SEMANTIC VALIDATION OF BIOLOGICAL MAPS IN SBGN

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Umut Utku ÇALIŞ
September 2019

Semantic Validation Of Biological Maps In SBGN
By Umut Utku ÇALIŞ
September 2019

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Uğur Doğrusöz(Advisor)

Çiğdem Gündüz Demir

A. Elif Erson Bensan

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

SEMANTIC VALIDATION OF BIOLOGICAL MAPS IN SBGN

Umut Utku ÇALIŞ
M.S. in Computer Engineering
Advisor: Uğur Doğrusöz
September 2019

Graph visualization is a research field where relational information is graphically represented in the form of graphs or networks. It is applicable in numerous areas from computer network systems, to biology, to software engineering. In such areas, graph visualization techniques provide effective visual analysis of graph based data. Systems Biology Graphical Notation (SBGN) facilitates a standard model for representing biological entities and their interactions by using graph visualization. SBGN-ML is an XML based format for keeping information about SBGN maps. libSBGN enables writing and reading SBGN-ML files in an easy manner and is meant to bring syntactic and semantic validation to SBGN maps. It is currently available in Java/C++ (libSBGN) and JavaScript (libSBGN.js) programming languages with varying support for aforementioned.

libSBGN enables important syntactic and semantic correctness concepts for manipulating SBGN maps and converting SBGN-ML files into several other formats. Syntactic validation of SBGN-ML files involves using a simple XML Schema Definition (XSD) file. This validation checks whether files are in correct form or not. However, this XSD file does not enable checking against semantic rules. For semantic validation of such files, the Schematron language was developed providing higher level semantic rule controls.

With this thesis, we first enabled high level semantic validation (schematron validation) of SBGN maps in libSBGN.js, which uses XSLT and transformation of process description maps in SBGN-ML files. By using Schematron rules which are written in XPath syntax and enabling human-readable messages of validation errors and source of errors, we developed an XSLT stylesheet. We obtained validation result report by transforming SBGN-ML files using this XSLT stylesheet. In the JavaScript version of libSBGN library, we used a web based XSLT processor

for transformation; hence, this library is now available for providing schematron validation in any SBGN related software. Furthermore, we added schematron validation checks to Newt, a web based SBGN pathway editor, using the updated libSBGN.js library. With this addition, Newt is now able to show validation results not only in a human-readable message text for the current map but also highlights the invalid map objects graphically, and, where appropriate, suggests a way to fix the problem automatically.

Keywords: Graph Algorithms, Graph Visualization, Systems Biology,Semantic Validation, Schematron Language, XSLT, Newt, libSBGN, SBGN.

ÖZET

SBGN NOTASYONUNU KULLANAN BİYOLOJİK HARİTALAR İÇİN ANLAMSAL DOĞRULAMA

Umut Utku ÇALIŞ
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Uğur Doğrusöz
Eylül 2019

Çizge görselleştirme, birbiriyle ilişkili bilgileri grafiksel biçimde çizge ya da ağlar şeklinde gösteren bir araştırma alanıdır. Bu gösterim, ağ sistemleri, yazılım mühendisliği ve biyoloji gibi birçok alanda uygulanabilmektedir. Bu alanlarda, çizge görselleştirme teknikleri çizge tabanlı spesifik verilerin etkili görsel analizini sağlamaktadır. Systems Biology Graphical Notation (SBGN) biyolojik varlıkları ve bunların etkileşimlerini göstermeyi sağlayan bir standardı çizge görselleştirme kullanarak tanımlamaktadır. SBGN-ML, SBGN haritaları hakkındaki bilgileri tutan bir XML tabanlı formattır. libSBGN ise SBGN-ML dosyaları üzerinde okuma ve yazma işlemini kolay bir şekilde sağlamakla birlikte SBGN haritaları için sözdizimsel ve anlamsal doğrulamayı da mümkün kılmaktadır. Bu üst seviye anlamsal doğrulama daha önce bahsedildiği gibi Java/C++ (libSBGN) ve JavaScript (libSBGN.js) programlama dillerinde farklılık gösteren seviyelerde desteklenmektedir.

libSBGN, SBGN haritaları üzerinde değişiklik yapmak ve başka birçok formata dönüştürmek için önemli sözdizimsel ve anlamsal doğrulama konseptlerine olanak sağlamaktadır. SBGN-ML dosyalarının sözdizimsel doğrulaması basit bir XSD dosyasını kullanmayı içermektedir. Bu doğrulama dosyaların doğru formatta olup olmadığını kontrol etmektedir. Fakat, bu XSD dosyası SBGN içerisindeki anlamsal kuralları kontrol etmeye olanak sağlamamaktadır. Üst seviye anlamsal kontrolü sağlayan Schematron dili böyle dosyaların anlamsal doğrulaması için geliştirildi.

Bu tezimizle birlikte, ilk olarak SBGN haritalarının XSLT ve süreç tanımlama haritalarının dönüşümünü kullanan üst düzey anlamsal doğrulaması libSBGN kütüphanesinin JavaScript versiyonu içerisinde sağlandı. XPath formatında yazılan, doğrulama hataları için okunabilir mesajları ve hataların kaynaklarını

gösteren Schematron kuralları kullanarak XSLT stil sayfası üretildi. Bu stil dosyası kullanılarak SBGN-ML dosyalarını dönüştüren doğrulama sonuç dosyası elde edildi. libSBGN kütüphanesinin JavaScript versiyonunda, dönüşüm için Web tabanlı XSLT işleyicisi kullanıldı ve bu nedenle kütüphane diğer ihtiyaç duyulan SBGN yazılımlarında da Schematron doğrulama sağlamak için kullanılabilir duruma geldi. Buna ek olarak, Schematron doğrulama kontrolleri, değişen libSBGN.js kütüphanesini kullanarak SBGN editörü Newt aracına eklendi. Bu eklemeyle birlikte Newt, seçilen haritalar için doğrulama sonucunu okunabilir bir mesaj şeklinde gösterebilmenin yanı sıra, doğrulama açısından sorunlu objeleri renklendirerek gösterebilir ve uygun bir şekilde doğrulama problemlerini otomatik olarak çözen yollar önerebilir hale getirildi.

Anahtar sözcükler: Çizge algoritmaları, çizge görselleştirme, sistem biyolojisi, anlamsal doğrulama, şematik dil, XSLT, Newt, libSBGN, SBGN.

Acknowledgement

I would like to express my interior thankfulness to my supervisor Prof. Uğur Doğrusöz for providing me a chance to work with him and defining a thesis topic that I studied with pleasure. His support and guidance always helped me to work on my research during my thesis. He allocated his very much time to review my thesis.

I would like to thank Prof. A. Elif Erson Bensan and Assoc. Prof. Çiğdem Gündüz Demir for reviewing and commenting about handwriting of my thesis.

I would like to express my deepest appreciation to my father and my mother for their support.

I am grateful to Mustafa Enes Karaca and Ozan Can Altıok for their support in terms of technical manner.

I am thankful to Büşra Temel and Sinem Akçakoyunlu for their support about English language.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A graph is a collection of nodes which indicate data points and edges that represent connection between nodes [4]. Graph visualization is an area that includes visual representation of nodes and edges in a graph in two dimensions by using manual or automatic layouts of graph objects [5].

Systems Biology Graphical Notation (SBGN) is a notational language which is designated and developed by scientists to model biological processes and pathways easily through graph visualization. It is composed of three languages, which are: process description (PD), activity flow (AF) and entity relationship (ER) [6].

libSBGN is a library that enables manipulating, reading and writing SBGN-ML files and it has support for Java and C++ [7]. A JavaScript version named libSBGN.js was also recently made available. SBGN-ML is an XML-based file format that enables storing and exchanging biological information in SBGN maps effectively [2].

Schematron language is a structural validation language which includes tree patterns (XPath expressions) to assert and produces result-based reports [8].

XSLT (Extensible Stylesheet Language Transformation) is a process that transforms an XML file. An XSL stylesheet indicates how an instance of any

class is transformed into an XML document by using certain rules [9].

Newt is a web-based editor for making graphical changes interactively on SBGN-ML files and uses Cytoscape.js and many related extension libraries [3].

1.1 Motivation

libSBGN provides syntactical and semantic correctness mechanisms for SBGN-ML files and helps conversion to desired formats.

Schematron language enables applying higher level semantic rule checks which are written in XPath. This language produces validation result report that indicates whether errors exist or not and represents source of errors if they exist by including XSLT. Hence, this language can be integrated to libSBGN library for checking SBGN maps in terms of semantic correctness.

Since Newt is a tool that represents SBGN maps in an organized way, schematron validation check can be added to it easily by using the libSBGN.js library in this tool. This validation control can be shown within this tool effectively and nicely through its advanced visualization facilities for SBGN-ML files such as highlighting, panning and zooming.

1.2 Contribution

In our thesis, we enabled higher level semantic validation for SBGN maps in process description language. We realized this by extending the JavaScript version of libSBGN library. In this process, we added schematron validation property to the libSBGN.js library. We produced a schematron rule set file, benefiting from process description rule file precisely in the Java version of libSBGN library. We provided validation of SBGN maps by using this rule set file. We produced a general structure realizing this validation in the libSBGN library, so this validation

can be imported to any desired project.

To integrate semantic level validation into the libSBGN.js library, an XSLT processor was required for transforming SBGN-ML files according to schematron ruleset file. For this purpose, we used Mozilla's web based XSLT processor which includes client side validation and produces validation result report efficiently.

Once higher level semantic validation was developed in the JavaScript version of libSBGN library, we were able to easily accommodate this validation support in Newt using its user-friendly graphical interface. Through this work, Newt now can show human-readable error messages and uses highlight facilities to show parts of SBGN maps which are problematic. Furthermore, where applicable, Newt suggests possible fixes to the problem, again by highlighting the manner in which problem can be resolved. Should the user agree to a suggested fix, Newt automatically fixes the problem, re-running validation and showing any remaining issues.

Chapter 2

Background and Related Work

2.1 Graph Visualization

Graphs or networks are representation of relational information in a mathematical manner, where each object called a *node* is represented by a shape (usually a point), and the relationship between two objects called *edges* are represented with a line or a sequence of line segments. Formally, a graph $G = (V, E)$ is a set of vertices V, also called nodes, connected by edges $e = \{u, v\} \in E$, where $u, v \in V$.

Graph visualization is a sub-field of information visualization which includes representation of interconnected nodes in a network through a visual representation. This area helps humans to understand global or local structures of data and it enables them to analyze a set of data with relations easily [5]. In graph visualization, in cases where the relationship is directional (e.g., a person likes another person), corresponding edge is drawn with an arrowhead to graphically indicate the direction of the relationship.

2.2 SBGN

Systems Biology Graphical Notation (SBGN) is a language that is developed by biochemists, modelers and computer scientists for enabling convenient representation of biological processes and interactions in the domain of system biology [6]. It has three sub-languages which are process description, entity relationship, and activity flow. SBGN brings comprehensive visualization as well as exchange of all kind of biological data.

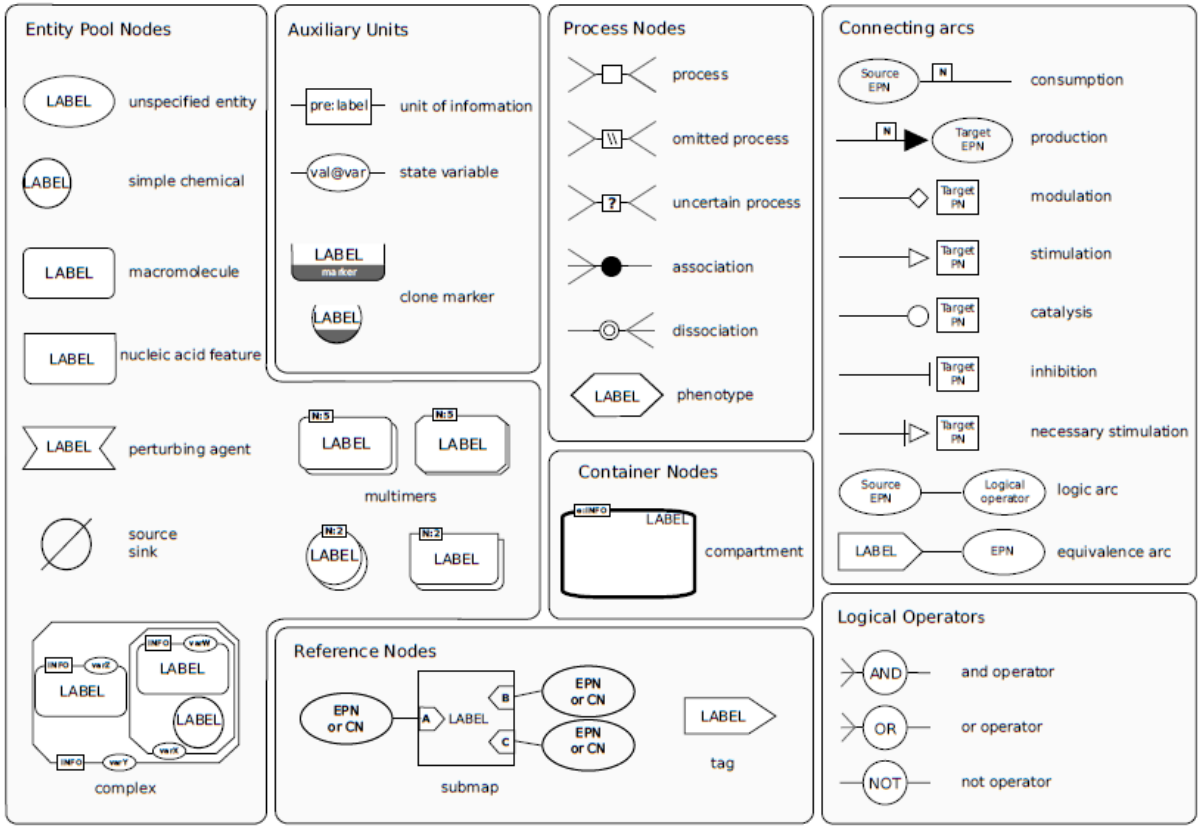

Figure 2.1: SBGN Reference Card [1]

2.2.1 Process Description Language

Process Description (PD) language is arguably the most popular of the languages of biological processes offered by SBGN and it shows all processes and interactions at a mechanistic level [1].

PD language contains nodes which are split into entity nodes describing biological components, and process nodes describing reactions and associations, and edges that show relationships between nodes such as consumption and production [10]. Main elements in this language are defined as entity pool nodes, process nodes, container nodes, reference nodes, connecting arcs and logical operators as seen in Figure 2.1 [6].

Entity pool nodes are unspecified entity, simple chemical, macromolecule, nucleic acid feature, perturbing agent and source and sink glyphs. Indeed, unspecified entity is that its type is unknown. Simple chemical is an element that represent chemical components. Macromolecules demonstrate biochemical elements which contribute to biological processes. Nucleic acid feature carries genetic information. Complexes consist of other biochemical entities. Perturbing agent shows external effects on biochemical networks. Lastly, source and sink is an element that represents creation of entity from unknown source [10].

Process nodes convert a set of entity pools to another set of entity pools. They can be defined as process, omitted process, uncertain process, association or dissociation [10].

2.2.2 SBGN-ML

SBGN-ML is a well-organized XML-based file format that indicates all geometry of SBGN maps by preserving biological meaning. It is developed based on two criterias, which are being easy to draw and read and being easy to interpret. Since file format includes all needed information, extra calculation is not needed [2] (Figure 2.2).

Semantic and syntax of SBGN-ML file format enables easy description of biological maps. File format keeps biological meaning of an SBGN map, so it specifies the nature of graphical elements (glyphs), following the SBGN terminology (e.g., macromolecule, process, etc.). In terms of syntax, file format indicates information relationships between SBGN objects such as the glyphs at both ends of an arc, the components of a complex, the members of a compartment. As a result, this semantic and syntax of SBGN-ML file format enables validation and analysis of biological networks [2].

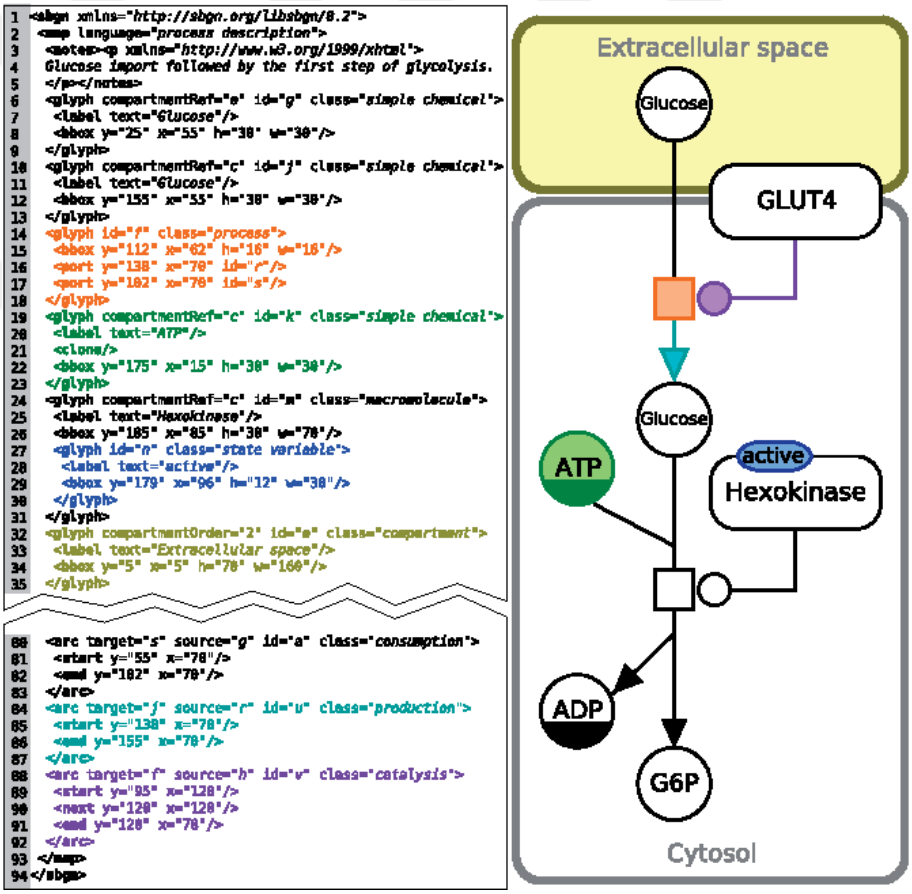

Figure 2.2: SBGN-ML Sample [2]

2.2.3 libSBGN

libSBGN is a software library for writing, manipulating and reading files in SBGN-ML format. It also yields semantic and syntactic validation for SBGN-ML files and facilitates conversion to other formats such as SBML and BioPAX. This library is available in Java/C++ (libSBGN) and JavaScript (libSBGN.js) programming languages with varying support [7]. Before the work detailed in this thesis, there was no semantic validation support in libSBGN.js.

