Geometric and Computational Aspects of Manipulation Rules for Graph-Based Engineering Diagrams

Johannes Bayer*, Yakun Li*, Sebastian Marquardt‡‡, Andreas Dengel*

*Smart Data and Services
DFKI, Kaiserslautern, Germany
E-mail: {johannes.bayer, yakun.li, andreas.dengel}@dfki.de

‡‡Bilfinger Digital Next, Heidelberg, Germany
E-mail: sebastian.marquardt@bilfinger.com

***Abstract*—The digitization of graph-based engineering diagrams like P&IDs or circuit drawings from optical sources as well as their subsequent processing involves both image understanding and semantic technologies. More precisely, after a raw graph has been obtained by an object detection and line extraction pipeline, semantic gaps (like resolving material flow directions) need to be overcome to retain a comprehensive, semantically correct graph. Likewise, the graph representation often needs to be altered to achieve interoperability with established CAE systems and to accommodate customer-specific requirements. Semantic technologies provide powerful tools to manipulate such data but usually require rather complicated implementation. Graphically presentable graph based rules provide a code-free mean to ease the interaction with domain experts. In order to be applicable in real-world applications, both geometric and computational aspects need to be considered. This paper explores these aspects and demonstrates use cases of such rule graphs.**

***Index Terms*—Piping and Instrumentation Diagram, P&ID, Rule Graph**

I. Introduction

The comprehensive digitization of interconnected engineering constructs like chemical plants [8] [9] [10] or electrical devices [11] [12] [13] requires a homogeneous data representation of all relevant describing documents. Piping and instrumentation diagrams (P&IDs) and circuit diagrams are such types of graph-based, symbolic documents. Often, optical media like plain paper or crude vector graphics are the only basis for the digitization. In order to be usable with like computer-aided engineering (CAE) or asset management software systems, information extraction is required.

This can be accomplished by a image processing, i.e. a graph extraction based on existing object and line detection algorithms, where electric and instrumentation symbols become nodes while their interconnecting lines become edges. However, the geometry of a layout does not necessarily denote the complete graph structure in a unique way. Likewise, structurally varying graph representations may be required by different target systems. So in order to obtain a homogeneous, semantic graph that fully represents the components and interconnections of the plant, subsequent graph processing is required.

LEVIA'22: Leipzig Symposium on Visualization in Applications 2022

II. Industrial Motivation

Readers of Piping and Instrumentation Diagrams use the whole information displayed on the P&ID to understand a specific area on the P&ID. An engineer is also capable of understanding the meaning of an area on a P&ID even if the specific information he is looking for is not displayed on the P&ID. The way he does that is by understanding the surroundings of the area of interest and deriving the context from this area.

The system a P&ID was initially created in might also have a different data model compared to the system is is being maintained in (typically a CAE system). Since both systems display the same visual information the need arises to shift the representation of the P&ID Graph from one system so that it gets accepted by another system from the data-perspective. This requires a manipulation of the underlying graph structure so that both systems are able to work with the data from the P&ID. For example, a valve in a CAE is a standalone entity whereas a valve often times must be part of a pipe section of today's CAE systems. Visually they look the same, but the way the graph must provide the data is different. This brings need to graph manipulation so both systems can work with the same meaning of data but represented in a different format.

The initial graph after the recognition in fig. 1 displays three scattered lines (yellow) but the final outcome must be one pipeline as this is what the graph is required to look like. This additional postprocessing need is what is required to fulfill the business case for PIDGraphs. For this purpose the rule graph applicator (RGA) was developed.

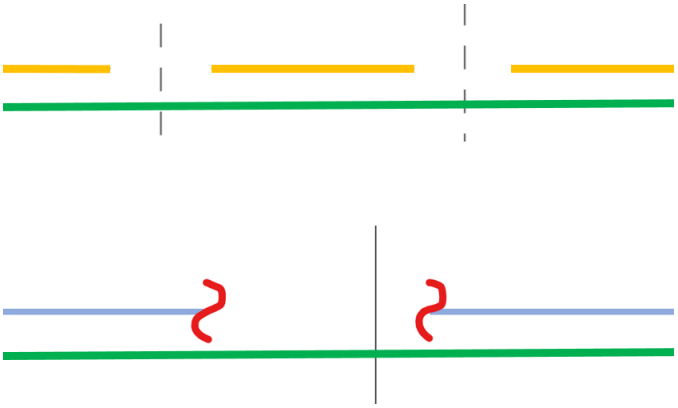


Fig. 1: Sample Application of the Proposed Graph Manipulation. Scattered line segment edges (yellow) as well as edges disrupted by dedicated break symbols (red) are joined (green) to obtain a semantically correct graph representation.

