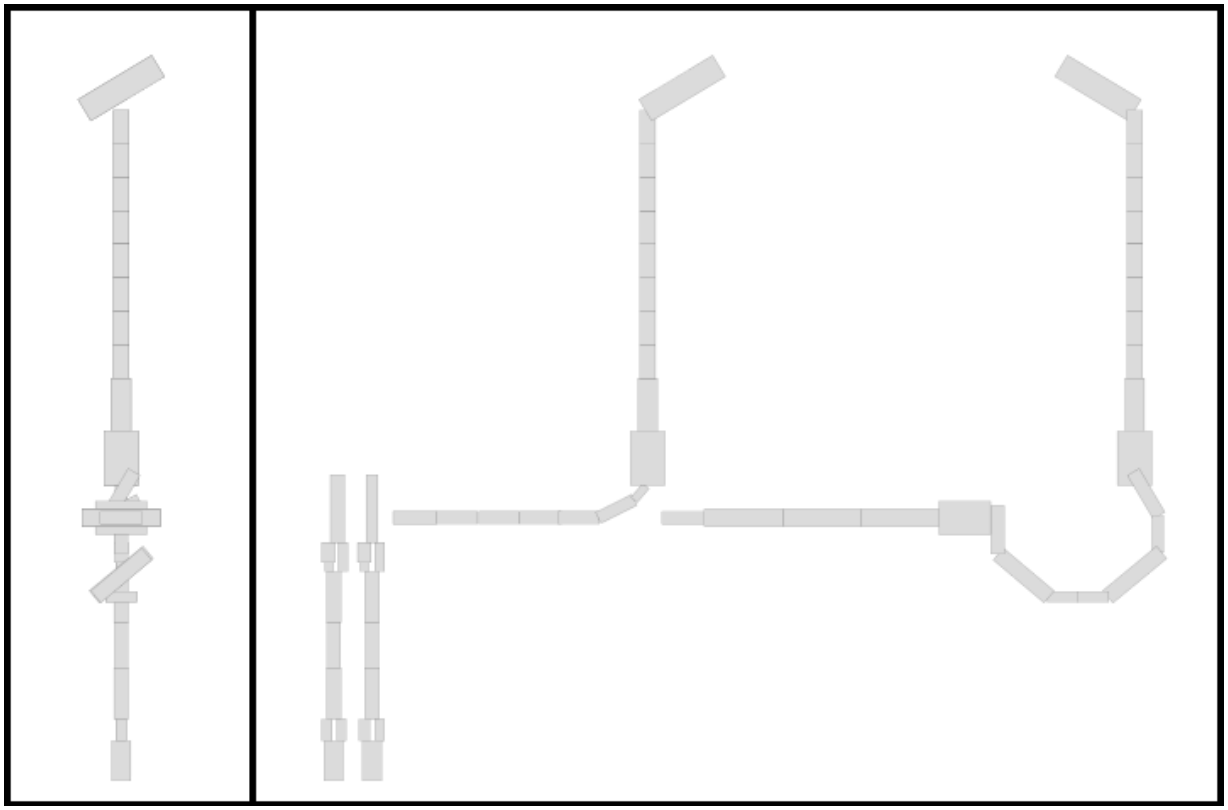

Figure 4.6 On the left are the nodes before manual interaction, on the right after the manual interaction is done. X-coordinates have been modified and some of the nodes have been flipped.

4.4 Drawing

'draw_SVG.py' handles the creation of SVG-file by using all the information discussed before this section. User can start this module by inserting command 'draw_svg' in the CLI. The 'draw_SVG.py' needs to import all the needed data right after the initiation. At this point the module prompts clarification about whether APROS or SMABRE mode is in use. Choosing the mode and importing data causes the arrangement modules to be executed. After arrangement is done the module creates basic information in XML that is needed for the SVG-file. In the drawing progress all the nodes are iterated through in order assigned by arrangement module, as explained in section 4.2.