2.3 Schematron Language and XSLT

Schematron language is a rule-based validation language that offers a way to test XML documents and it is expressed in XML using XML elements and XPath. This language specifies and tests statements about an XML document in terms of attributes, content and elements. Schematron program is called a schema and it is a well-organized XML document. Elements in this program are commands in the Schematron language [11].

In schematron validation, it is needed to have a schematron rule set file which specifies tests to be done on XML sample file (Figure 2.4). A kind of schematron processor is necessary to realize this process. As a result, after applying schematron processor, a validation report can be obtained (Figure 2.3) .

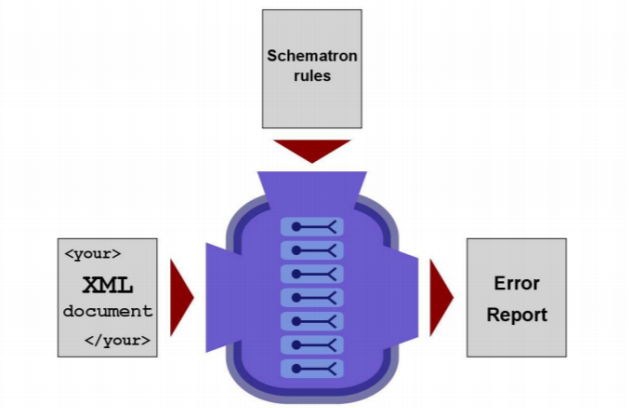

Figure 2.3: Schematron Work Process Architecture

```
<sch:schema xmlns:sch="http://www.ascc.net/xml/schematron" xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <sch:ns uri="http://integration.team/blog" prefix="ns0"/>
    <sch:pattern name="surname and lastname">
        <sch:rule context="ns0:Blog/ns0:AuthorInfo/Surname">
            <sch:assert test="string-length(text()) > 0">Surname needs to be filled in.</sch:assert>
        </sch:rule>
        <sch:rule context="ns0:Blog/ns0:AuthorInfo/Lastname">
            <sch:assert test="string-length(text()) > 0">Lastname needs to be filled in.</sch:assert>
        </sch:rule>
    </sch:pattern>
    <sch:pattern name="Title validation">
        <sch:rule context="ns0:Blog/ns0:Title[text() != '']">
            <sch:assert test="contains('ABCDEFGHIJKLMNOPQRSTUVWXYZ',substring(text(),1,1))">Title needs to start with a capital.</sch:assert>
        </sch:rule>
    </sch:pattern>
    <sch:pattern name="Address validation">
        <sch:rule context="ns0:Blog/ns0:AuthorInfo/CompanyAddress[../Company/text()='Integration.Team']">
            <sch:assert test="text() = 'Veldkant 33A, 2550 Kontich'">Wrong address for Integration.Team</sch:assert>
        </sch:rule>
    </sch:pattern>
</sch:schema>
```

Figure 2.4: Schematron Rule Set Sample

In schematron program, `context` attribute on `<rule>` sets the context and determines when testing is done. `assertion` and `report` attributes are used for testing purposes. `<assert>` means *tell us if it is not true* and `<report>` means *tell us if it is true* [11].

2.3.1 XPath

XPath is a query language that provides traversing through an XML document. It is commonly used for search elements or attributes with desired patterns [12].

An XPath expression defines a pattern which is source for XSLT in order to

select single node or lists of nodes. XPath has seven types of nodes which are root, *element*, *text*, *attribute*, *comment*, *processing instruction* and *namespace*. XPath uses path expressions to select node or a collection of nodes from an XML document (Figure 2.5).

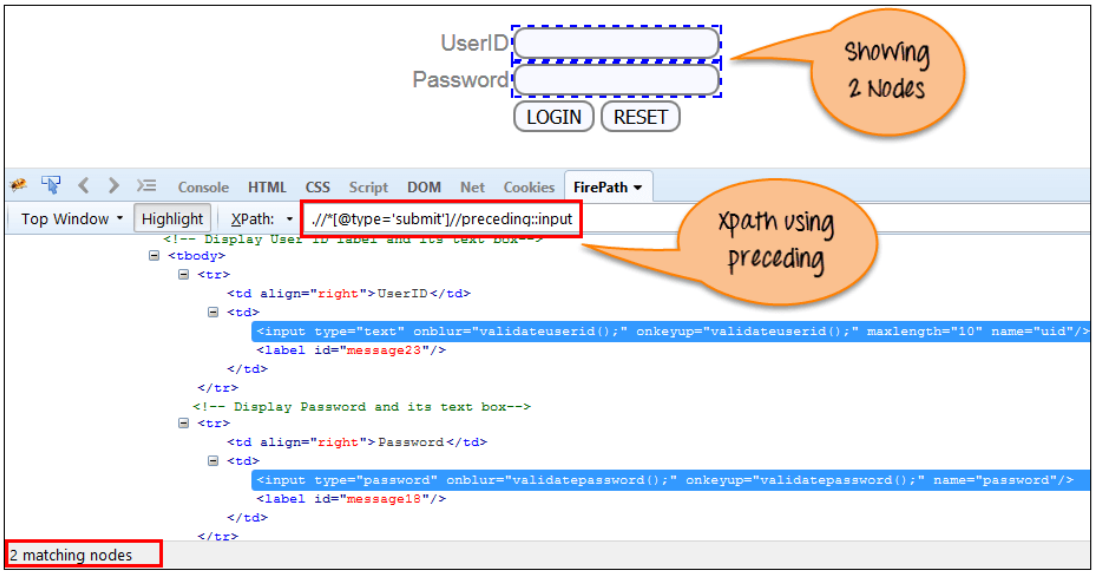

Figure 2.5: XPath Sample

2.3.2 XSLT

XSLT (Extensible Stylesheet Language Transformation) was created for transforming the structure of an XML document [13]. It supports publishing data and conversion between different formats. XML data can be converted by using XSLT with Document Object Model (DOM) or SAX (Simple API for XML) (Figure 2.6). Output tree structure can be obtained from the input with the help of structural transformations that can contain selecting-projecting-joining, aggregating, grouping and sorting data (Figure 2.7).

Transformation process relies on template rules. A rule consists of template patterns and template body. Template patterns are simple XPath expressions. In addition to this, template body includes result elements and XSLT instructions. A pattern that matches nodes in the source tree is searched and once a matching

pattern is found, the template body is initialized. In this process, it is demanded that templates are found to apply to nodes. After processing a node, its children are processed if they exist [13].

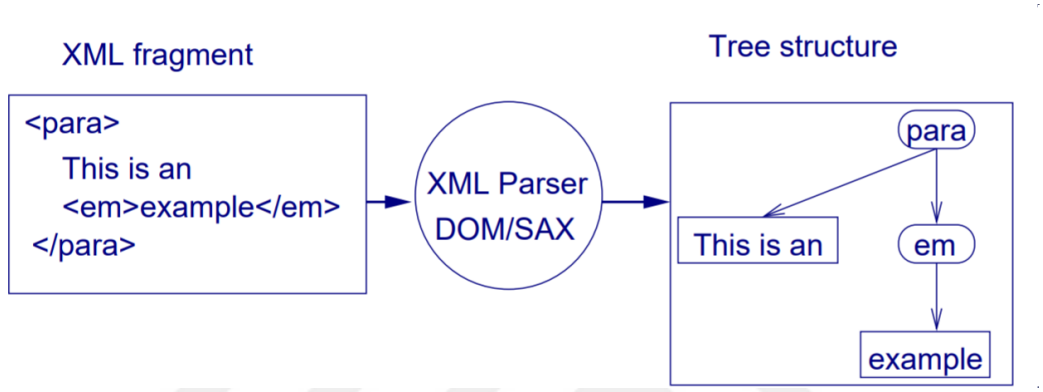

Figure 2.6: XSLT Tree Structure Conversion

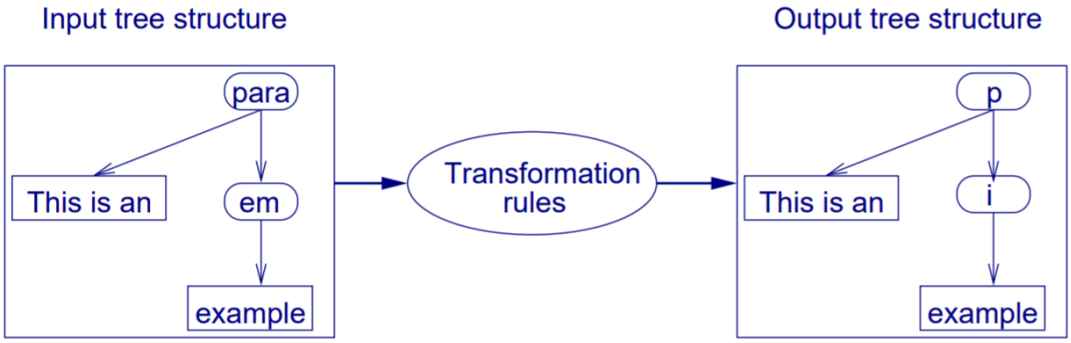

Figure 2.7: XSLT Output Tree Transformation

XSLT language utilizes XML syntax and includes sets of pattern matching rules as mentioned above (Figure 2.8). In this language, available data types are *node-set*, *boolean*, *number*, *string* and *result tree fragment*. Variables are global variables which are accessible everywhere, local variables that are defined in a certain template. XSLT parameters are global parameters which are available in outside of the stylesheet and local parameters that are accessible within a single template. In stylesheet structure, `<xsl:stylesheet>` and `<xsl:transform>`

elements are important points and outermost elements. `<xsl:include>` element enables including a referenced module and `<xsl:import>` element provides definitions in the imported module. Template rules are defined by elements that are `<xsl:template>`, `<xsl:apply-templates>`, `<xsl:call-template>`. `<xsl:value-of>`, `<xsl:element>`, `<xsl:attribute>`, `<xsl:text>`, `<xsl:comment>`, `<xsl:processing-instruction>`. These elements generate output. Variables and parameters are defined by `<xsl:variable>`, `<xsl:param>`, `<xsl:with-param>` elements. Information can be copied from source by using elements that are `<xsl:copy>, <xsl:copy-of>`. `<xsl:if>`, `<xsl:choose>`, `<xsl:when>`, `<xsl:otherwise>`, `<xsl:for-each>` elements enable conditional processing of data. Lastly, final output can be obtained by using elements which are `<xsl:output>`, `<xsl:document>` [13].

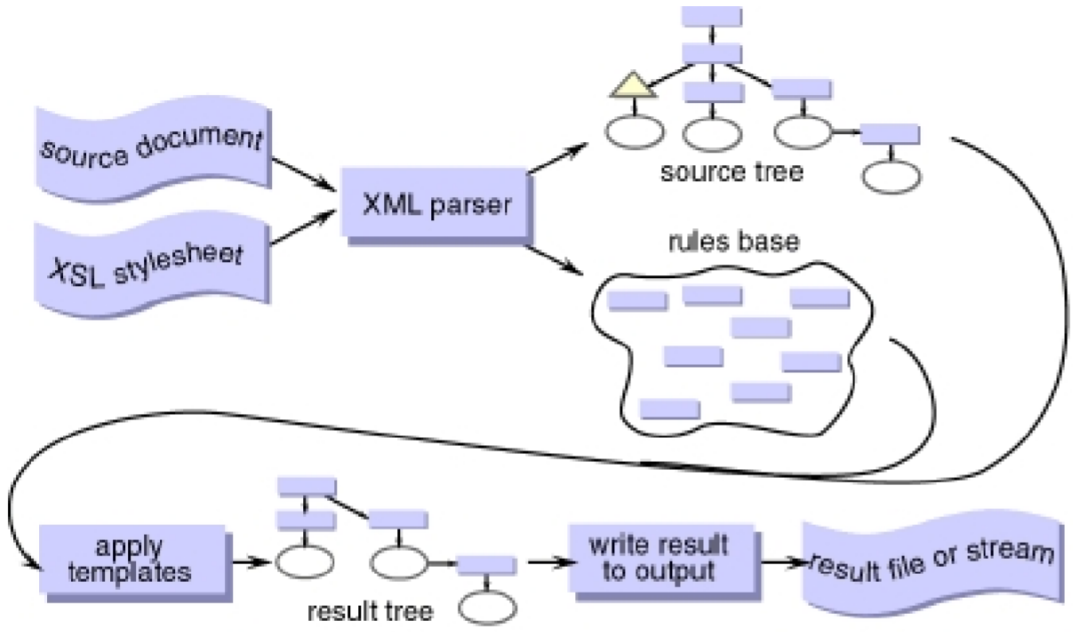

Figure 2.8: XSLT General Overview

2.3.3 XSLT Processors

XSLT processor is a software which enables formatting an XML file into formatted output file. There are several XSLT processors which use XSL standard and support in many programming languages such as SAXON, Xalan, xsltproc, XP, MSXML, 4XSLT and XSLTProcessor [14].

Among these processors, we chose XSLTProcessor to apply semantic validation on client side in the libSBGN.js library. XSLTProcessor is the processor which applies an XSLT stylesheet transformation to an XML document. It is developed by Mozilla and it has many browser support except Internet Explorer. Unlike other processors, it can be used in JavaScript and it operates on client side [15].

2.4 Cytoscape.js

Cytoscape.js is a pure JavaScript open source library that lets users to perform advanced graph analysis and visualization. Cytoscape.js allows users to interact with graphs and it provides users to call user events thanks to using defined functions from client. It can be easily integrated to any project because it nearly supports all browsers [16].

In graph model of this library, there are many graph theory components such as directed graphs, undirected graphs, mixed graphs, loops, multigraphs and compound graphs [16]. There are two components in the architecture of the library which are called core and the collection. A core is the entry point into the library. By using core, programmer or user can perform several operations on graphs. The core enables using several functions that return a collection, set of elements in the graph. These functions deeply provide graph traversals and filtering on graph data. A collection is immutable and new collections are created when different elements in the graph are needed instead of mutation.

Cytoscape.js provides mechanisms, called extensions, for a developer to extend

its functionality and behavior. As an example, the user interface widgets can be constructed, and new automatic layout algorithms may be added.

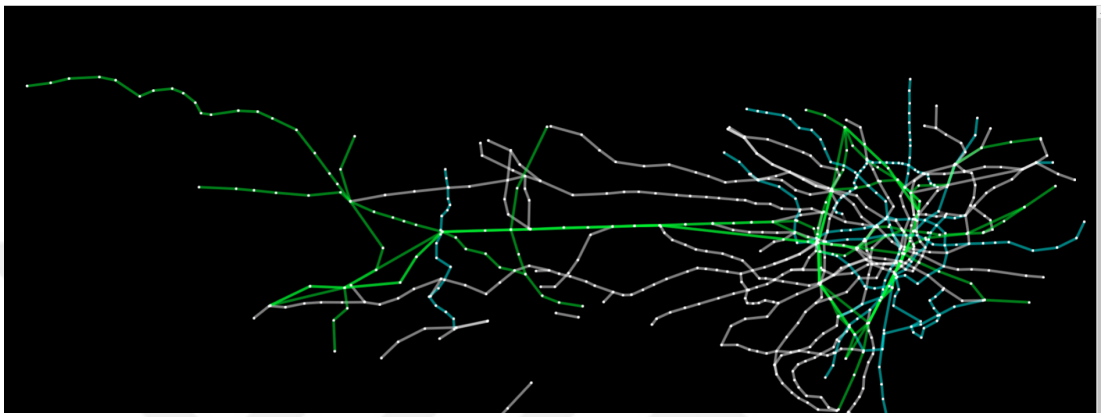

Figure 2.9: Tokyo Railways Plan By Cytoscape.js

2.5 Newt

Newt is a web based library for representing and editing maps in SBGN. It has a layered architecture, where the base is provided by Cytoscape.js and its various extensions such as cytoscape.js-cose-bilkent for automatic layout of SBGN maps with support for compound structures. SBGNViz.js layer is an API responsible for viewing SBGN maps, whereas ChiSE.js provides API for editing such maps [3].