III. Related Work

Graph-based manipulation rules have been proposed for graph representations of algebraic systems [4]. Likewise, graph-based graphical user interfaces exist for code-free software development [7].

Rule-based modifications for engineering graphs like P&IDs have already been investigated [2]. However, based on the dedicated query language of a graph database, this approach required the users (process engineers) to write queries in a rather abstract manner.

IV. Subgraph Isomorphism

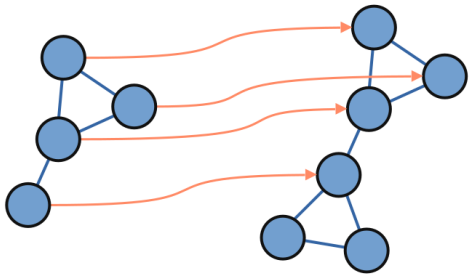


Fig. 2: Two Isomorphic Graphs.

Both P&IDs as well as the rules for introducing local alterations within them can be described as graph structures. Therefore, the task of finding appropriate locations for such rule applications equals the subgraph isomorphism problem [14]. More precisely, for graph structures $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$, a subgraph isomorphism $f_I : N_1 \to N_2$ injectively maps all nodes of the first graph to the nodes of the second graph while requiring the presence of all interconnections between the nodes of the first graph to also exist within the second graph (see fig. 2):

$$(n_1, n_2) \in E_1 \implies (f_I(n_1), f_I(n_2)) \in E_2 \qquad (1)$$

V. Rule Graphs

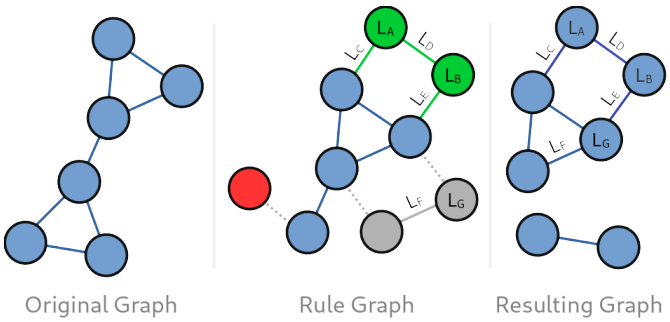


Fig. 3: A *rule graph* transforms an *original graph* into a *resulting graph*. Insertions are marked green, (label) substitutions are marked grey and deletions are marked red. The dashed grey lines indicate transition edges from isomorphic nodes to their manipulative counterparts. Unlabeled substitution nodes are auxiliary structures only. Within the engineering domain, the label of a node or edge may equal to a type or property of the respective component.

Rule Graphs [1] have been proposed as an intuitive concept for engineering graph manipulation (see fig. 3). They are a variety of triple graph grammars [3].

This paper outlines implementation issues during the application of Rule Graphs [1] to real-world P&IDs. Since rule graphs attempt to provide an approachable way to model modifications, both graph and geometric aspects need to be handled.

VI. Rule Design Considerations

In order to describe symbolic engineering drawings in a strict graph-based manner, *all* contained information need to be modeled either as node, edge or graph labels.

As nodes and edges of the application graphs bear complex data structures as labels (e.g. domain-specific properties, positioning, symbol type, connectors), some of these properties directly determine the graphical appearance of the application graph. Rule graphs need to be designed to resemble these aspects for matching and manipulation purposes. In fact, the only difference between rule and application graph design is the need for encoding the role of a node inside the rule. Two basic approaches to model the role of node inside a rule are considered:

- Role as additional Node property. Allows reusing the rest of the graph implementation. Not suitable for all roles (e.g. deletion nodes).
- Role as dedicated node types. Requires key aspects like symbol types to be modeled differently from application graphs. Dedicated design allows for avoiding ambiguities.

As avoiding ambiguities was a primary objective to ensure well-defined rule application semantics, the second scheme is considered in this paper. Dedicated nodes could be introduced to the rule graphs to model symbol connectors. But as they

increases the rule graph complexity and to achieve a "one-to-one mapping" regime, they were also discarded in the descried implementation.