One of the main purposes in 'draw_SVG.py' is to force every node to go through some stages. These stages consists several conditions and preconditions that decide what to do with the current node. These conditions can be for example: checking if current node is the first one or the last one in the current chain of nodes, checking current elevation and comparing it to previous one/next one, calculating angles and deciding the color of the node. Through these stages the coordinates of the nodes are also calculated as explained in

section 4.3. The 'draw_SVG.py' module writes all necessary information in the SVG-file in XML after all necessary conditions have been gone through. The written information is: name of the node, coordinates of the node, style information and the data that the node possesses as extra information. Extra information helps user to verify the node dimensions. Written extra information is elevations, lengths, running ids, etc. User can view the written node information on InkScape simply by clicking at the specific node. The view on InkScape after clicking specific node is shown in Figure 4.7.

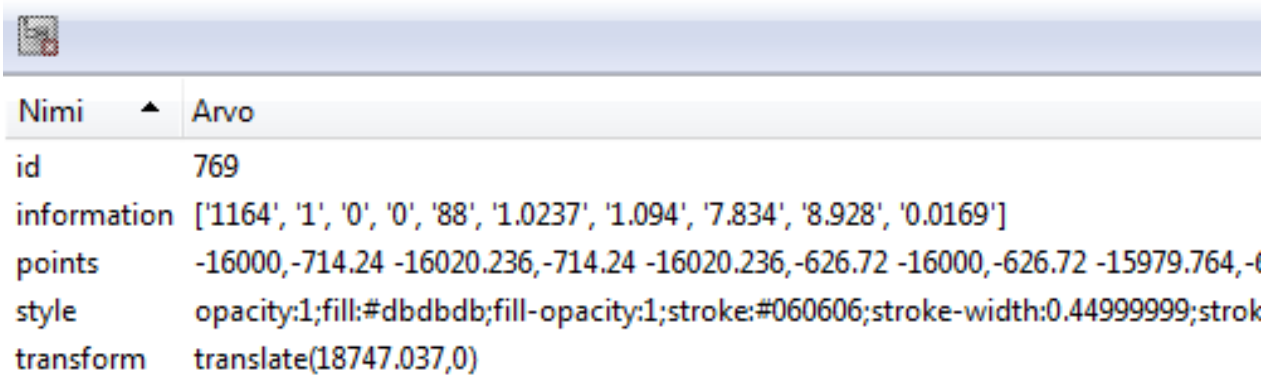

Figure 4.7 Values that are shown in InkScape after clicking on node number 769.

When one chain of nodes has been looped through the module forces new x-coordinates for the next chain of nodes. This will keep on going as long as there are nodes left. There are some procedures when the writing of the final node is done. After the final node is written the module creates elevation scale for better visualization and writes ending layer for SVG-file. Finally the module names the SVG-file as 'SVG_Output_X.svg' where X is a running number. Since SMABRE part of the program only draws nodes and does not illustrate any specific values such as temperature, flow velocity or junction elevations it is not necessary to write or create any extra layers.

4.5 Junction checker

The junction checker was an extra project and it was created due to it being requested. In SMABRE it is very crucial for the working model that the elevations of the junctions are correct. If the simulated model is enormous, there can be some mistakes when inputting the elevations for junctions. The mistakes are hard to spot and it takes long time to verify all

the elevations if checked manually. The purpose of this module is to have fast and reliable way to verify that the junction elevations are correct.

The correct elevation is depending on the junction's type. Typically the junction's elevation should be between the two connecting nodes. 'Junction_checker_smabre.py' iterates over junction database and verifies the elevations by comparing the junction's own elevation and its type and then comparing the elevation with the nodes it is connected to. The module can return four different values while junctions are being checked. These values are shown in Table 4.1.

Table 4.1 Junction checker's returned values.

Returned value	Description
Pass	Junction elevation is correct.
Elevation warning	Junction elevation is within the limits (+-0.001m).
Partial pass/error	Junction elevation is within the limits, but not necessarily in the right spot. Verification by the user is recommended.
Elevation error	Junction elevation does not pass the check. Verification by the user is recommended.