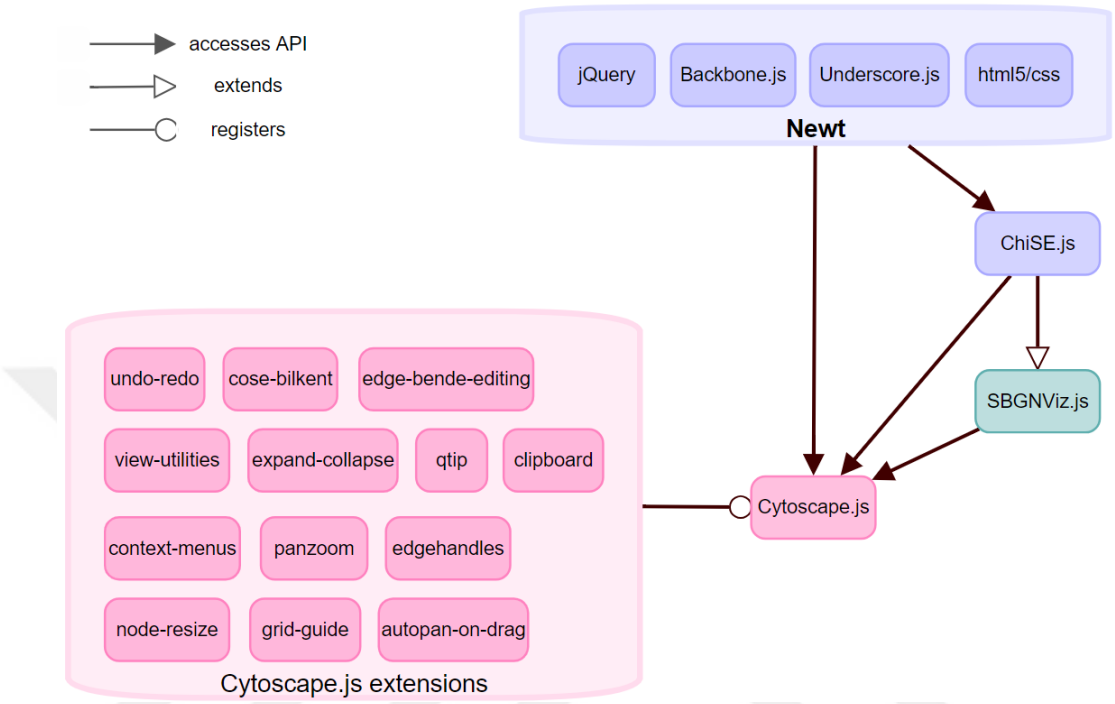

Figure 2.10: Architecture of Newt [3]

Newt has many advantages as an editor for pathways compared to others. First, it is simple to get started with a web-based UI. It provides convenient construction and annotation of pathways, full support for compound structures like automatic layout and facilities for querying, viewing, and editing pathways.

In Newt, Process Description and Activity Flow languages of SBGN notation are supported. An existing map may be uploaded into the editor for editing or a new one can be created . Map objects can be manually or automatically laid out or simply aligned through provided tools. Users can focus on sub-parts in a map according to their interest by removing or hiding other parts in a map to reduce the complexity of larger maps. Users can also expand and collapse compound nodes (i.e., nested maps) as a way of reducing complexity. Newt also enables users to highlight desired parts of maps. Search by node label is one of the many ways highlight can be applied. Lastly, it also provides an interface to perform live queries to the pathway database named Pathway Commons, which is freely available to enable biological pathway and molecular interaction data [17] [3].

Pathway Commons is a database that yields collecting and distributing biological pathways and related data. Data is collected from other related databases and it is stored in BioPAX format. BioPAX facilitates detailed representation of many kinds of biological concepts such as biochemical reactions, gene regulatory networks and genetic interactions [18].

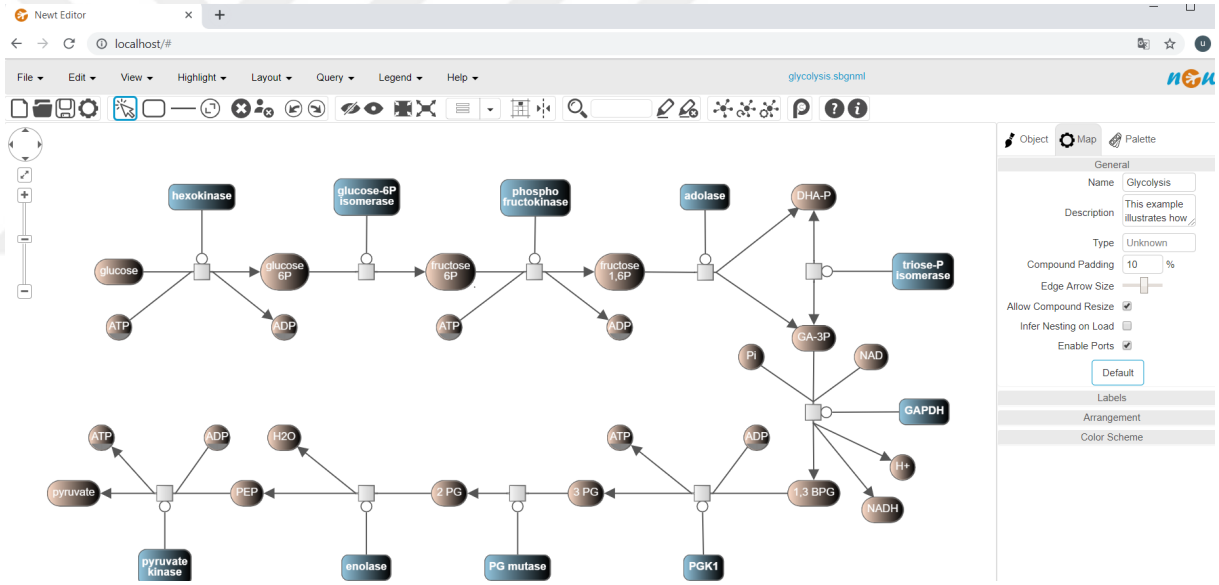

Figure 2.11: Sample screenshot from the Newt Editor

Chapter 3

Semantic Validation in libSBGN.js Library

3.1 Adding Semantic Validation

As mentioned earlier, libSBGN.js library provides syntactic validation which ensures that an XML file conforms to a specified structure. On the other hand, semantic validation which yields affirmations about the existence or inexistence of patterns in XML trees was not provided prior to this work. With this thesis, we incorporated a schematron validator to the libSBGN.js library to add semantic rule checks.

Specifically we implemented a schematron validator by using Mozilla's XSLT processor and DOM parser. In addition to processing of the schematron validation, a well-organized report handling was implemented.

3.1.1 Producing XSLT Stylesheet File

For the purpose of adding Schematron validation to the libSBGN.js library, firstly, we have to execute XSLT, which requires an XSLT stylesheet file. Producing this XSLT stylesheet file needs external XSLT files which are readily available in the Java version of libSBGN library. Specifically we used `iso_svrl_for_xslt2.xsl` file in the Java version of libSBGN library as a stylesheet for this external XSLT. By using this XSLT stylesheet file, we transformed `sbgn_pd.sch` file in the Java version of libSBGN library.

XSLT stylesheet file (`iso_svrl_for_xslt2.xsl`) is designed to run with "Skeleton" implementation of Schematron language. The skeleton provides basic templates for handling outputs with Schematron implementation. Thus, this file is a source for Schematron implementation with named templates.

Other file (`sbgn_pd.sch`)helps defining rule patterns for Schematron validation in process description manner and it also introduces error messages for every rule pattern.

As a result, we transformed rule description file for schematron validation in process description manner with the help of external XSLT processor by using template file (`iso_svrl_for_xslt2.xsl`) as a sylesheet file. The output file is the stylesheet file for other XSLT needed for adding Schematron validation to the libSBGN.js library. We copied this output file to the file structure of libSBGN.js library for direct usage.

3.1.2 DOM Parser

DOM Parser deals with XML as an object graph in memory named Document Object Model (DOM). The parser crosses input XML file and produces DOM objects represents elements in XML file.

In the Javascript version of DOM parser library, all browsers have their own

XML parsers that can convert text into DOM objects. These parsers enable parsing XML content from a string to DOM objects. We utilize DOM parser to parse content of file which will be validated. We use `parseFromString` method of DOM parser for this process. This method takes two required arguments: content of the file as a string and a string that determines returned object type (we especially give `text/xml` as this string). This method returns `DOMObject` or `Document` according to type argument.

Methods

`DOMParser.parseFromString()`

Syntax

```
var doc = domparser.parseFromString(string, mimeType)
```

Return

Either `Document` or `XMLDocument` depending on the **`mimeType`** argument.

Parameters

The method accepts 2 arguments (both are required):

`string`

The `DOMString` to be parsed. It must contain either HTML, xml, xhtml+xml or svg document.

`mimeType`

A `DOMString`. This string determines a class of the the method's return value. The possible values are the following:

Figure 3.1: DOM Parser parseFromString Method

With the help of this library, DOM object portion can be changed with new DOM objects by using the value of the `Element.innerHTML` and `outerHTML` properties. We set these properties for appending semantic validation result object to DOM by using `innerHTML` property as shown in Figure 3.2.

```
var tmp = document.createElement("div");
tmp.appendChild(result);
result = tmp.innerHTML;
```

Figure 3.2: DOM Parser innerHTML Property

3.1.3 XSLT Processor Model

XSLT processor enables stylesheet transformation that includes producing a new XML document as output by transforming input XML file. In order to add client side semantic validation into the libSBGN.js library, we use Mozilla's XSLT processor in this work. We utilize obtained XSLT stylesheet file which is described in previous subsection during this transformation. The file to be validated is transformed by using this stylesheet file with the help of an XSLT processor. We acquire an output file containing the result of semantic validation.

Mozilla's XSLT processor facilitates methods for importing stylesheet file, manipulating paramater values and applying desired transformation. In this work, we were able to import the stylesheet file by using provided function. First, we get the content of stylesheet by using `XMLHttpRequest` function of the library. We are then able to initialize the request for file reading by using the `open()`method. This method takes parameters which are method type, URL and a boolean that indicates asynchronization condition. We set `GET` as method type and assigned file name as URL as shown in Figure 3.3. We also execute synchronous file read by setting this parameter. We determine file content type as `msxml-document` by assigning `responseType` property. Finally, we can obtain the file content by using `XMLHttpRequest.responseXML` property with a `XMLHttpRequest`. This property returns the data which includes the `Document` containing the HTML or XML. Then, we are capable of determining the stylesheet file for inner XSLT which is needed for adding semantic validation by using `importStylesheet` function in Mozilla's XSLT processor library (Figure 3.4). This method takes a parameter for file content as a `Document` node. One can give the file content as a `Document`

node, which is a parameter for this function.

```
xhttp.open("GET", filename, false);
try {xhttp.responseType = "msxml-document"} catch(err) {} // Helping IE11
xhttp.send("");
return xhttp.responseXML;
```

Figure 3.3: XSLT Processor File Read

In addition to these, we apply XSLT by using `transformToFragment` function in our implementation of XSLT as seen in Figure 3.4. This function takes two required parameters. One of them is a `Document` obtained from DOM parsing of the file which will be validated. The other one is an owner `Document`. This function transforms node by using stylesheet file and returns an XMLDocument object as a output. Hence, we can obtain validation result as an `XMLDocument` object with the help of this function.

```
xsltProcessor = new XSLTProcessor();
xsltProcessor.importStylesheet(isoContent);
result = xsltProcessor.transformToFragment(xml, document);
```

Figure 3.4: XSLT Processor Transform Method

3.1.4 Validation Result Reporting

For the purpose of reporting validation results effectively, we parse the `XMLDocument` obtained from XSLT and produce output tags for validation result with the help of xml2js, a node.js library. This library provides XML parsing as simple and easy as possible with the help of the `parseString` function.

After producing output tags, we can determine whether the file is valid or not according to rules for semantic validation. We are able to check all tags obtained from XML parsing of output file that is produced from XSLT. Unless `svrl:failed-assert` tag is produced, the file is not erroneous and is assumed to

be in compliance with SBGN. In such cases, we simply return an empty list as the validation result. If this tag exists, however, it indicates that the file has one or more problems in defined rules with respect to semantic validation. To represent errors in invalid files effectively, we use an object-based design, and create an instance of an `Issue` and fill out every property of this object for erroneous file as exemplified in Figure 3.5. Each `Issue` object includes three properties: *text*, *role* and *pattern*. Here the text indicates error message for failures in a certain rule; the pattern represents the rule name, and finally, the role shows identity information of the element which is erroneous in the file. To conclude, we return array of `Issue` objects in order to demonstrate all failures about semantic validation for a validated file. Every element in the array indicates information about associated error in terms of rule name, error message and erroneous part of file.

```
var errors = [];
if(parsedResult["svrl:schematron-output"]["svrl:failed-assert"] == undefined)
    return errors;
var errCount= parsedResult["svrl:schematron-output"]["svrl:failed-assert"].length;
for(var i=0;i<errCount;i++){
   var error = new Issue();
   error.setText(parsedResult["svrl:schematron-output"]
       ["svrl:failed-assert"][i]["svrl:text"]);
   error.setPattern(parsedResult["svrl:schematron-output"]
       ["svrl:failed-assert"][i]["$"]["id"]);
   error.setRole(parsedResult["svrl:schematron-output"]["svrl:failed-assert"]
       [i]["svrl:diagnostic-reference"][0]["_"]);
   errors.push(error);
}
```

Figure 3.5: Semantic Validation in Result Reporting

Rules are determined in the stylesheet file for inner XSLT along with error messages for every rule as seen in Table 3.1.

Table 3.1: Error Descriptions For Rules

Rule Code	Description
pd10101	Arc with class consumption must have source reference to glyph of EPN classes
pd10102	Arc with class production must have target reference to glyph of PN classes

Table 3.1: Error Descriptions For Rules

Rule Code	Description
pd10103	The 'source and sink' glyph can be connected to at most one consumption arc
pd10104	The 'dissociation' glyph can only be connected to one consumption glyph]
pd10105	Arc with class production must have target reference to glyph of EPN classes
pd10106	Arc with class production must have target reference to glyph of EPN classes
pd10107	The 'source and sink' glyph can be connected to at most one production glyph
pd10108	The association glyph can only be connected to one production glyph
pd10109	Modulation arc must have source reference to glyph of EPN classes or a logical operator
pd10110	Modulation arc must have target reference to PN classes
pd10111	'and', 'or', and 'not' glyphs must be the source for exactly one arc
pd10112	If there are compartments defined, top-level glyphs must have a compartmentRef
pd10124	Arc with class logic arc must have source reference to glyph of EPN classes, or logic gates
pd10125	Arc with class logic arc must have target reference to a logical operator
pd10126	The 'not' glyph can only be the target of one logic arc glyph
pd10127	Arc with class equivalence arc must have source reference to glyph of EPN classes
pd10128	Arc with class equivalence arc must have target reference to glyph of classes 'tag', 'submap' or 'terminal'
pd10129	All state variables associated with a Stateful Entity Pool Node should be unique and not duplicated within that node.
pd10131	EPNs should not be orphaned (i.e. they must be associated with at least one arc)

Table 3.1: Error Descriptions For Rules

Rule Code	Description
pd10132	All process nodes (with the exception of phenotype) must have an LHS and RHS.
pd10133	All EPNs on the LHS of a process must be unique.
pd10134	If more than one set of stoichiometries can be applied to the flux arcs of the process then the stoichiometry of the flux arcs must be displayed.
pd10135	If the stoichiometry is undefined or unknown this should be indicated by the use of a question mark (“?”)
pd10140	This ’glyph class’ is not allowed in Process Description
pd10141	All process nodes should have at least one input and at least one output pointing to the arcs.
pd10142	Logic arc must be connected to either ’OR’, ’AND’ or ’NOT’

3.2 Testing of Semantic Validation

With the aim of testing added semantic validation in the libSBGN.js libray, we wrote a unit test file called `schematronValidatorTest` as represented in Figure 3.6. This test file performs semantic validation property testing for a specified file. A sample file content is read and the test method compares expected result with real result obtained from semantic validation. If sample file is considered as valid, test file expects that empty result array is returned. Otherwise, when the sample file is invalid, the test file expects a non-empty array is returned with specific content.

```
var schematronValidator = require('./schematronValidator');
var fs = require('file-system');
var file=fs.readFileSync('pd10101-fail.sbgn.xml', 'utf8');
console.log(file);
var result = schematronValidator.SchematronValidation.isValid(file);
console.log(result.length > 0);
```

Figure 3.6: Semantic Validation Testing

In fact, as sample test files for Schematron validation testing in the libSBGN.js library we were able to use the ones available in the Java version of libSBGN library. We had a chance to fix programming errors faced in schematron validation implementation in the libSBGN.js library quickly with the help of the testing process including usage of these sample files.

3.3 Architecture

As previously mentioned, the libSBGN.js library enables manipulating, reading and writing SBGN-ML files and syntax validation of these files. The library and the support for semantic validation have been designed in a modular manner; hence, this library can be used as a dependency for projects requiring processing of SBGN-ML files. This was achieved by putting the stylesheet file directly in the file structure of the libSBGN.js library and adding new semantic validation related methods to the main class in the file `libsbgn.js`. This enables seamless integration of semantic validation facilities into existing software tools. As an example, such support was easily introduced into Newt as will be described in the next chapter.

Chapter 4

Adding Semantic Validation to Newt

4.1 Changes in Associated Libraries

As explained in the previous chapter, semantic validation was added to the libSBGN.js library. These changes were pushed to a new branch called `develop` on related github repository to be merged into the `master` branch for a public release in the near future. Newt does not directly use the libSBGN.js library but it reaches out to this library with the help of another library named SBGNViz.js [19] . This required some changes in the SBGNViz.js library as well.

To realize required changes in the SBGNViz.js library for adding semantic validation checks to the libSBGN.js library, we added a new method providing validation of SBGN-ML files, which simply delegates to the associated method in the libSBGN.js library. We pushed all these changes to the `unstable` branch of the SBGNViz.js library. In order to use required functionality in the libSBGN.js library, we changed libSBGN.js dependency in the SBGNViz.js library as well. In the Newt repository, SBGNViz.js library was already used as dependency, so we also changed this dependency to point to the branch with new changes for

adding semantic validation control to the Newt.

4.2 Validation User Interface in Newt

Before this thesis, Newt did not provide Schematron validation control and only performed syntactic validation checks with a simple XSD file. After making needed changes in the libSBGN.js library and SBGNViz.js library as described in the previous section, changes in graphical user interface of the tool were made to expose the functionality to the users in a user friendly and intuitive manner. Visually, we added one button in the toolbar with a thick symbol and "Validate Map" tooltip description as shown in Figure 4.1 and an item under the View menu with "Validate Map" label as seen in Figure 4.2. This button or the menu item triggers the action to validate the currently loaded SBGN-ML map. This executes the required method in the SBGNViz.js library which in turn reaches out to the needed functionality in the libSBGN.js library for performing the validation and getting the results back. The validation result report is obtained as an array.