VII. Methodology

In order to narrow down the amount of allowed subgraph isomorphisms and to specify graph manipulations, node and edge labels are introduced into the rule graph along with execution mechanics.

A. Rule Graph Structure Constraints

Apart from the constraints for rule graph construction defined in [1], other restrictions need to be met in order for the execution to be sound:

- Every substitution node must be connected to at most one original node
- Every deletion node must be connected to at most one original node
- Insertion edges must only connect insertion nodes and original nodes
- Insertion nodes must only be connected via insertion edges
- Deletion edges must only connect original nodes and substitution nodes

B. Value Handling

Node and edge labels of rule graphs can have static and variable values. Variables can be defined in the isomorphic part of the rule graph and recalled during manipulations. For the matching of node and edge types as well as connectors, regular expressions [15] are used.

C. Single Rule Application

Given a subgraph isomorphism between original graph and the isomorphisc part of the rule graph (a subgraph of the rule graph consisting of its original nodes and original edges), a single rule application is performed as a sequence of modifications. Each modification is based on a rule graph item and done in the following order:

1) Insert Nodes
2) Insert Edges
3) Substitute Nodes
4) Substitute Edges
5) Delete Edges
6) Delete Nodes

Alternatively, walking through the rule graph for its application could help to reduce computational overhead. However, this approach is considered to be more error-prone.

D. Multiple Rule Application

Generally, a rule can be applied in different ways to an original graph. Usually, the same rule should be applied to distinct graph parts. Therefore, strategies are needed for a favorable outcome:

1) Single Isomorphism: Given the list of subgraph isomorphisms between the isomorphic rule part and the original graph, one of them is used. Since the subgraph isomorphism calculation is deterministic, a (user-) predictable and understandable behavior can be achieved.

2) Disjoint Isomorphisms: Given the list of all subgraph isomorphisms, a subset of isomorphisms is chosen so that their *sets of departure* are pairwise disjoint. If this subset equals the overall set of isomorphisms, the result there is one one possible resulting graph. The (very costly) calculation of subgraph isomorphisms only needs to be done once. Since the list is finite, the algorithm terminates in finite time. In many situations, this is preferred option.

3) Iterative Isomorphisms: Apply a rule until there the list of subgraph isomorphisms is empty. Very costly due to multiple isomorphism calculations, not guaranteed to finish (a rule that adds a single node may be considered) and the result is not guaranteed to be unique.

E. Connector Handling

Connectors are considered part of node labeling that can be referenced in the edges. When inserting edges between new nodes or when matching edges between nodes from the original graph, node connectors can be conveniently and explicitly specified by referring to a fixed connector id. For edges between nodes of the existing graph and newly added nodes, other techniques are required.

Connectors could be encoded in rules by dedicated connector nodes. However, this would make the rule design more complex and would require to transform the graph structure to match the original one.

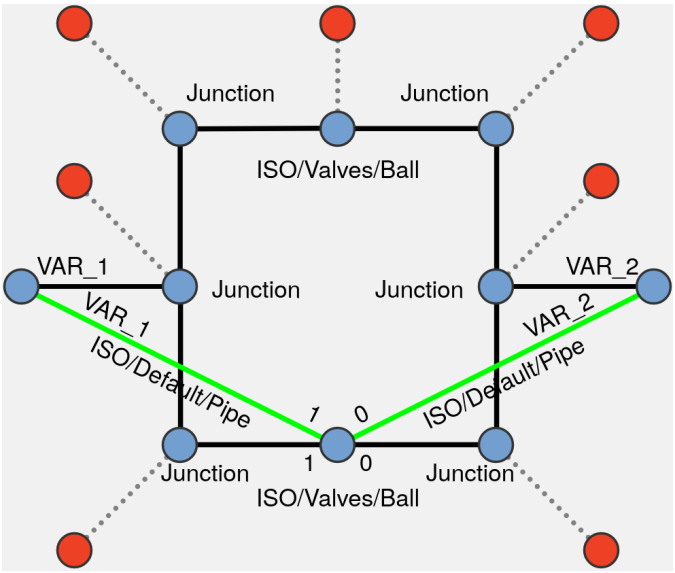
As a compromise, variables are introduced, that are read during the matching process with the original graph and are used for the generation of inserted and altered edges (see fig. 4).