The checking progress is simple and most importantly very fast. In 10 seconds junction checker can verify up to 100 junctions. After the verification progress has ended the module prints out all the warnings and errors. These warnings and errors include the type of the junction, elevation of the junction, ‘from’ node’s elevations and indexes, ‘to’ node’s elevations and indexes. User can use this information to verify and browse the mistakes manually. This is always recommended since some of the errors are actually not real errors due to some models. The junction checker in action is illustrated in Figure 4.8. Some of the returned warnings and errors are found in Figure 4.9.

```
Insert the name of the database you wish to open: test_data.dat
1 Passed!
2 Passed!
3 Passed!
4 Passed!
5 Passed!
6 Elevation warning
7 Passed!
8 Passed!
9 Passed!
10 Passed!
11 Passed!
12 Passed!
13 Elevation warning
14 Passed!
15 Passed!
16 Passed!
17 Passed!
18 Passed!
19 Elevation warning
20 Passed!
21 Passed!
22 Passed!
23 Passed!
24 Passed!
```

Figure 4.8 Junctions 1-24 checked.

```
1027: ['Junction_Type: 4',
       'Junction_Elev: 13.474',
       'From_node: 12124 (node index:794)',
       'From_Bot_Elev: 13.304',
       'From_Top_Elev: 13.474',
       'To_node: 12224 (node index:795)',
       'To_Bot_Elev: 13.304',
       'To_Top_Elev: 13.474'],
1048: ['Junction_Type: 4',
       'Junction_Elev: 15.168',
       'From_node: 12134 (node index:814)',
       'From_Bot_Elev: 13.304',
       'From_Top_Elev: 15.168',
       'To_node: 12234 (node index:815)',
       'To_Bot_Elev: 13.304',
       'To_Top_Elev: 15.168'],
1059: ['Junction_Type: 3',
       'Junction_Elev: 1.037',
       'From_node: 1324 (node index:825)',
       'From_Bot_Elev: 1.065',
       'From_Top_Elev: 2.830',
       'To_node: 1414 (node index:826)',
       'To_Bot_Elev: 0.072',
       'To_Top_Elev: 1.493'],
1077: ['Junction_Type: 2',
```

Figure 4.9 Some of the errors displayed.

5. RESULTS

In this section the results will be presented and part of VVER-1000's node dimensions will get verified. In Figure 2.2 there was VVER-1000 nodalization done manually and in the following Figure 5.1 is one sector, one loop and a pressurizer of a generic VVER-1000 done by the program. The resulting figure is after some manual interaction regarding x-coordinates and flipped angles.