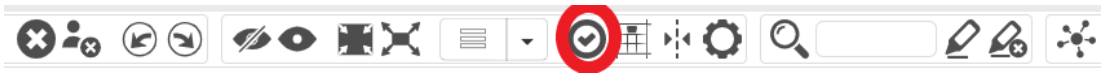

Figure 4.1: Validate Map Button in Toolbar

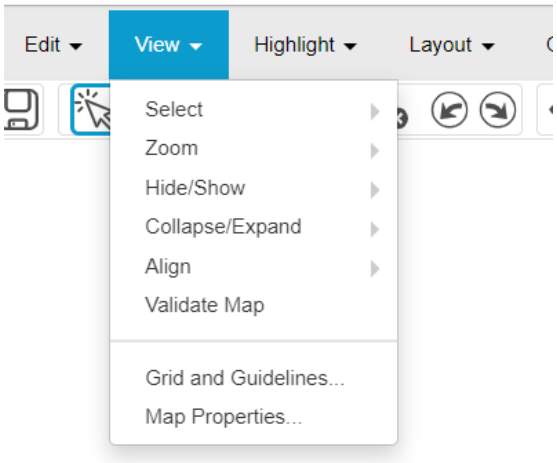

Figure 4.2: Validate Map Link Under View Menu

We present the results of semantic validation to the users through a new tab in the right panel called Console. In this tab, we report validation result messages, highlight any erroneous part(s) of the map, and, where applicable, suggest a fix for the problem as detailed in the next subsections.

4.2.1 Providing Validation Result Messages

As noted earlier, upon a semantic validation check, Newt displays a dynamic Console tab in the right panel to report the results. This tab shows basically the validation results according to the acquired validation result array for the loaded SBGN-ML file (i.e., the current SBGN map) in the tool. Interaction with other parts of Newt including the menu and the toolbar, the canvas and other tabs in the right panel are disabled so that the user is not allowed to make changes to the map as the results are reported. This is especially crucial as the map might be changed by the validation process, which in turn might make validation results invalid, resulting in an inconsistency.

Since currently only SBGN PD maps can be validated and Newt supports other map types, we first make sure that the current map is of type PD. If the file's map type is not PD, we report this to the user in red as shown in Figure 4.3. When the user clicks the Dismiss button, the Console tab is hidden and the disabled user interaction is back to normal.

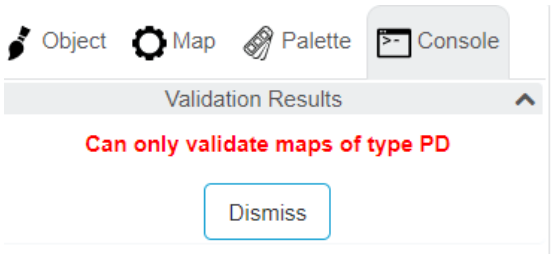

Figure 4.3: Console Tab Seen for an Incorrect Map

Assuming the map is of type PD, validation checks are performed. In case, the map is found to be in compliance with SBGN PD language, the Console displays

a “Map is valid” message in green. Again, a “Dismiss” button is displayed to go back to regular interaction and editing mode (Figure 4.4).

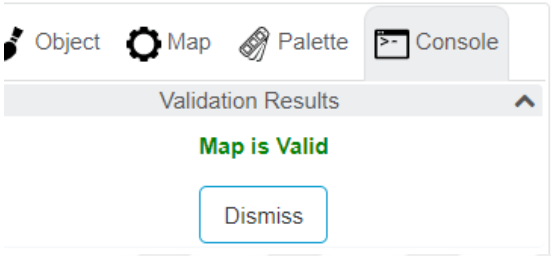

Figure 4.4: Console Tab Seen for a Valid Map

In cases, where the validated SBGN-ML map fails with at least one rule for semantic validation, Newt displays a “Map is invalid” message in the Console tab in red (Figure 4.5). Specified error message for failed rule is also shown on the Console tab. This error message is obtained from validation result array. As described in the previous chapter, validation result array contains an Issue object for every error in every entry and this object’s text property gives this error message. “Next” and “Previous” buttons are also added appropriately when the file has more than one error. A “Dismiss” button is also added for completing validation reporting and going back to regular editing mode.

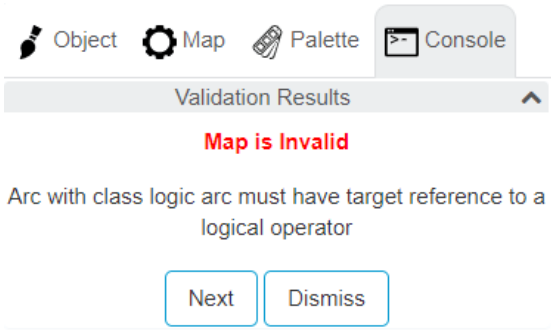

Figure 4.5: Console Tab Seen for an Invalid Map

Furthermore, when applicable we suggest a fix for each error (available for most rules) in Schematron validation. For this purpose, a fix button with process sign and a sentence revealing the suggested fix are shown. This button executes

the suggestion, fixing the related part of the map. In cases, where the user might select among multiple possible fixes, a radio button group is provided. For example, the user might be opted to choose among multiple logical operators available in a map for reconnecting a logic arc with an invalid source. Every radio button in radio button group represents one possible way for correcting the error and only one of them is checked as default when the Console tab is dynamically created. These settings will also be described in detail in related subsections (Figure 4.6).

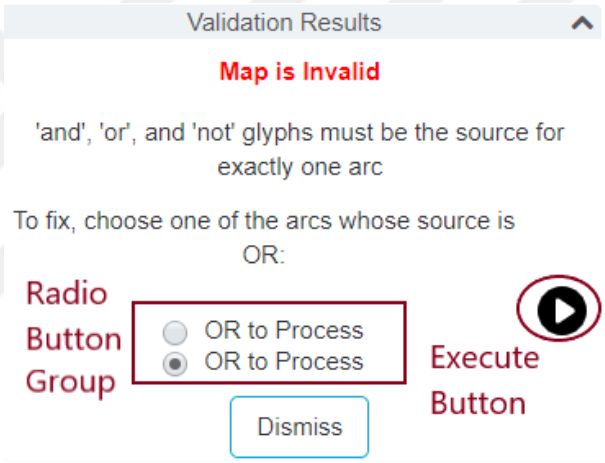

Figure 4.6: Console Tab Seen for Error Fix Suggestions

4.2.2 Highlighting Erroneous Parts on Map

A crucial part of the validation process in Newt is proper visual highlighting of the erroneous part(s) on the map. As stated previously, a validation result array is returned from the call to the SBGNViz.js library and the libSBGN.js library. This result array includes an Issue object and *role* property of this object instance gives identity information of erroneous elements in the graph. By using the `elements()` function in Cytoscape.js library with this identity information, one can get a handle to the object instance of the associated erroneous element in the map.

Cytoscape.js library allows highlighting map elements in a number of different colors. We highlight erroneous edges and/or nodes in the map in red as exemplified in Figure 4.7. We use the `highlight()` function in the view-utilities extension of Cytoscape.js library and this function requires a parameter which is the instance of object of element to be highlighted.

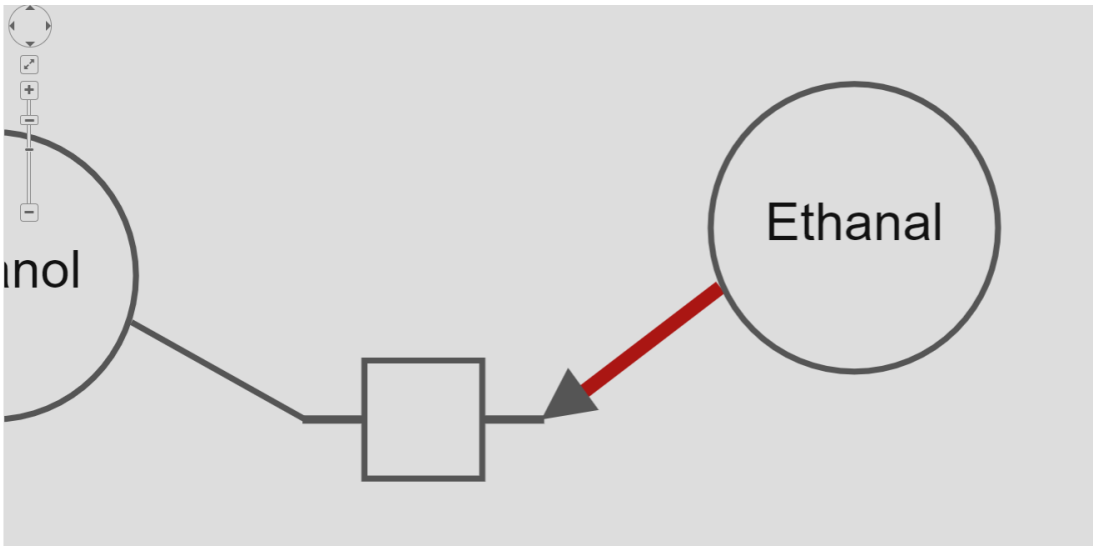

Figure 4.7: Highlighting Erroneous Elements in Graph

For failures in some rules, highlighting only erroneous elements is not enough. Elements that are directly or indirectly connected with the erroneous element might also need to be highlighted as shown in Figure 4.8.

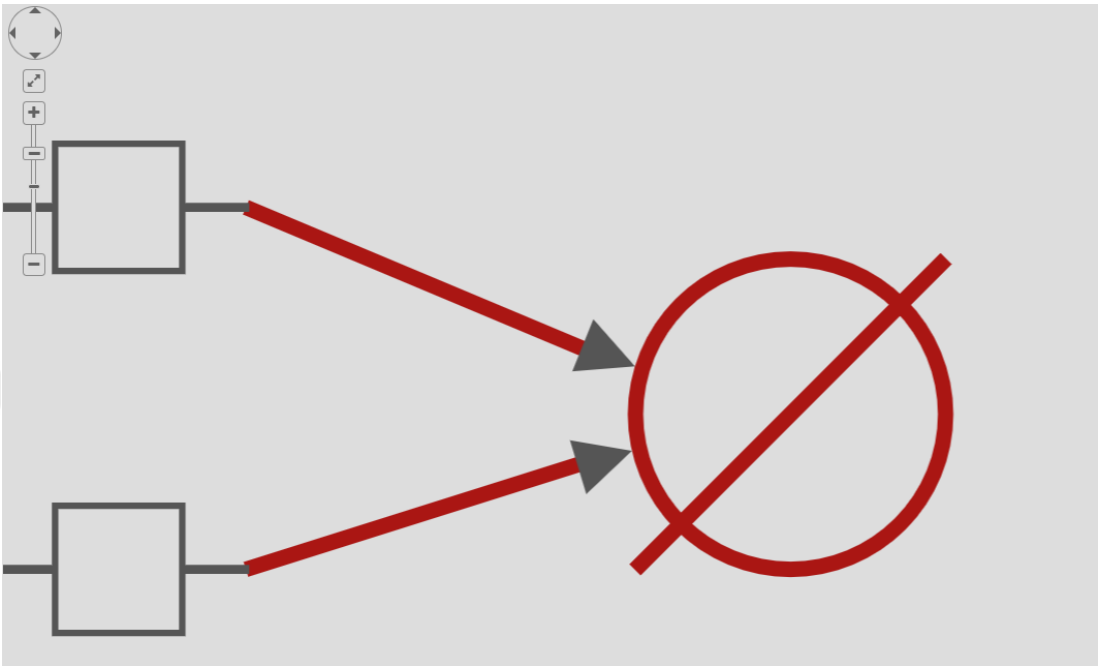

Figure 4.8: Highlighting Connected Elements in Graph

As described in previous subsections, a graph might fail in more than one rule for semantic validation. In such cases, the first error in the validation result array is displayed in the Console tab and the erroneous part(s) of the map is highlighted.

Newt also provides a suggestion to fix for most failing rules. As noted earlier, to implement this we add a fix action button in the Console tab and radio button group for allowing user to choose in which manner to apply the fix. Each choice is either a node or an edge. Even though we try to display node labels for the nodes and source and target node labels for the edges to uniquely identify these choices, in cases where nodes do not have labels (such as process nodes), we also highlight the associated node or the edge on the map. Thus, the user can click on different radio buttons to see a new choice highlighted in the map and can make an informed decision.

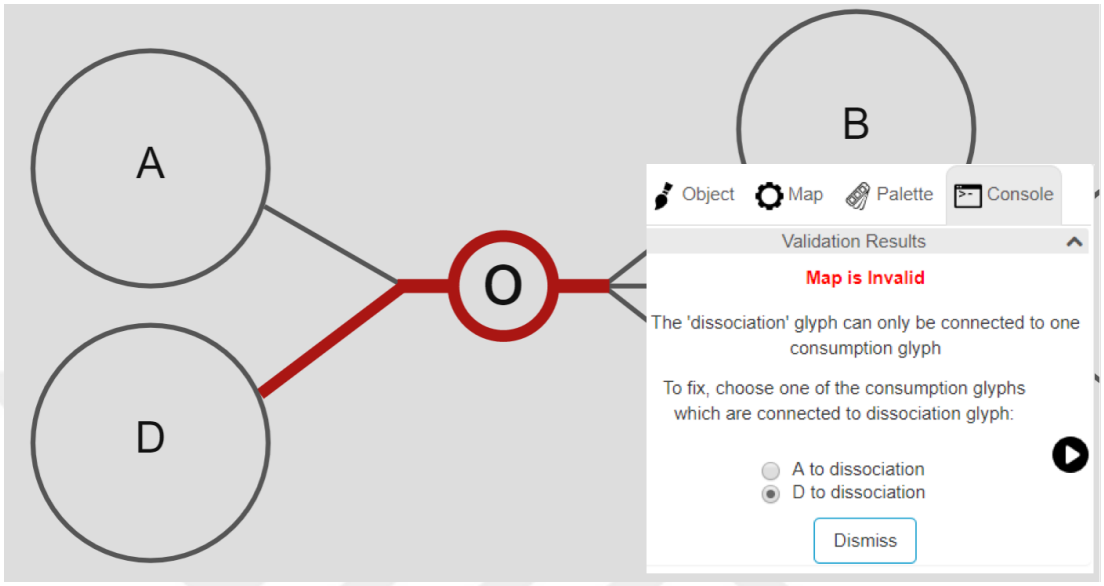

Figure 4.9: Highlighting Default Radio Button Option

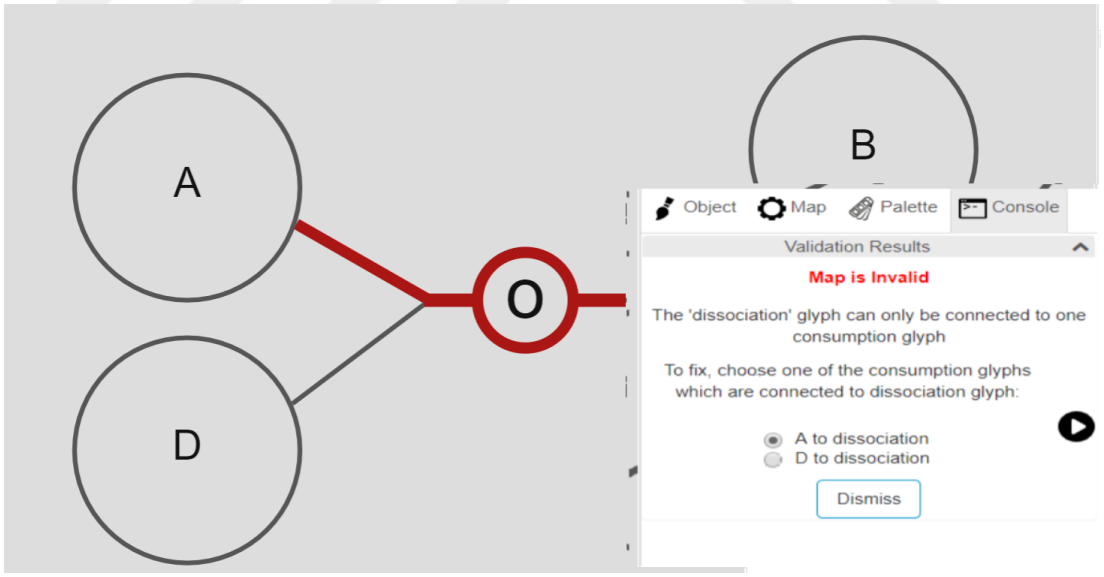

Figure 4.10: Highlighting Chosen Radio Button Option

Once a decision is made and the suggested fix is applied, Newt reruns validation to eliminate the error that was just fixed and re-display any remaining errors. Note that sometimes application of a fix might result in a new invalid situation; thus a re-run is necessary.

4.2.3 Suggesting a Fix

As expressed before, we provide error fix suggestions for failures in most rules for semantic validation. In order to do so, an error fix button as well as a sentence explaining the fix suggestion was added to the Console tab when the particular error is shown to the user. The fix button will invoke Cytoscape.js core methods to change the map topology to fix the specific problem. The map is re-validated to determine whether or not any more fixes remain. Notice that sometimes fixing one problem might lead to another problem. These fix suggestions for each error will be detailed in this subsection.

With rule pd10101, failure is that consumption edge does not have source node with EPN classes as exemplified in Figure 4.11. Similarly with rule pd10102, a consumption edge does not have target node with PN classes. To fix a failure with either of these rules, we suggest that consumption edge's source and target nodes swapped to reverse the arc. To realize this, source and target nodes are swapped by using edge `move()` function in Cytoscape.js library as represented in Figure 4.12. `portsource` and `porttarget` attributes of the edge are also changed by setting these properties.