F. Geometric Constraining

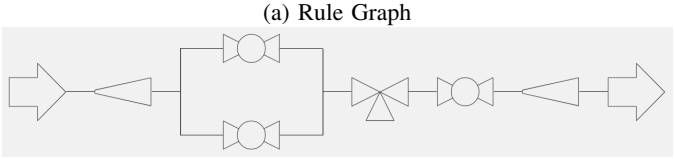
So far, only subgraph isomorphisms based on the original graph structure have been considered as a basis for the rule application. In many situations however, geometric aspects need to be regarded. For example, symbols also needs to be matched based on their local proximity. Geometric constraints for edge line length and angles are introduced in order to narrow down the set of subgraph isomorphisms between rule graph and original graph (see fig. 5). In order to support constraining the geometry between arbitrary (unconnected) nodes, dedicated edges are introduced in the rule graphs (dashed lines).

G. Inserted Node Positioning

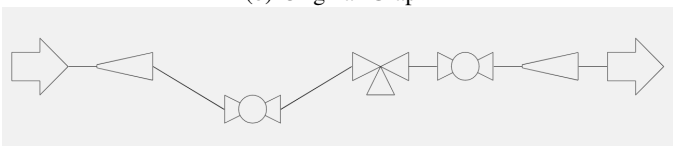
In order to keep the rule graph design approachable, the geometry of insertion subgraphs needs to be preserved when calculating the positions of inserted nodes.



(a) Rule Graph

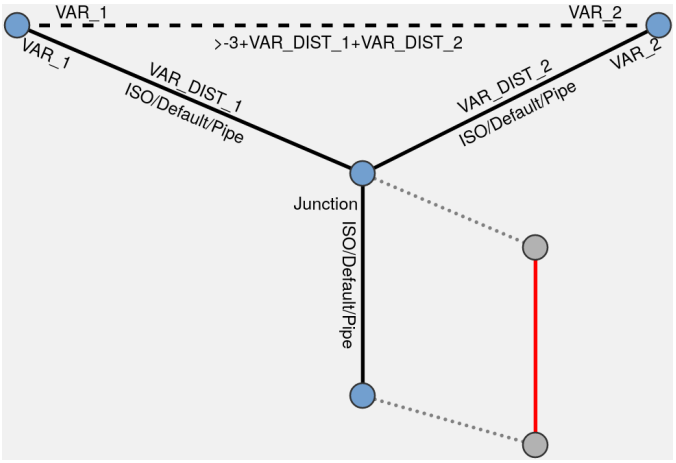


(b) Original Graph

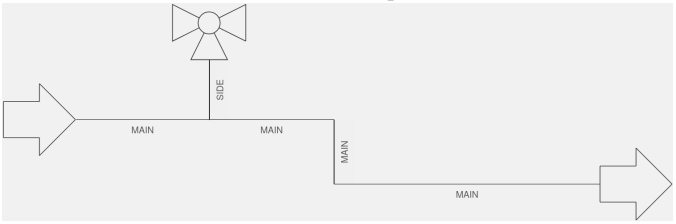


(c) Resulting Graph

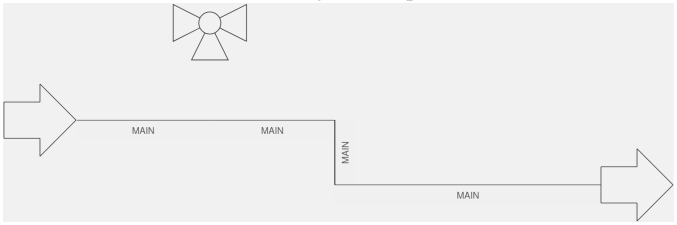
Fig. 4: Sample Application of a Rule for simplifying a parallel structure. Node and edge Type indicators placed below each item when possible. Connector variables are prefixed "VAR_".



(a) Rule Graph



(b) Original Graph



(c) Resulting Graph

Fig. 5: Geometric constraining narrows down the set of subgraph isomorphisms before application. This rule utilizes the triangle inequality to delete the branching edge and to keep the edges that form a continuous line.