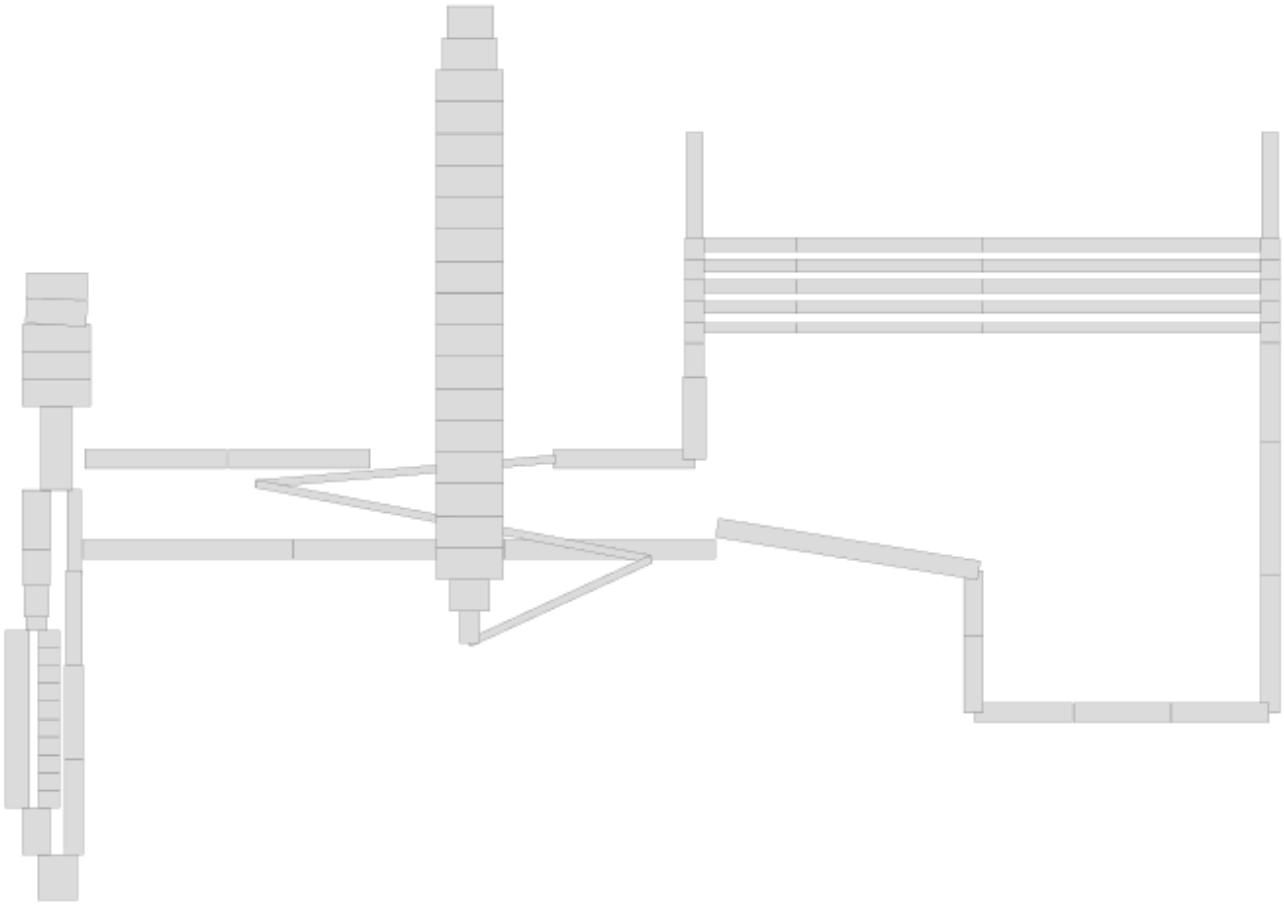

Figure 5.1 Part of VVER-1000 nodalization done by the program.

When looking at the SVG-file after it has been made, the user can verify the node dimensions. There are four possible mistakes that can be pointed out in the previous picture. Upon closer inspection one of the node elevations in cold leg appears to be wrong. The top elevation of this node is 0,425 meters and the node it is connected to has the central elevation of 0. Junction checker also picks up this mistake as a partial pass/error

since junction's elevation in this connection is also 0, making it correct on one node. Figure 5.2 shows this conflict.

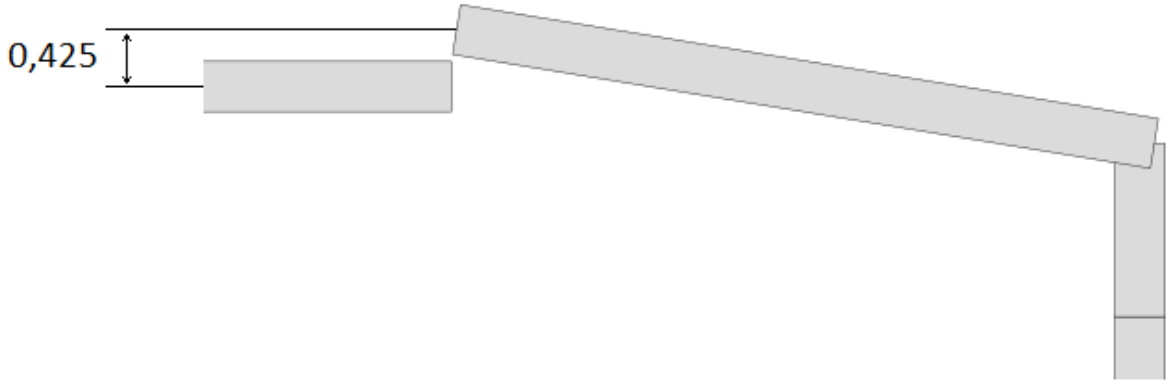

Figure 5.2 Elevation difference in the cold leg.