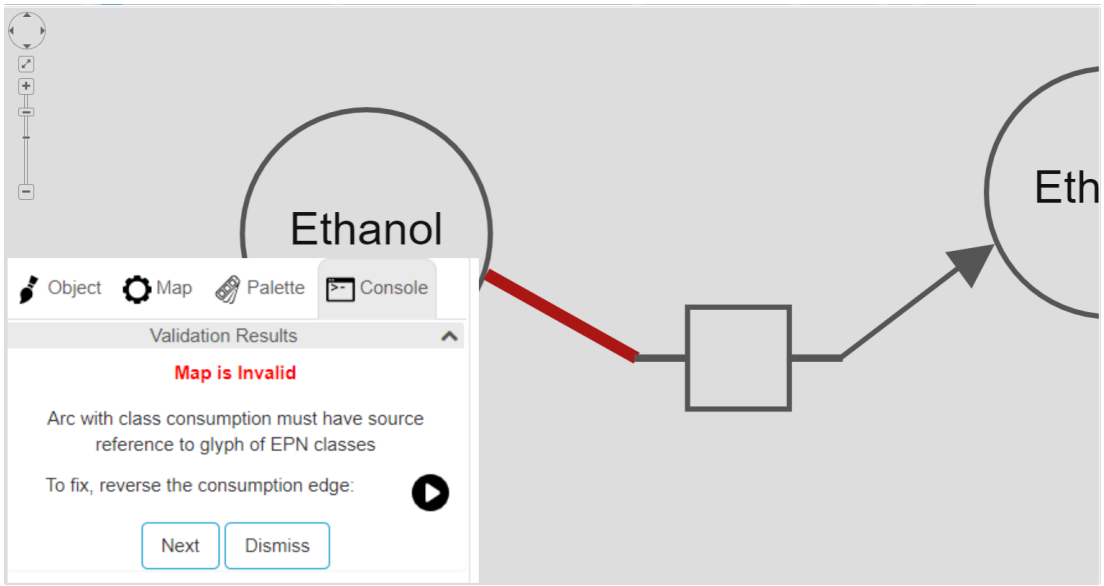

Figure 4.11: An example where rule pd10101 is violated

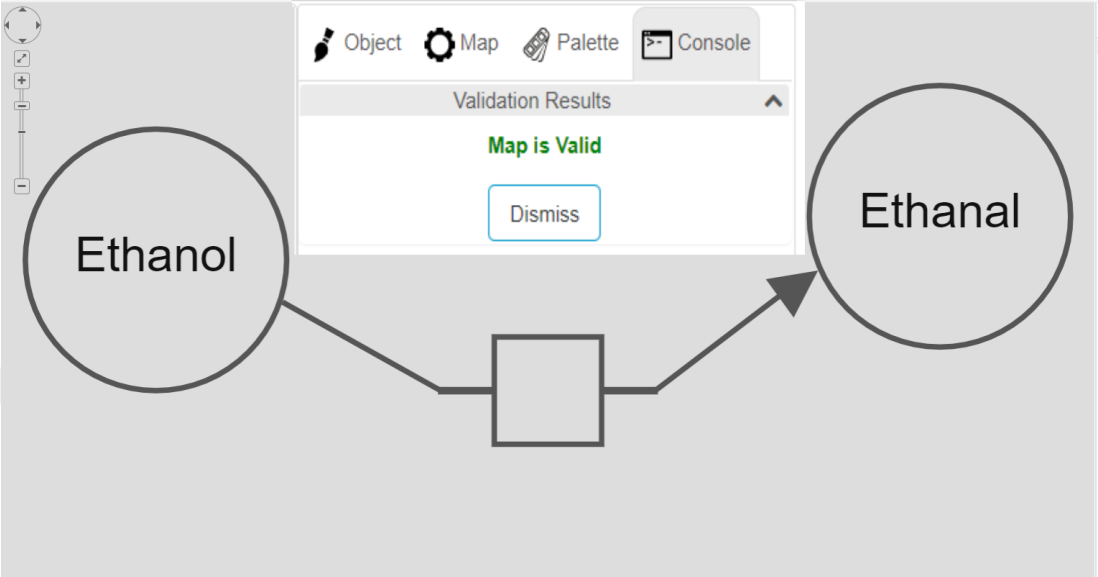

Figure 4.12: After the rule pd10101 violation is fixed

Validation failure in pd10103 rule means that source and sink glyph is connected to more than one consumption glyph. A symmteric case is in pd10107 rule where source and sink glyph is connected to more than one production glyph as seen in Figure 4.13. To correct such failures with pd10103 and pd10107, source and sink glyph is suggested to be replicated in appropriate positions. With pd10103 rule, each consumption arc connected to the erroneous node is also replicated. Similarly with pd10107 rule, each production arc connected to the erroneous node is replicated as shown with an example in Figure 4.14. These fixes include creation of new nodes and edges at appropriate locations using respective Newt methods of `addNode()` and `addEdge()`. Lastly, the erroneous node itself is removed by Cytoscape.js library's `remove()` function.

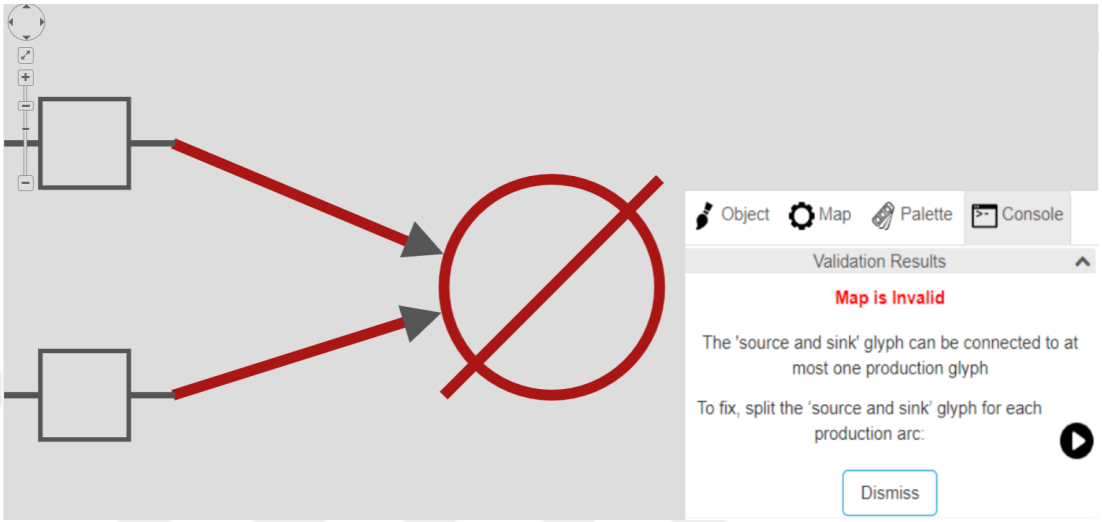

Figure 4.13: An example where rule pd10107 is violated

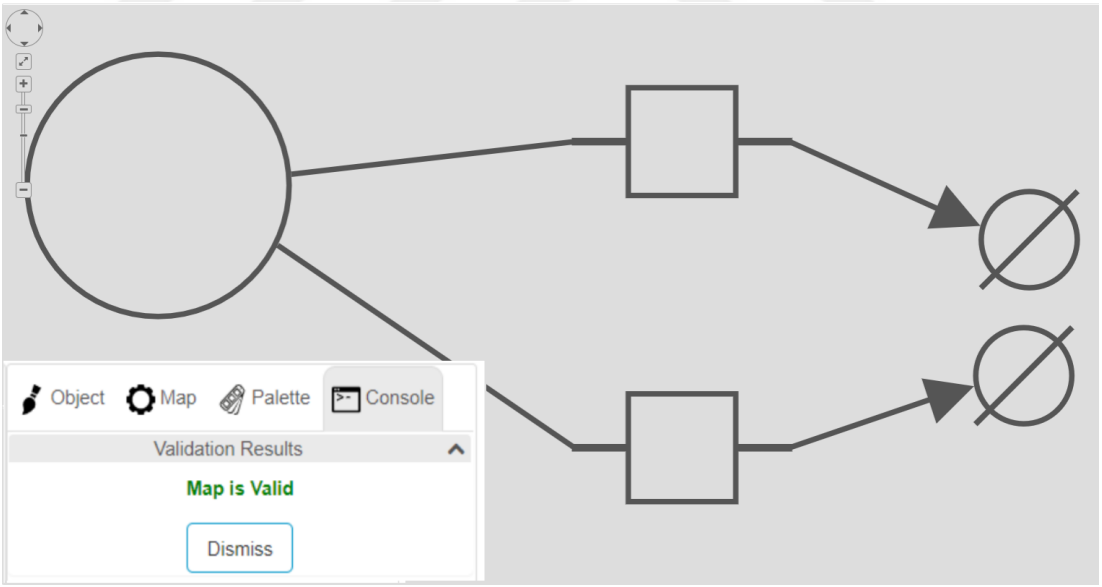

Figure 4.14: After the rule pd10107 violation is fixed

When rule pd10104 fails, it means that a dissociation node is connected to more than one consumption edge. As a dissociation node is allowed to only have a single source, we let the user choose one of the sources and suggest removing the others through a radio button group. These edges are represented with a description including source node's label name and dissociation node's type name

as exemplified in Figure 4.15. In handling of fix action, chosen edge remains and other connected edges are destroyed by using Cytoscape.js library's `remove()` function as exemplified in Figure 4.16.

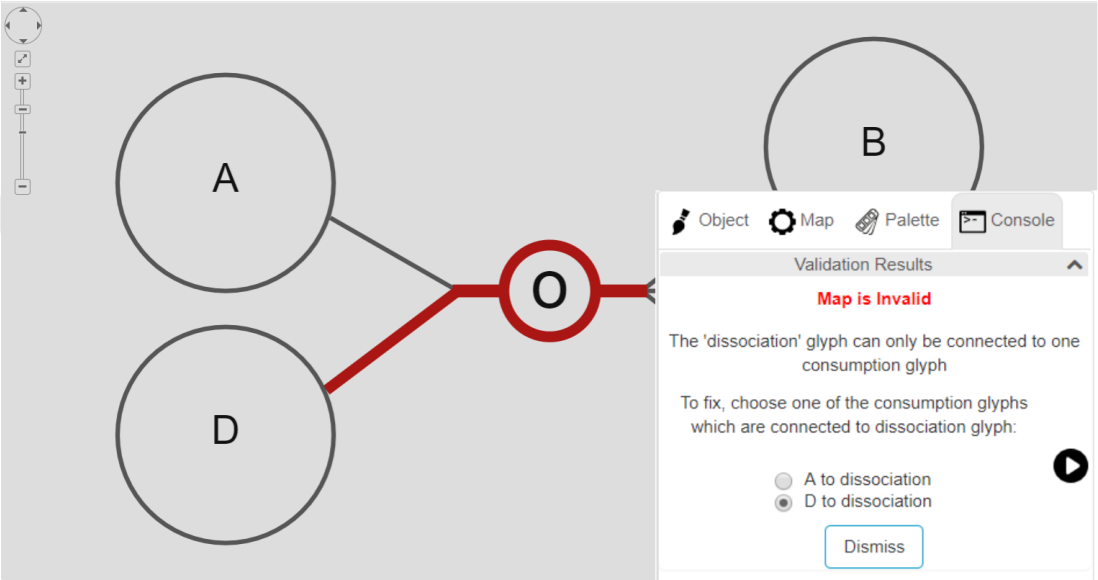

Figure 4.15: An example where rule pd10104 is violated

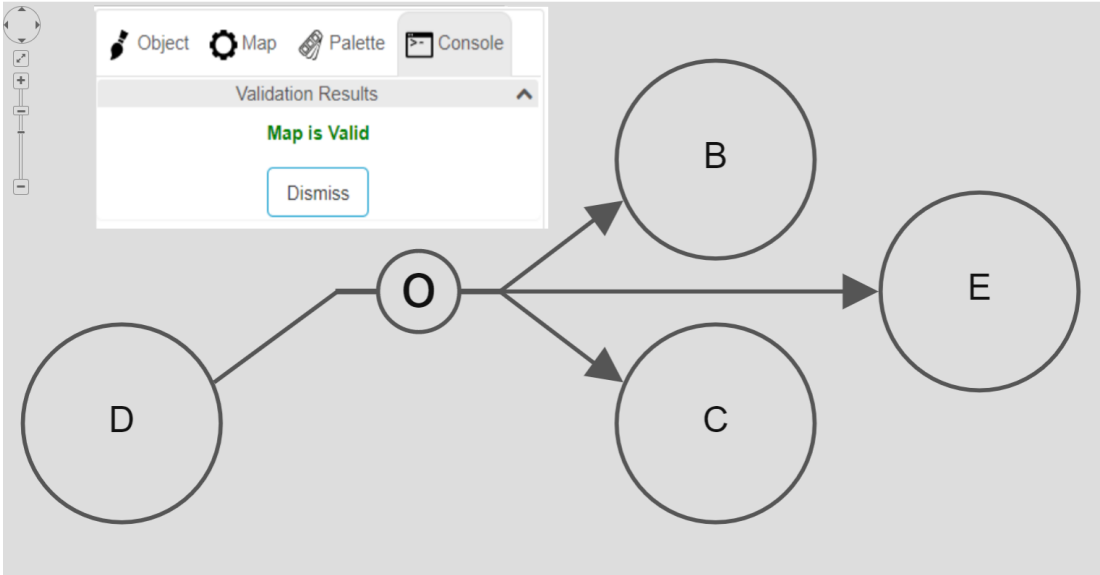

Figure 4.16: After the rule pd10104 violation is fixed

When rule pd10105 fails, it means that production edge does not have source

with PN classes as seen in Figure 4.17. Similarly with rule pd10106, production edge does not have target with PN classes. To fix errors in pd10105 and pd10106 rules, we offer that production edge source and target are reversed. In fulfilling the fix, source and target nodes are reversed by utilizing edge's `move()` function in Cytoscape.js library as seen in Figure 4.18. `portsource` and `porttarget` attribute of the edge are also altered by setting associated properties.

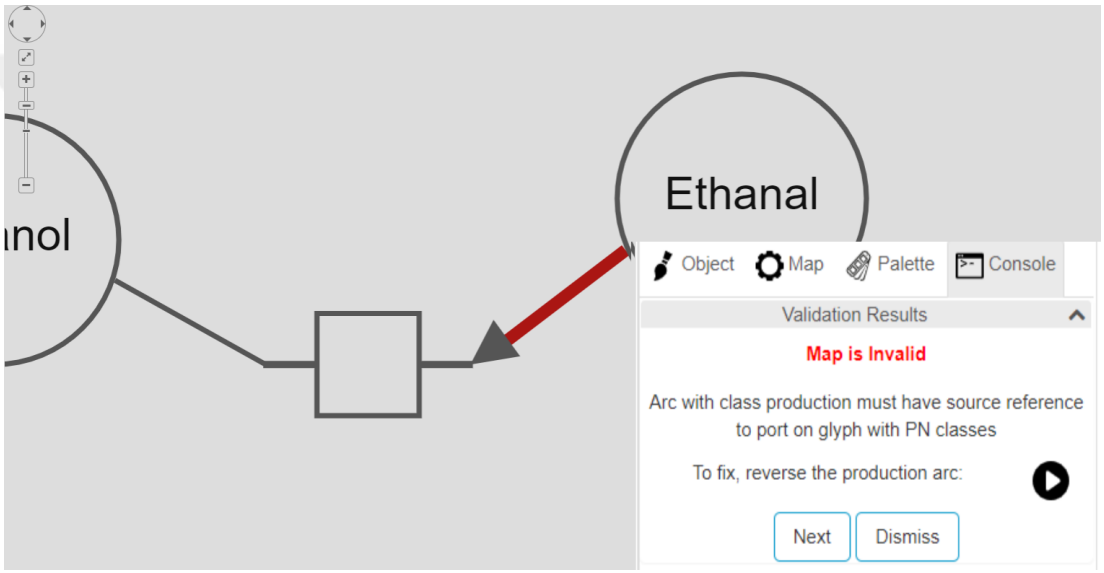

Figure 4.17: An example where rule pd10105 is violated

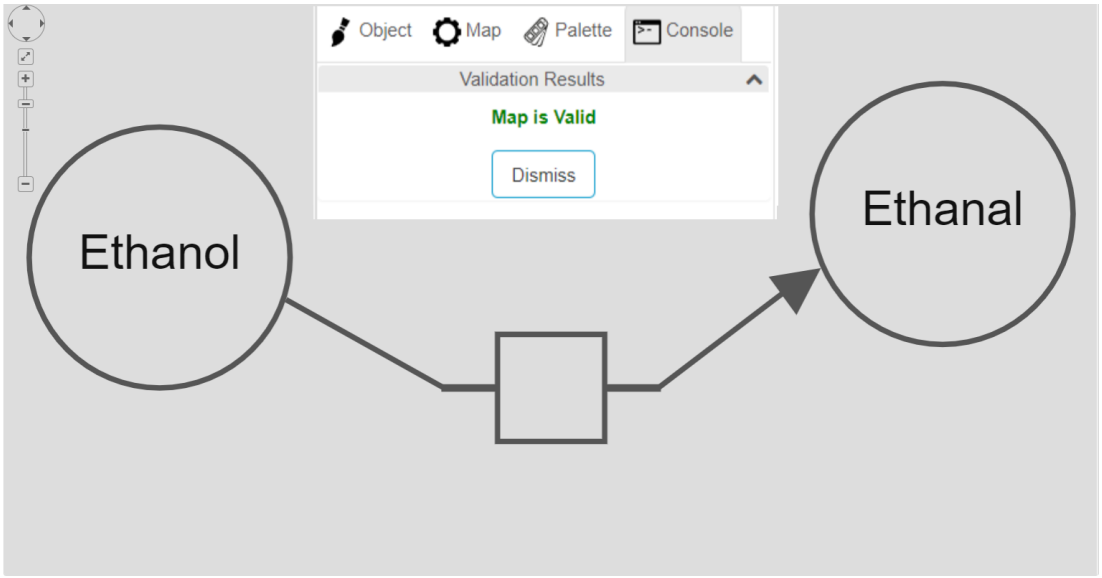

Figure 4.18: After the rule pd10105 violation is fixed

Rule pd10108 ensures that an association node is not connected to more than one production edge. The user is presented with a radio button group representing each possible edge as the only production edge. These edges are represented with a description containing association node's type name and target node's label name as exemplified in Figure 4.19. Upon executing a fix, the chosen edge remains and other connected edges are deleted by invoking Cytoscape.js library's `remove()` function as shown in Figure 4.20.