H. Node Substitution

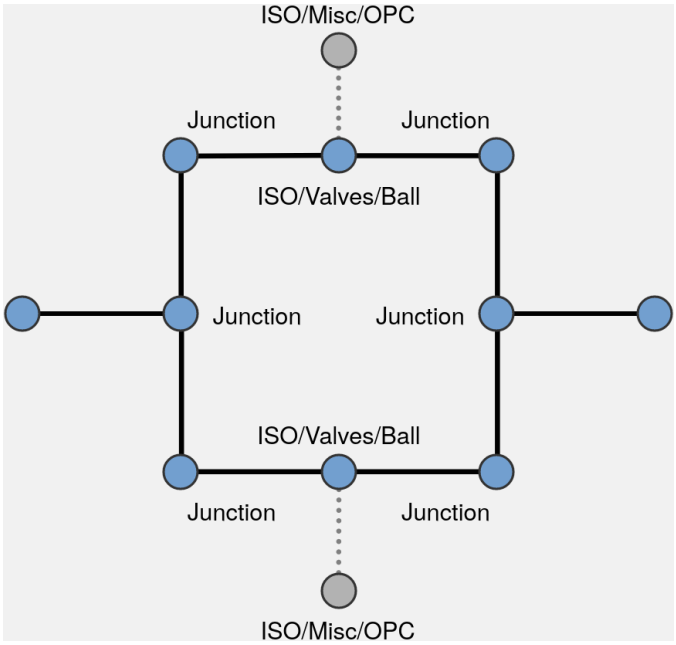
Details about how to define a Node Substitution Rule (NSR) could be seen from [1]. Nodes in the application graphs could contain different *connectors* and *properties* (described more detailed below) and they need to be handled carefully while applying the NSR. For example, the NSR defined in figure 6a could substitute parallel nodes of type *ISO/Valves/Ball* to type *ISO/Misc/OPC*. This NSR could be defined to properly produce the visual effect shown in figure 6d after applying to original graph in figure 6b.

1) Node Type Substitution Only: It is feasible to only substitute the value of *nodeType* attribute of original node in application graph from *ISO/Valves/Ball* to *ISO/Misc/OPC* in the given example and one result is shown in figure 6c. The red dots shown in the figure are the untouched *connectors* defined in the original node. The *connectors* defined the connection points of the node where the connected edges should connect to.

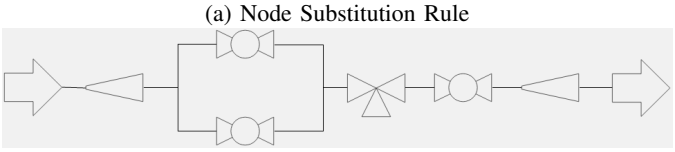
2) Connectors Mapping: It could be seen from section VII-H1 that substituting connectors is also necessary to produce visually pleasing and correct result. Therefore, there is feasibility provided to define attributes in the substitution node (e.g., grey colored nodes in figure 6a) in the rule graph to map connectors specifically. For example, it is feasible to define {*connector_1: 0*} in the substitution node to map the *connector* with id 1 of node in original graph to a new connector id 0 which exists in nodes of type *ISO/Misc/OPC*.

It is time consuming if there are many connectors to map in substitution nodes of the rule graph. Therefore, an automatic mapping method has been proposed to map the connectors in the original node (`orgConns`) to the connectors of new node type (`newConns`) as shown in algorithm 1. After getting the result mapping, the old connector id is updated in the connected edges with the new one and `-1` indicates a fallback connecting to the center of node.

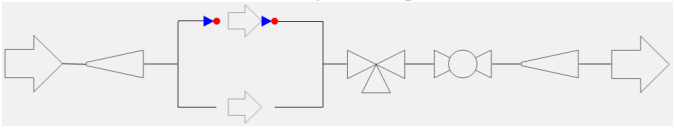
Additionally, after adapting the width and height of the original node based on the new type, then the expected visual



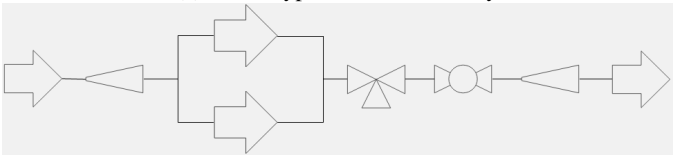
(a) Node Substitution Rule



(b) Original Graph



(c) Node Type Substitution Only



(d) After Proper Node Substitution

Fig. 6: Node substitution example.

result shown in figure 6d could be achieved.