One might say that an obvious mistake has been found. In truth this is correct since circulation pump is located in the spot between these two nodes. "This is more unique method in SMABRE to deal with some elevation differences with the water levels caused by circulation pump". (Hämäläinen Anitta, personal communication, November 19th, 2015) This is also why it is very important to check partial pass/error caused by junction checker; most of the pass/errors are correct even though at first they might seem as errors.

Second possible mistake can be observed in the upper plenum: there seems to be one node that is slightly tilted. This is caused by the fact that the node's length is 0,001 meters too much for it to have the angle of 90 degrees. This is the reason why the program forces small angle for this node. Closer inspection in Figure5.3

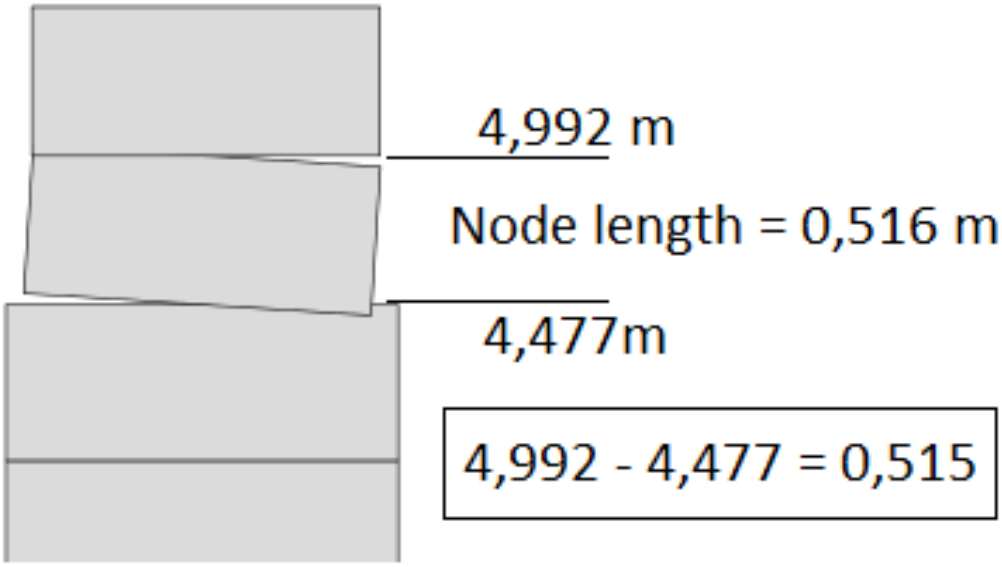

Figure5.3 Small tilt found in the node, because it is slightly longer than it should be.

The junction elevations in these kinds of tilted nodes are usually either pass or warning. They are in correct elevation and within the margin. This will cause that they are not treated as errors. “The node is 0,001 m too long. There are still some cases in some SMABRE models, where the nodes are supposed to have slight tilts like this in the purpose of creating longer flow within the node”.(Hämäläinen Anitta, personal communication, November 19th, 2015) The conclusion is that this might be an error. It is up to the developer of the model to verify if the node is meant to be longer or not.

One possible mistake is more hidden, because it cannot be seen just by looking at the SVG-file. The user who changes the x-coordinates of the nodes can easily spot this mistake while doing so. There are two nodes that are overlapping each other. Figure 5.4 shows the overlapping.