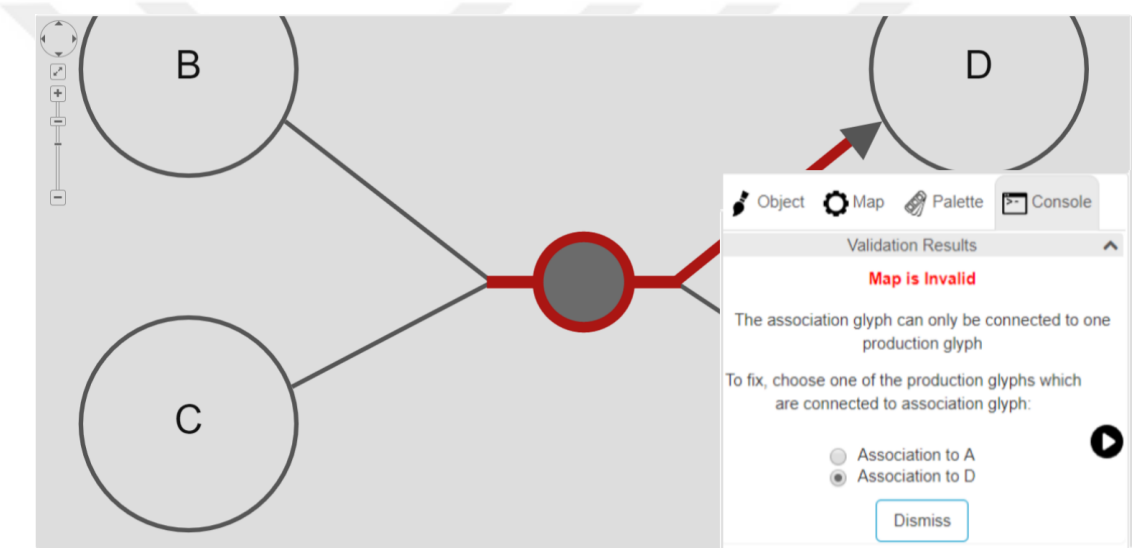

Figure 4.19: An example where rule pd10108 is violated

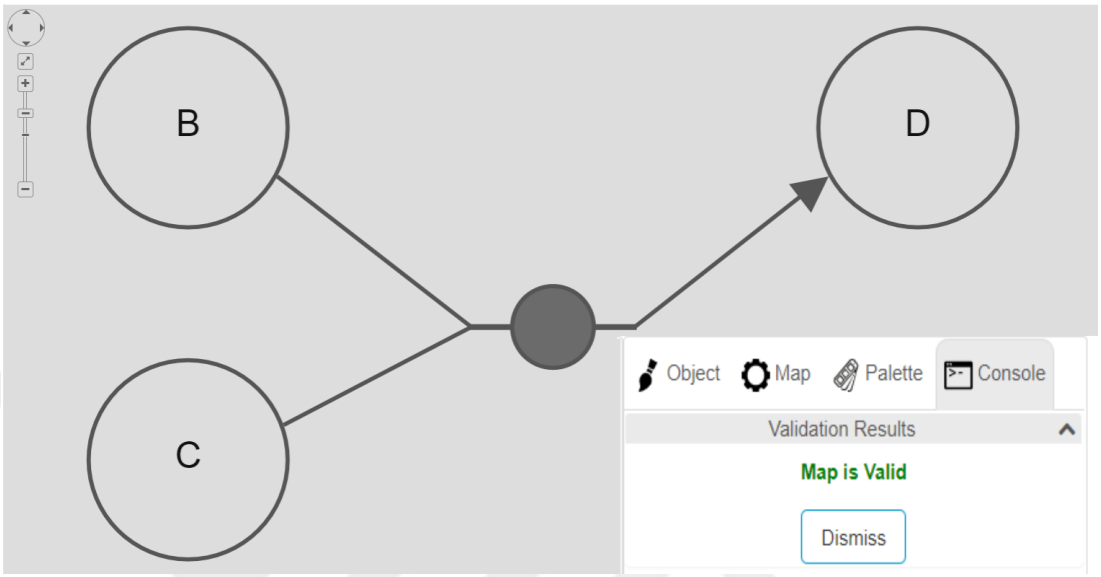

Figure 4.20: After the rule pd10108 violation is fixed

Another type of failure with code pd10109 occurs when a modulation arc does not have a source node with EPN class or logical operator. Similar with this rule is pd10124 rule, which occurs when a logic arc does not have source node with EPN class or logic gate as exemplified in Figure 4.21. Similarly, with pd10127, an equivalence edge does not have source with EPN class. For correcting failures with all of pd10109, pd10124 and pd10127, we suggest the user to choose one nearby appropriate node to connect the associated arc in fault. Such nodes' label names are presented to the user as a radio button group. Here we define a node *nearby* if that node's distance to source or target node of the arc in fault is not greater than that of an ideal edge length as defined in Newt. The fix is to simply set the chosen node as source node or target node of the erroneous arc as exemplified in Figure 4.22. To realize this fix, Cytoscape.js library's `move()` function is used and `portsource` attribute of the arc is set accordingly.

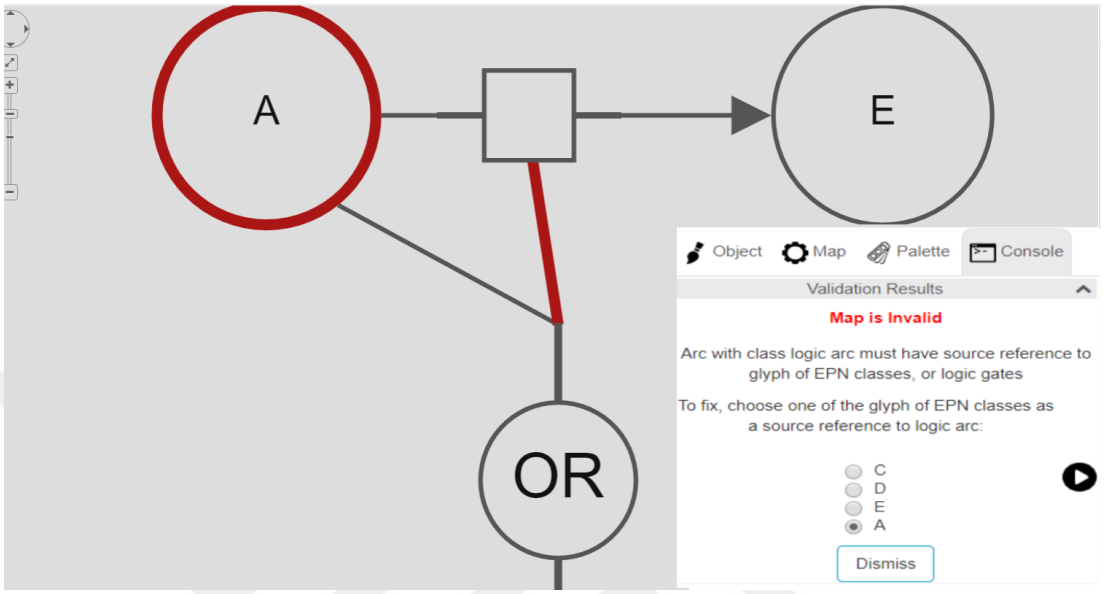

Figure 4.21: An example where rule pd10124 is violated

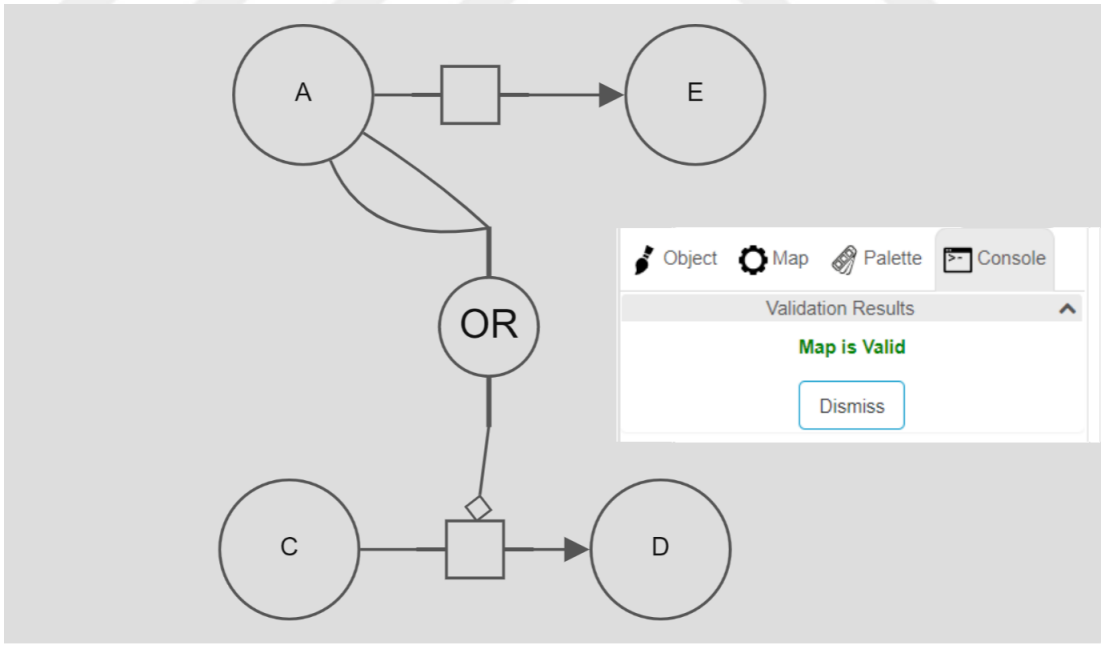

Figure 4.22: After the rule pd10124 violation is fixed

Validation problem with pd10110 rule occurs when a modulation arc does not have a target node with PN classes. To fix such an error, a change of the target node is offered. In order to achieve this, the user is forced to choose a nearby

node with a PN class. Hence, nearby nodes with PN classes are presented to the user as a radio button group with the nodes' type names as seen in Figure 4.23. In fixing the problem, the target of the problematic arc needs to be altered by Cytoscape.js library's `move()` function and `porttarget` attribute of the arc needs to be set accordingly as well (Figure 4.24).

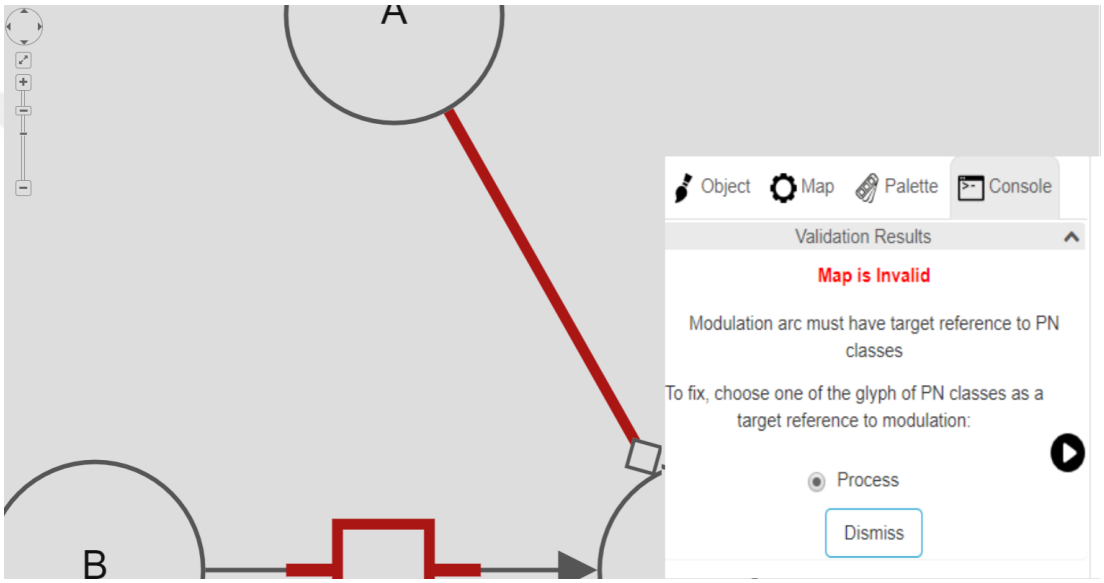

Figure 4.23: An example where rule pd10110 is violated

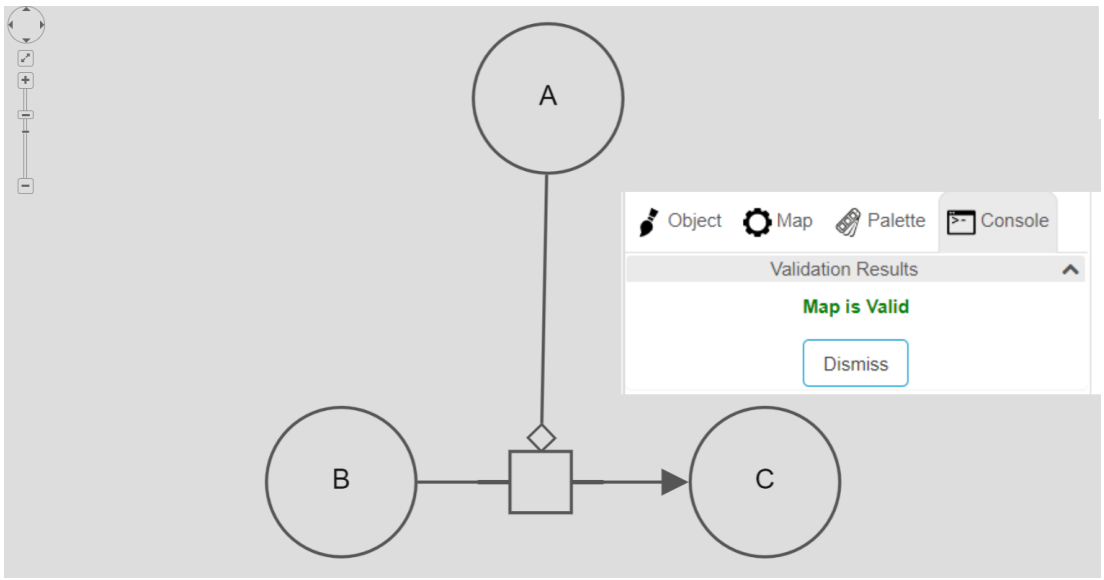

Figure 4.24: After the rule pd10110 violation is fixed

Problem in pd10111 occurs when a logic glyph is the source of more than one arc. For correcting a failure with this rule, we offer the user removing all arcs whose source is a logic gate except exactly one. In a radio button group, these edges are represented with a description which includes the logical gate type and the type of target nodes with PN classes as exemplified in Figure 4.26. These arcs are highlighted on the graph as well. In accomplishing of correcting the problem, the chosen edge by the user remains and all other connected edges whose source is the erroneous logical gate are removed with the help of Cytoscape.js library's `remove()` function as exemplified in Figure 4.26.

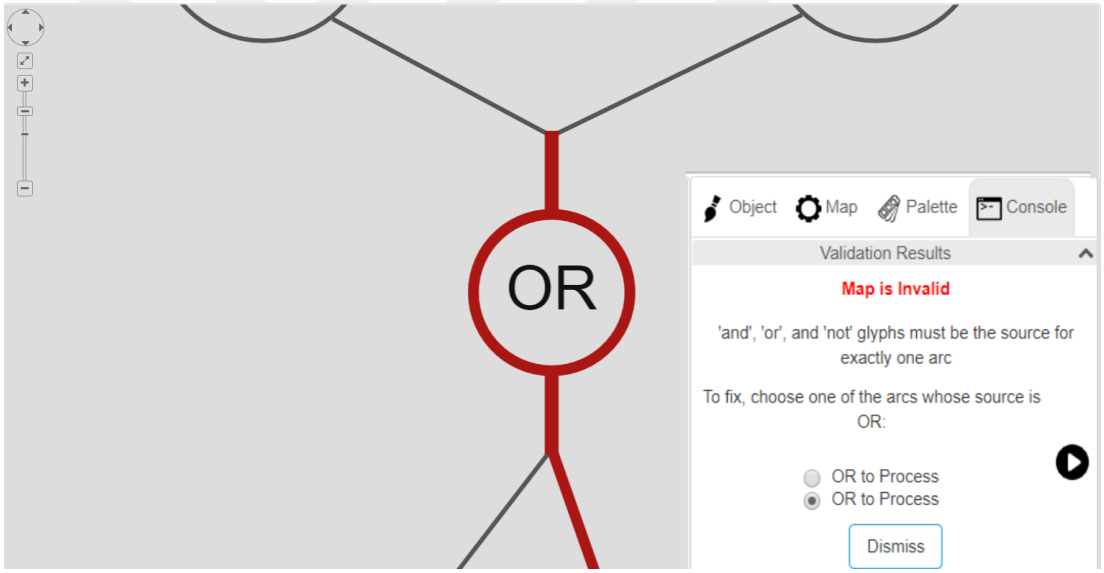

Figure 4.25: An example where rule pd10111 is violated

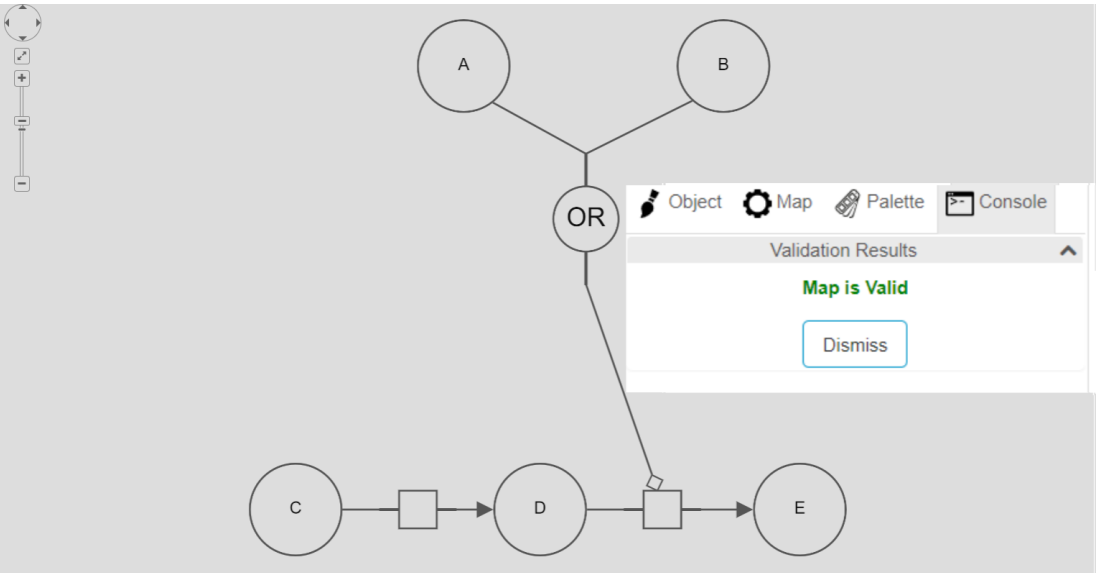

Figure 4.26: After the rule pd10111 violation is fixed

Yet another failure with code pd10112 occurs when a top-level nodes does not have a compartment reference even though there does exist compartments (i.e., cellular locations) defined in the map. While demonstrating this failure, we display a radio button group on the Console tab to list choices to the user for a compartment for each top-level node as exemplified in Figure 4.27. These compartments are presented to the user with compartment's label name. When the user chooses to fix the problem, the specified top level node is placed inside the selected compartment by using Cytoscape.js library's `move()` function, setting the parent attribute (Figure 4.28).