3) Properties Substitution: Even though the expected visual result could be achieved by following the above steps for node substitution, the nodes of type *ISO/Valves/Ball* contain *properties* lists different from nodes of type *ISO/Misc/OPC*. The *properties* list contains defined *Property* and each *Property* in the list has different value. Therefore, *properties* adaption is also necessary to keep the semantic meaning of the node and furthermore the credibility of the application graph.

The following three options are provided to allow user chooses how they would like to handle *properties* substitution for node in NSR.

- The first option is to completely replace all existing *properties* by the *properties* of the new node type.
- The second option allows user to keep all existing *properties* and add *Property* from new node type if it is not already in existing properties.
- The third option allows user to keep existing *Property* if they are not in the *properties* list of new node type, otherwise it will be overwritten.

Since a *Property* may contain *properties* as well, the properties substitution is performed recursively. The substitution method is summarized as pseudo code in 2 and 3 (the *properties* attribute is *props*).

Algorithm 1 Connectors Mapping

Input: orgConns, newConns
Output: mappedConns

```
mappedConns = {}
for orgConn in orgConns do
    mappedConns[orgConn.id] = -1
    closestNewConn = minDist(orgConn, newConns)
    if signEqual(orgConn, closestNewConn) then
        mappedConns[orgConn.id] = closestNewConn.id
    end if
end for
```

Algorithm 2 Properties Mapping Main

Input: oldProps, newProps, mgOption
Output: mergedProps

```
mergedProps = []
if mgOption == "option1" then
    mergedProps = newProps
else
    mergedProps = Alg3(oldProps, newProps, mgOption)
end if
return mergedProperties
```

I. Rule Programs

While individual rules can be considered functions which map graphs to graphs, multiple different rules could be used in an orchestrated manner as *rule programs*.

1) Implementation Architecture: As the termination of the RGA cannot be guaranteed in general [4], rules need to be designed for a limited application count. Individual rules within a rule program are expected to be designed to introduce limited, local changes only, which should be usable by other rules.

Program nodes and *program edges* are introduced as structural elements to be used inside rule program. They define the states and transitions of an automaton and hence the execution flow. Figure 7 contains one example, where the edges and nodes in cyan color are program edges and nodes. The rule program starts to be executed from the `Begin` program node, which has connected program edge with `0` indicating that no execution required from the previous step. Afterwards, the state `Substitute` become active, where the upper rule will substitute any node which has name `To_Be_Substituted` and the lower rule will assign name

Algorithm 3 Properties Mapping Iterative

Input: oldProps, newProps, option
Output: mgProps

```
1: mgProps = deepcopy(oldProps)
2: for prop in newProps do
3:     if prop not in mgProps then
4:         {if the new prop not exist in old props}
5:         mgProps.append(prop)
6:     else if prop.props and oldProps[prop].props then
7:         {if the new prop and corresponding old prop (obtained
           by oldProps[prop]) both contain props}
8:         mgProps[prop].props = Alg3(oldProps[prop].props,
           prop.props, option)
9:     else if option == "option3" and notEqual(prop, old-
       Props[prop]) then
10:        {if overwrite when new and old prop are not equal}
11:        mgProps[prop] = prop
12:    end if
13: end for
14: return mgProps
```

`To_Be_Substituted` to any node which is isolated. Both rules will be executed `5` times maximum. The two rules connected to the `Substitute` node could be executed in any order, and if any of them is executed at least once, then they will both be attempted to be executed at least once again. This feature makes sure that the rules connected through the same program node will be executed exhaustively locally first. After the `Substitute` step is done, the `Connect` step is performed if the rules connected to `Substitute` are executed at least `5` steps in total. The *Connect* program node also forms a loop with the `Substitute` program node, which indicates that it will loop back to `Substitute` if the rule connected are executed at least once (program edge between `Connect` and `Loop`, marked `1`).

With this control of the rules attached to the program nodes and edges, flexibility to execute the modification of the original graph in intended order is achieved. Figure 9 is another example which correct the direction of edges in the graph based on the **Arrow_Symbol** neighborhood. If the direction of the edge does not align with the direction of the **Arrow_Symbol**, the existing edge will be deleted and a new edge will be inserted with the correct direction. In the end, the **Arrow_Symbol** will be removed. One example could be found in figure 8 which shows before and after the edge direction correction rule applied to the example graph.