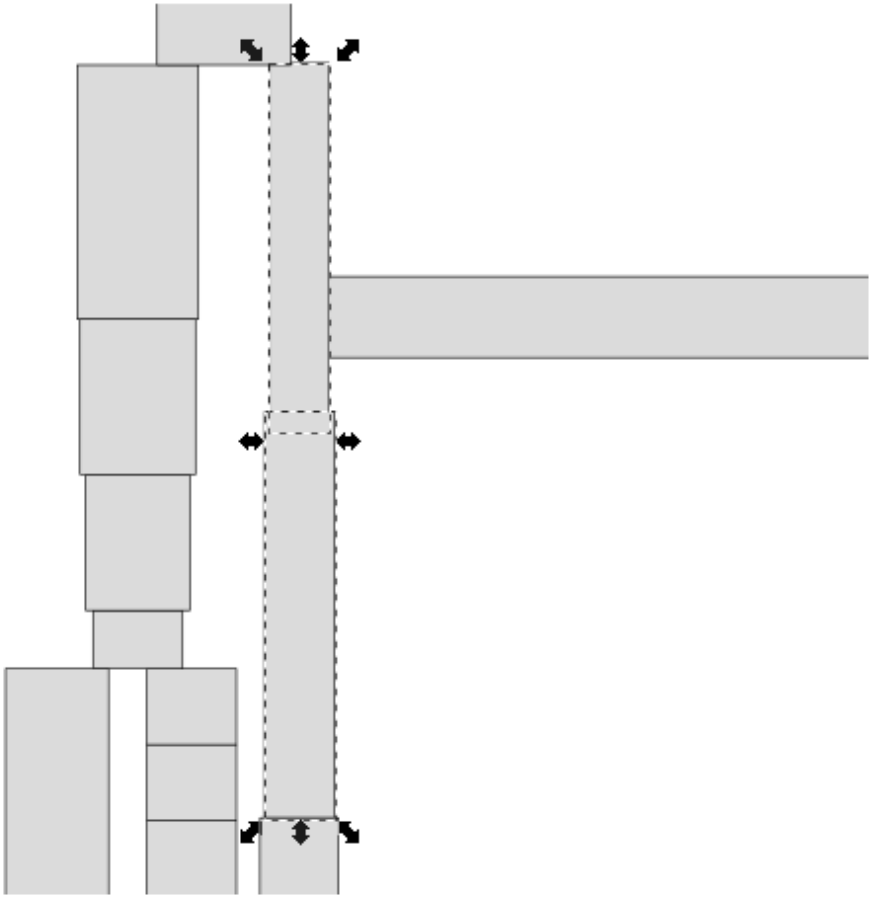

Figure 5.4 Two overlapping nodes.

The junction checker returns error for the junction's elevation between these two nodes. These overlapping nodes exist in SMABRE models for other special purposes. The conclusion is that there is a possible mistake. The developer of the model should verify the junction's elevation and purpose.

The last possible mistake is found in the hot leg. It appears that one node is missing behind the pressurizer making the hot leg look shorter than it should be. There are three possible explanations for this: One node is actually missing from the model, the lengths of the nodes should be longer or the designed model is supposed to be presented like this. The nodes are connected correctly and it is not picked up by junction checker as error. The verification is upon the designer of the model. This mistake has been greatly adduced during manual interaction for illustrating this mistake better. Missing node or the missing