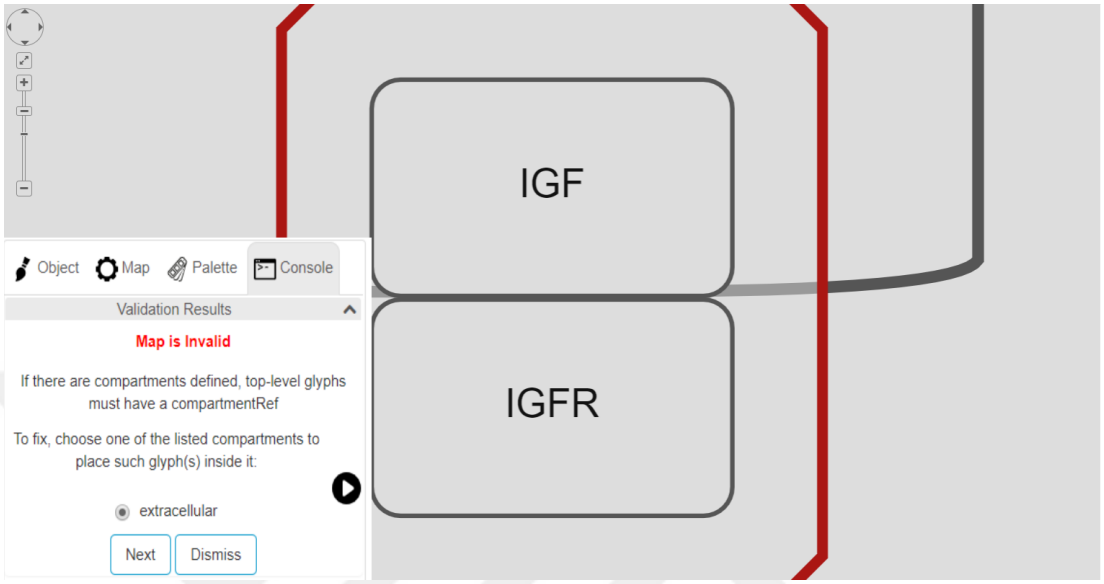

Figure 4.27: An example where rule pd10112 is violated

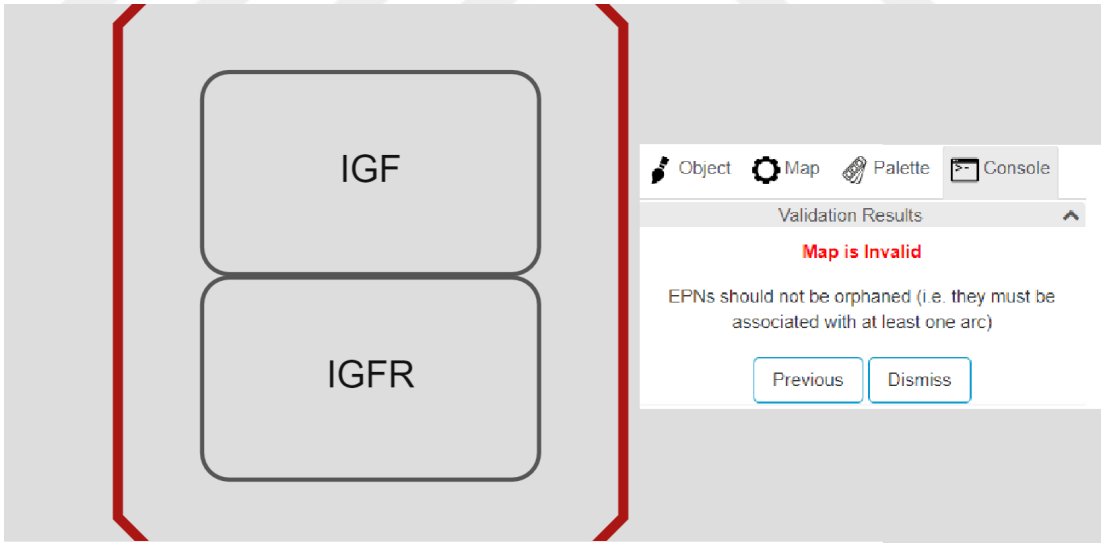

Figure 4.28: After the rule pd10112 violation is fixed

Rule pd10125 fails when a logic arc does not have a target node which is a logical operator. For correcting such a failure, we suggest that one of any existing logical operators becomes the target node of erroneous edge. Hence, we list all nearby logical operators on the Console tab in a radio button group as exemplified in Figure 4.29. On the Console, these logical operators are represented with their

type names and also highlighted to eliminate any ambiguity. Upon invocation of the fix, the erroneous edge is removed and a new edge is added by using the `addEdge()` function in Newt as can be seen in Figure 4.30.

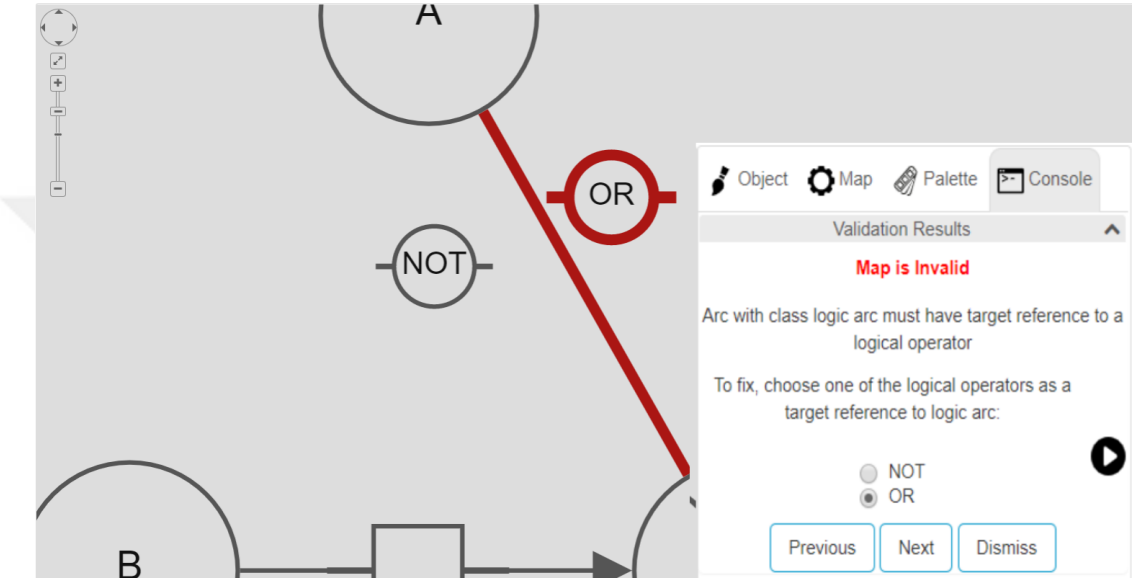

Figure 4.29: An example where rule pd10125 is violated

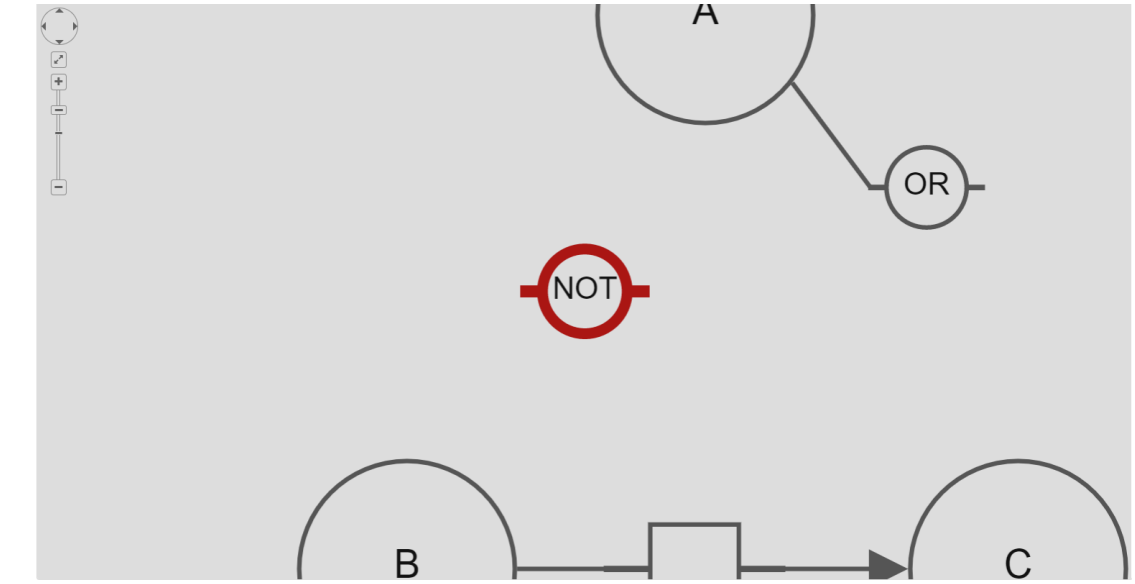

Figure 4.30: After the rule pd10125 violation is fixed

To fix the problem in rule pd10126, choosing a logic arc that will be the only

logic arc connected to the associated logical operator is suggested since validation failure in this rule is that "not" logical operator is the target node of more than one logic arc. Hence, the user is imposed to choose one of the existing logic arcs to fix problem. Logic edges are represented with radio buttons with a description containing the node's label name and the logical operator's type as exemplified in Figure 4.31. These edges are also highlighted on the map. In fixing this error, the chosen edge remains and all other edges whose target node is the erroneous "not" operator are removed by utilizing Cytoscape.js library's `remove()` function as shown in Figure 4.32.

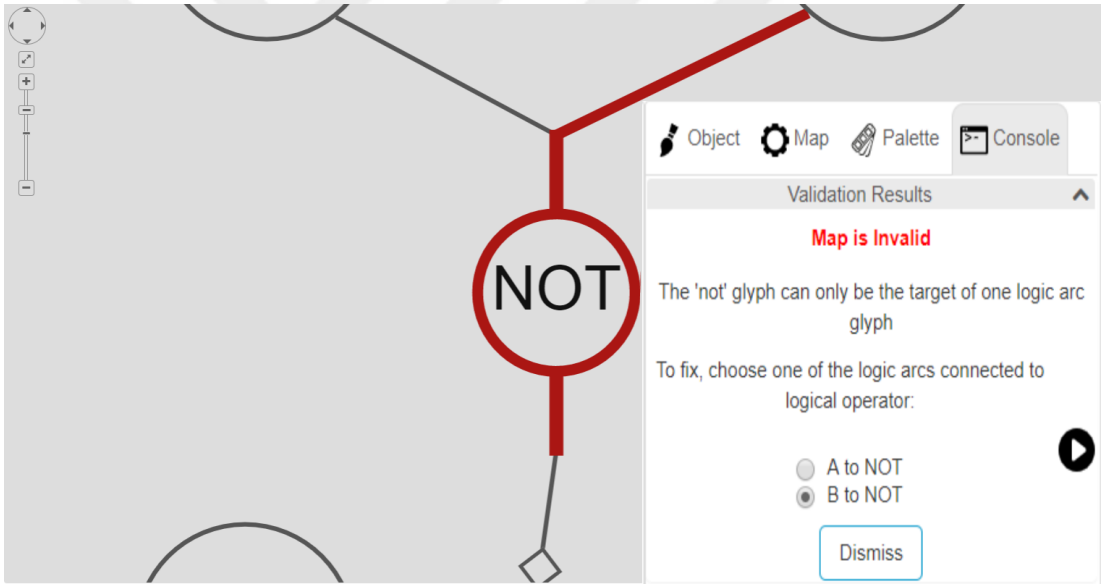

Figure 4.31: An example where rule pd10126 is violated

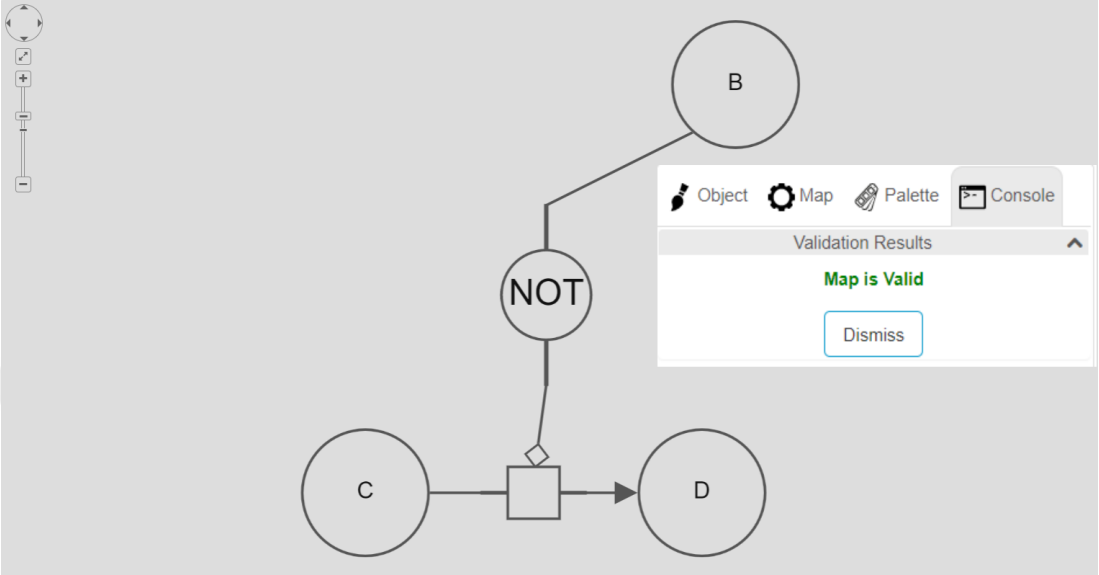

Figure 4.32: After the rule pd10126 violation is fixed

Rule pd10128 fails when an arc with class equivalence arc does not have a target node with class "tag" or "submap" or "terminal". Thus, changing the target node is needed to fix the error. For this, the user is demanded to choose a nearby node which is a tag or a submap. Hence, nearby nodes which are of class tag or submap are presented as radio buttons with a label consisting of the node's type name as exemplified in Figure 4.33. Upon executing the fix, the target of the problematic edge is replaced with an instance of the chosen node's type via Cytoscape.js library's `move()` function and the `porttarget` attribute of the edge is set accordingly (Figure 4.34).