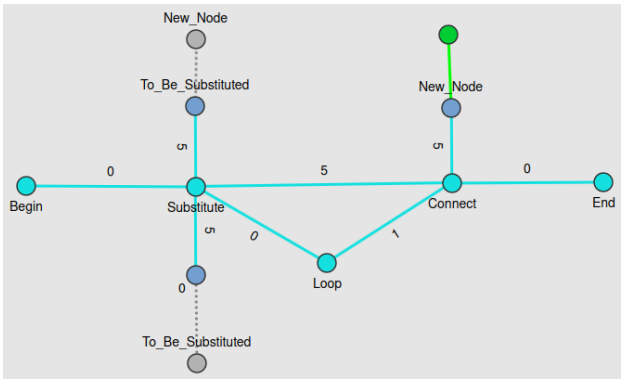
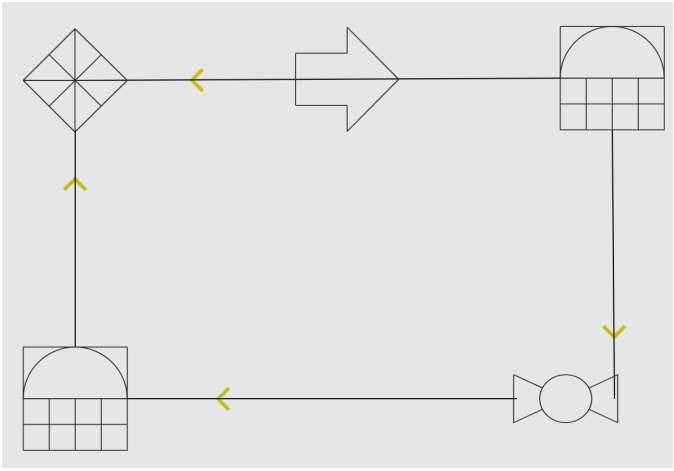
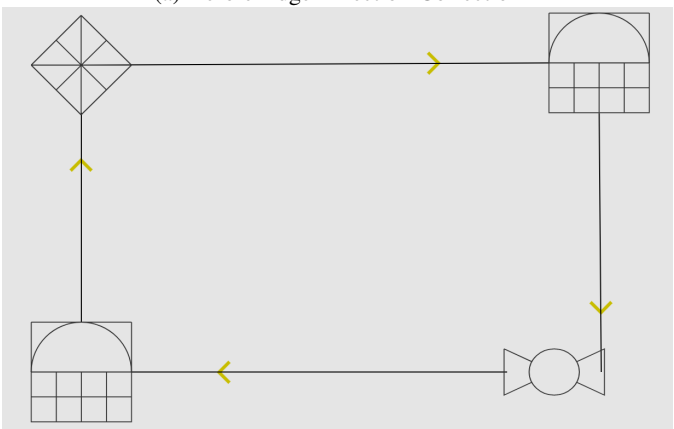


Fig. 7: Rule Program Example



(a) Before Edge Direction Correction



(b) After Edge Direction Correction

Fig. 8: Edge Direction Correction Example.

VIII. Experiment

A. Off-Page Connector Refinement

During todays P&ID engineering processes, P&ID are generally created within CAE systems. These take the context of a node into account when being created. P&IDs house different types of node symbols. One of these are the so called *off-page connectors* (OPCs). These are displayed as arrow symbols and indicate a connection to another P&ID. During the placement process of such OPCs, the CAE distinguishes them based on if they connect two pipes of P&IDs or whether the OPCs connect wires (signal lines) of P&IDs. Since OPC symbols share the same visual appearance the properties of whether they are a piping component or an instrumentation component can only be deduced by the type of edge connecting to it. In order to