lengths are shown in Figure 5.5

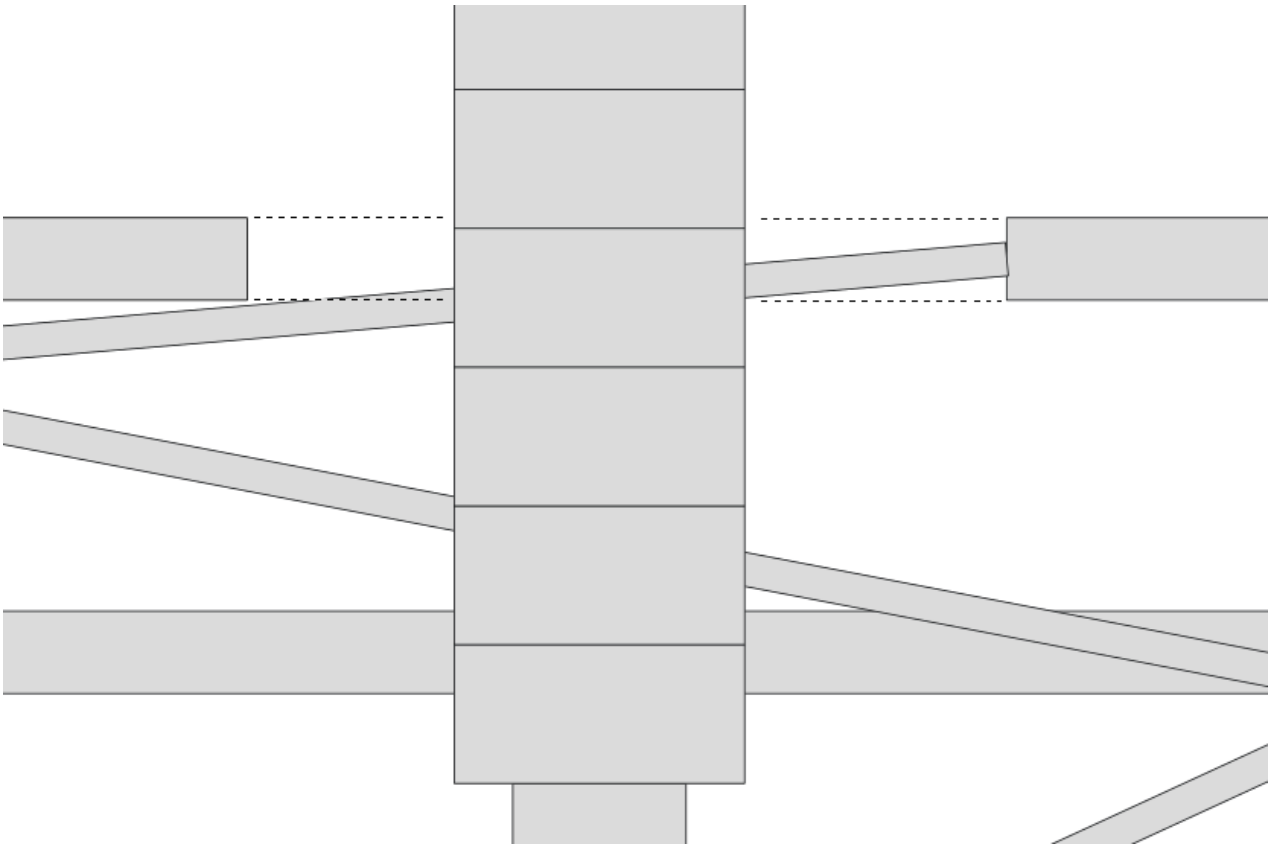

Figure 5.5 The missing node or the missing lengths behind pressurizer.

6. SUMMARY

The created program is used as a tool to illustrate node dimensions that are in SMABRE and APROS models. Neither SMABRE nor APROS supports illustrations of node dimensions. Before the illustrations of node dimensions were created completely manually and they were created how the nodes should be, not how they actually are. The program is ideal when verifying that the node dimensions and the junction elevations are correct. The model will be more reliable when these are completely correct.

The drawn node dimensions displayed in results section did not contain any crucial mistakes. Building up picture on InkScape and verifying node dimensions were easy and fast to do. This program has achieved the initial goals and it is working, even though there was not much testing performed. Downside for the program is that it requires manual work for x-coordinates and some angles when dealing with the SVG-files but this is small price to pay for the gained results.

The finished program contained about 3500 program lines and it took about 4 months of development and it has been distributed within VTT nuclear safety department.

REFERENCES

Aaron Asadi, Ross Andrews, Alex Hoskins, Greg Whitaker, Perry Wardell-Wicks, 2015 The Python Book: The ultimate guide to coding with Python. ISBN 9781785460609

Hämäläinen Anitta, Research Team Leader, Reactor analysis, personal communication 19.11.2015

Hämäläinen Anitta, Räty Hanna, 2014. BWR load rejection transient with internally coupled TRAB3D-SMABRE. Report identification code: VTT-R-00925-14.

LieHetland Magnus, 2008. Beginning Python From Novice to Professional, Second Edition. ISBN-13 (electronic): 978-1-4302-0634-7

Miettinen Jaakko, 2000. Thermohydraulic Model SMABRE for Light Water Reactor Simulations. Helsinki University of Technology, Department of Engineering Physics and Mathematics, Licentiate thesis. Chair: Tfy-56.

Seppälä Malla, 2007. HEXTRAN-SMABRE Calculation of the VVER-1000 Transient Benchmark, Main Steam Line Break. Report identification code: VTT-R-08573-07.

W3. XML in 10 points. Updated 22.9.2014, [retrieved November 26, 2015]
From: http://www.w3.org/XML/1999/XML-in-10-points