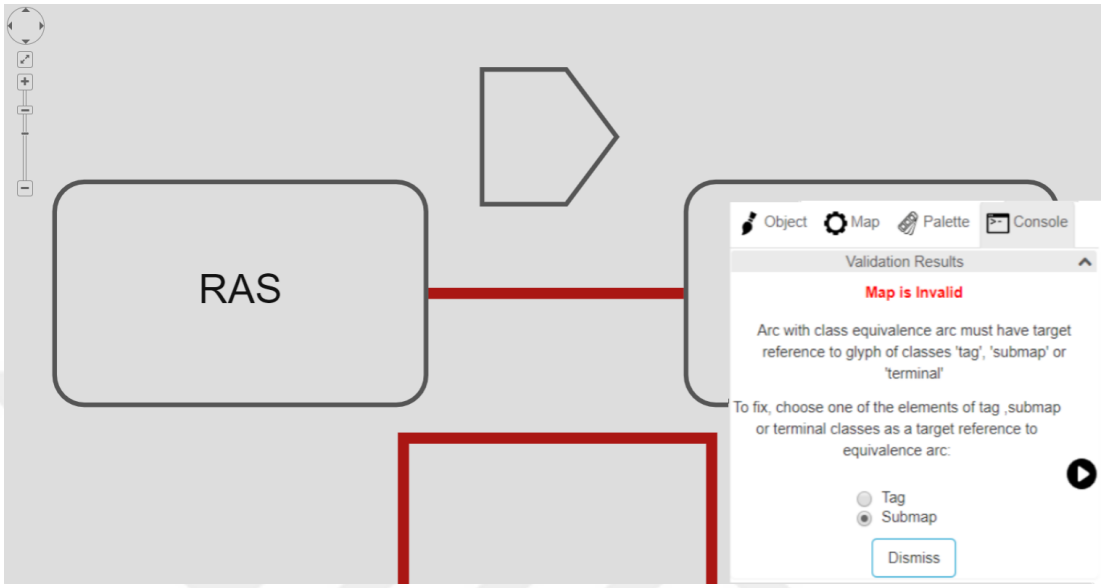

Figure 4.33: An example where rule pd10128 is violated

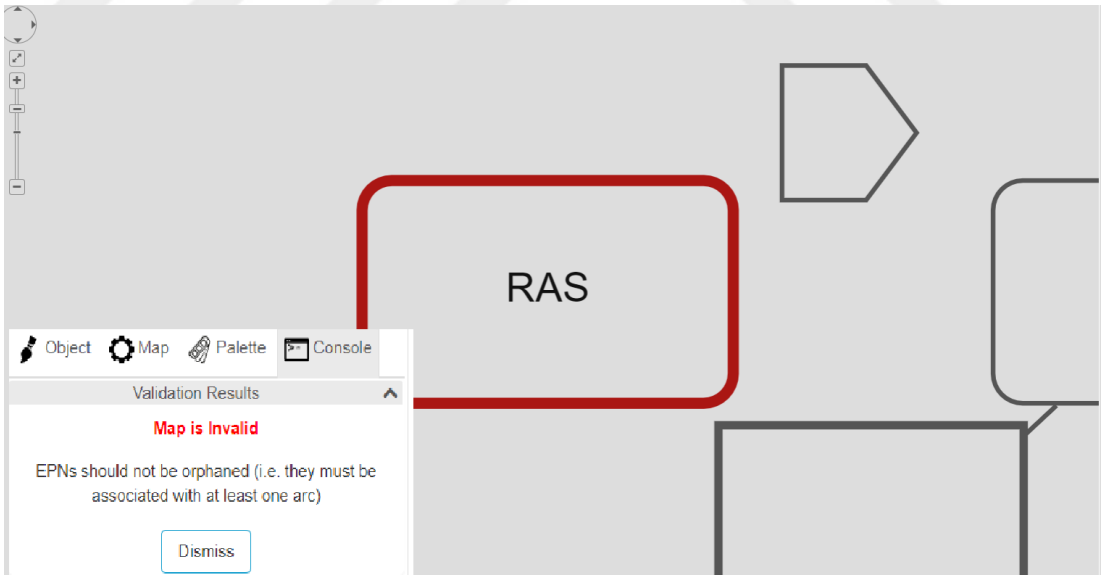

Figure 4.34: After the rule pd10128 violation is fixed

Finally, rule pd10142 will fail when a logic arc is not connected to any logical operator. In representing of this kind of error, a radio button group is displayed on the Console tab for choosing the new edge's type to be placed instead of erroneous logic arc. Options for radio buttons are production and consumption

edge types. These edge types are demonstrated with a description containing the edge type name as seen in Figure 4.35. In handling of this error's fix action, the logic arc is removed by utilizing Cytoscape.js library's `remove()` function and a new edge is added with chosen type by using the `addEdge()` function in Newt as exemplified in Figure 4.36. In this case, it is assured that if a production edge is added, it is connected to the output port of the associated process. Similarly, when a consumption edge is added, this edge needs to be connected to the input port of the associated process.

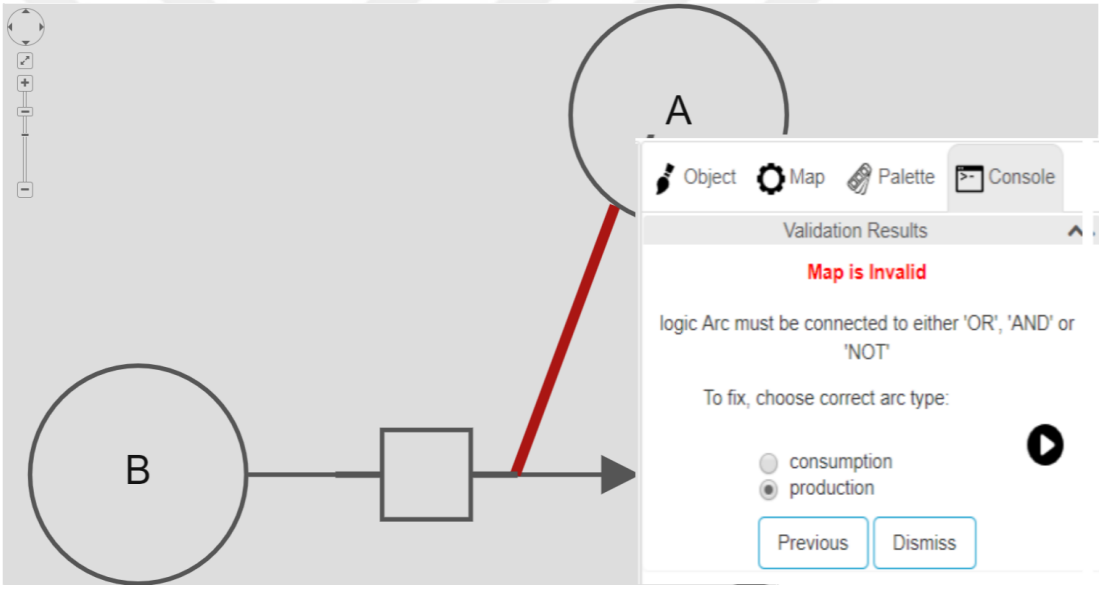

Figure 4.35: An example where rule pd10142 is violated

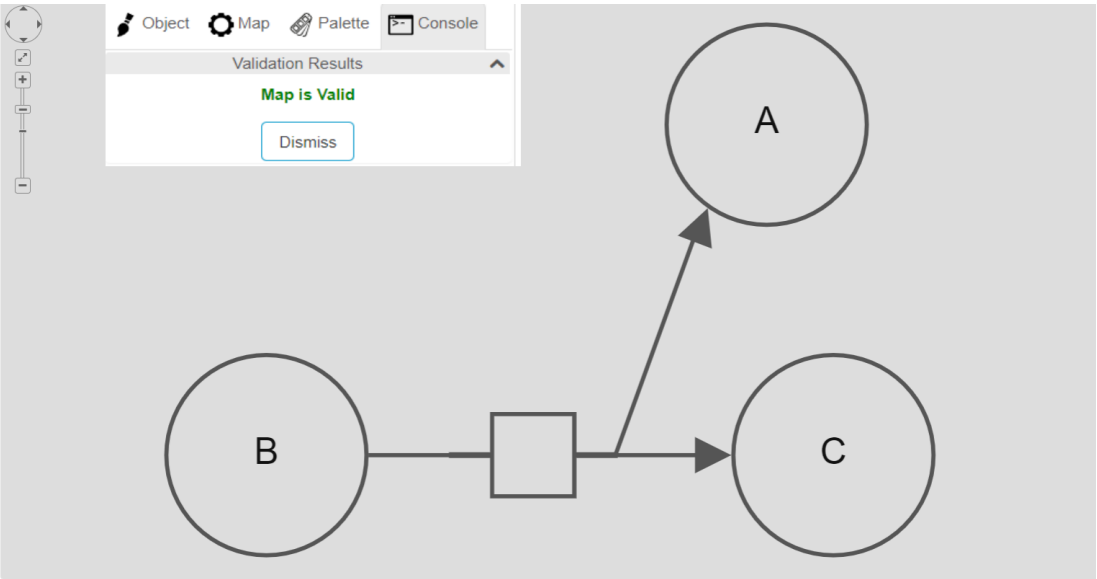

Figure 4.36: After the rule pd10142 violation is fixed

4.2.4 Visual Arrangements for Effective Representing of Validation Control

In order to represent validation control impressively in the Newt, we make several visual arrangements. One of them is that we zoom in erroneous part of graph after semantic validation action is taken by utilizing Cytoscape.js library's animate() function. In the Newt, user can also zoom in all graph by using specific functionality in the Newt. If user defined zoom level is higher than defined zoom level in validation rendering, we use user defined zoom level in validation rendering. Furthermore, one other arrangement in zooming is that if erroneous part of graph is not visible on the screen after zooming, we make visible that part on the screen by using Cytoscape.js library's fit () function.(4.37)

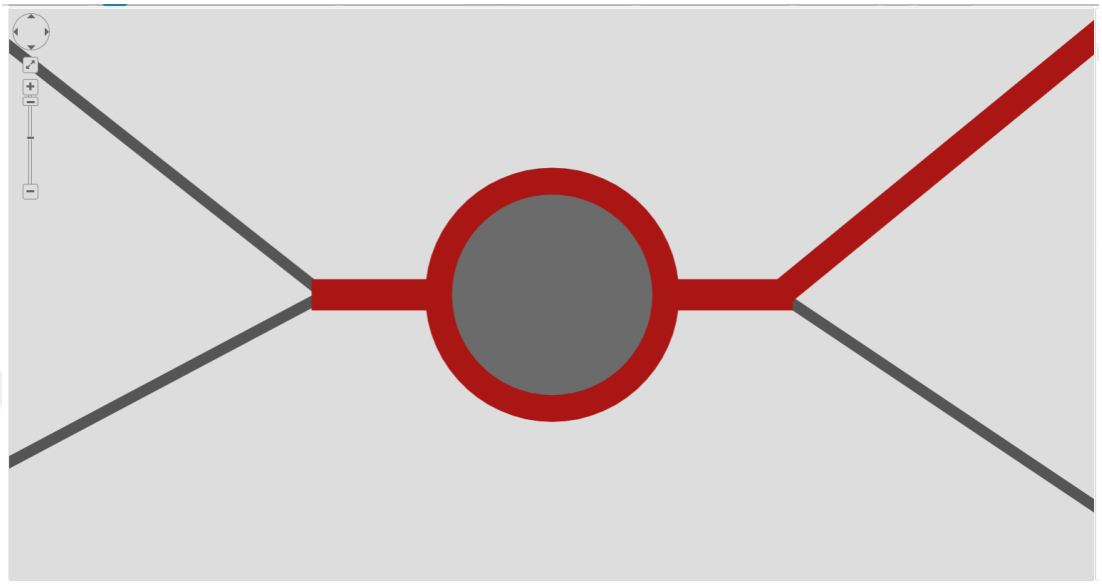

Figure 4.37: Zooming Erroneous Part in Graph

Another visual arrangement for representing validation control is made after user press dismiss button on the console tab. In this adjustment, all graph is centered and padding is applied to graph. Graph returns the position on the screen before validation action is taken.

Finally, when fix action is applied to graph, graph is validated again by applying validation action. Then, if graph is valid, same adjustment in dismiss button action is applied to the graph. Otherwise, we zoom in remaining erroneous part of graph again.

4.3 Functional Changes in Newt

For the purpose of adding validation check to Newt, some new functionality had to be added. One such method is to call realated functionality in SBGNViz.js library to obtain the validation result. For this, different versions of SBGNViz.js library and libSBGN.js library had to be integrated.

In addition, a Console tab on the right panel was dynamically added and removed as needed. Validation messages are displayed on this tab as well as error fix suggestions. The Console tab was created in simple HTML table format with validation messages and the fix button. Radio button group was used for providing different fix choices for failures in some rules.

Furthermore, Cytoscape.js library's highlighting functionality was applied with color red for highlighting erroneous part(s) of the graph. To turn user's focus to erroneous graph elements, again this library's `animate()` and `fit()` functions were used to zoom and pan in an animated fashion.

Lastly, Cytoscape.js topology related functions were used in handling of the fix. For instance, edge's `move()` function in this library was used for making changes on source or target node. Again `portsource` and `porttarget` attributes were set for edges for fixing certain errors. `remove()` function was a popular function to fix certain types of errors as well.

Chapter 5

Conclusion

With this thesis, we added semantic validation to the libSBGN.js library by implementing a client side XSLT processor and utilizing a DOM Parser. We also provided efficient validation report handling with proper object design. The SBGNViz.js library, which Newt is built on, was extended to use the libSBGN.js library for semantic validation of SBGN maps as well as for simple read and write operations on SBGNML files.

Newt's user interface was modified for visually communicating the validation results upon the use of the modified libSBGN.js library for the current SBGN map. To achieve this, we added a new Console tab on the right panel. This tab is rendered only upon invocation of validation. A message is shown to indicate whether or not the validated map is valid. If not valid, a validation error description message is shown. In addition for most rules used in validation, error fix suggestions are provided. When multiple choices are available for a fix, options are expressed as a radio button group. Finally, an execute button is used to apply the suggested fix.

To aid users in understanding the error, erroneous part(s) of the map are highlighted in red. In fact, we not only show the problematic part(s) of the map but also how a particular suggested choice fixes the problem by highlighting the

appropriate part of the map. To bring focus on such parts of the map, we also adjust the pan and zoom levels in an animated fashion.

Suggested fixes are meant to change the topology of the SBGN map by removing / adding map elements or *rewiring* existing connections. Fixed map is validated again to determine whether or not other errors remain. Sometimes, a fix will result in a new error, whereas in other times, the map will become valid.

5.1 Future Work

At the libSBGN.js library level, we completed all planned work so that semantic validation can be added to SBGN tools such as Newt in a straightforward manner. The problems in a validated SBGN map can now be effectively and visually communicated to the user through Newt's enhanced user interface. In addition, for most of the rules defined in libSBGN.js library, we are able to offer fix suggestions.

Main reason we cannot offer a fix for certain rules or we cannot suggest a *better* fix for certain rules is the fact that we do not have the opportunity to compare the local SBGN map with those stored and integrated in a central database.

In the future, we hope to integrate an SBGN database to Newt, similar to that done in this thesis work [20], so that semantic validation can check for patterns available in pre-stored and integrated known pathways to come up with more *intelligent* suggestions for a fix.

Bibliography

[1] A. Rougny, V. Touré, S. Moodie, I. Balaur (Roznovat), T. Czauderna, H. Borlinghaus, U. Dogrusoz, A. Mazein, A. Dräger, M. L. Blinov, A. Villéger, R. Haw, E. Demir, H. Mi, A. Sorokin, F. Schreiber, and A. Luna, "Systems biology graphical notation: Process description language level 1 version 2.0," *Journal of Integrative Bioinformatics*, vol. 16, 06 2019.

[2] M. P. Van Iersel, A. C. Villéger, T. Czauderna, S. E. Boyd, F. T. Bergmann, A. Luna, E. Demir, A. Sorokin, U. Dogrusoz, Y. Matsuoka, *et al.*, "Software support for sbgn maps: SBGN-ML and LibSBGN," *Bioinformatics*, vol. 28, no. 15, pp. 2016–2021, 2012.

[3] "Newt: Pathways simplified." http://newteditor.org/. Accessed: 2019-06-26.

[4] "Graph visualization: Why it matters." https://linkurio.us/blog/why-graph-visualization-matter/. Accessed: 2019-06-26.

[5] W. Cui, "A survey on graph visualization," Master's thesis, Hong Kong University of Science and Technology, Japan, 2008. Supervisor: Qu, Huamin.

[6] N. Le Novere, M. Hucka, H. Mi, S. Moodie, F. Schreiber, A. Sorokin, E. Demir, K. Wegner, M. I. Aladjem, S. M. Wimalaratne, *et al.*, "The systems biology graphical notation," *Nature biotechnology*, vol. 27, no. 8, p. 735, 2009.

[7] "libSBGN library." https://github.com/sbgn/sbgn/wiki/LibSBGN/. Accessed: 2019-06-22.

[8] "Schematron: validating XML using XSLT." http://www.ldodds.com/papers/schematron$_x$*sltuk.html*. *Accessed* : $2019 - 06 - 26$.

[9] "The extensible stylesheet language family (XSL)." https://www.w3.org/Style/XSL/. Accessed: 2019-06-26.

[10] S. Moodie, N. Le Novere, E. Demir, H. Mi, and A. Villeger, "Systems biology graphical notation: process description language level 1 version 1.3," *Journal of Integrative Bioinformatics*, vol. 12, no. 2, pp. 213–280, 2015.

[11] D. L. Wendell Piez, "Introduction to schematron." http://www.mulberrytech.com/papers/schematron-Philly.pdf, 2008. Accessed: 2019-06-23.

[12] "XPath Tutorial." https://www.tutorialspoint.com/xpath/. Accessed: 2019-06-23.

[13] D. Olteanu, "XSLT 1.0 tutorial." https://www.cs.ox.ac.uk/dan.olteanu/tutorials/xslt1.pdf/. Accessed: 2019-06-24.

[14] "Chapter 2. XSL processors." http://www.sagehill.net/docbookxsl/XSLprocessors.htmlXSLTprocessors/. Accessed: 2019-06-24.

[15] "XSLTProcessor." https://developer.mozilla.org/en-US/docs/Web/API/XSLTProcessor/. Accessed: 2019-06-24.

[16] "Cytoscape.js: Graph theory (network) library for visualisation and analysis." http://js.cytoscape.org/. Accessed: 2019-06-26.

[17] E. G Cerami, B. Gross, E. Demir, I. Rodchenkov, O. Babur, N. Anwar, N. Schultz, G. D Bader, and C. Sander, "Pathway Commons, a web resource for biological pathway data," *Nucleic Acids Research*, vol. 39, pp. D685–90, 11 2010.

[18] "Pathway Commons: A resource for biological pathway analysis." http://www.pathwaycommons.org/. Accessed: 2019-08-19.

[19] M. Sari, I. Bahceci, U. Dogrusoz, S. Sumer, B. Aksoy, Babur, and E. Demir, "SBGNViz: A tool for visualization and complexity management of SBGN process description maps," *PLoS ONE*, vol. 10, p. e0128985, June 2015.

[20] M. E. Karaca, "Efficient Querying of SBGN Maps Stored in a Graph Database," Master's thesis, İhsan Doğramacı Bilkent University, Turkey, 2019. Supervisor: Uğur Doğrusöz.