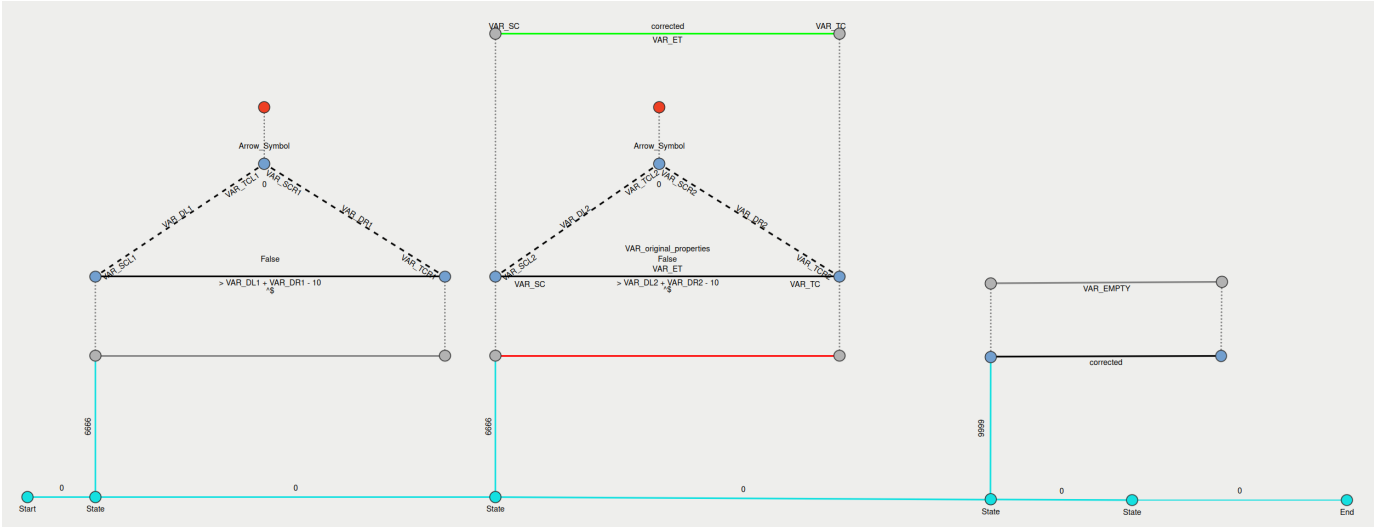


Fig. 9: Rule Program - Edge Direction Correction

fully enrich the OPC node with its intrinsic meaning in CAE applications a property addition should be done, depending on the connecting edgeType.

The second application for properties in the scope of the RGA are common equipments such as valves on P&IDs. Valves can either be ordinary physical valves in the process plant or the valves serve as a controlling function the process-plant. The property of whether a valve has a control function can however not be deduced from the valve node but rather for a different part of the application graph. In order to predict the property controlling function:true/false for a node (valve) a local subgraph has to be taken into account. The determination if a valve has a cooling function is derived from its graph neighborhood. The prediction node in this context it the first degree neighbor node with the nodeType actuator. The Actuator is connected via a certain edge. This existence of a subgraph of node (valve) edge (edgeName) node (actuator) is the subgraph which serves as a property predictor of the initial Valve node.

B. Decomposition of Symbols

During creation of P&IDs with legacy Software such a Bentley Microstations engineers typically are tasked with delivering a proper P&ID on a graphical basis. This allows for customization of Symbols on a P&ID for practical reasons to speed up the creation of P&IDs. A typical example are flanged valves. Flanged valves are two elements which do not have the same meaning and can exist without the other. A process plant can have individual flanges and individual valves. A combination of these two is also possible. In order to derive the smallest possible elements of an application graph node-substitution can be applied. This is relevant due to state of the art CAE systems considering flanged valves not as one node but rather as two separate nodes. To achieve this split of semantic meaning a node has the original flanged valve node needs to be split in its two base objects. Namely a valve and a flange. Similarly, a motorized valve can be split into a motor and a valve. However in order to retain the P&IDs graph structure edges need to be incorporated into the symbol split workflow. This scenario is being handles by the RGA via node substitution and ensures P&IDs can be rebuilt from the data natively stored in the application graph without additional data manipulation on the CAE application side.

C. Recognition Based on Context

Two identical looking symbols are differentiated by their context, resulting in graph-based recognition.

Connector occupancy as a properties/predictor service: When dealing with OPCs it is important to regard whether an OPC serves as a inflow indicator to the process or as a outflow indicator of a process. This distinction cannot be made based on node appearance as both node types share the same visual features. It can also not be derived based on rotation of the node, as inflow and outflow can happened on either ends of the P&ID document. Thus the RGA can act as a additional predictor service such as prediction if an arrow serves as inflow or outflow, based on the connector occupancy status. If an OPC has connector (`id=0`) occupied this means its main property Flow-direction can be considered inflow. This connector occupancy is the sole possible predictor hint to deliver successful properties.

IX. Future Work

A. RDF-Based Implementation

So far, the described RGA has been implemented in a proprietary environment. Reimplementing the mechanisms in open frameworks like Apache Jena [5] or rdflib [6] would allow for simplified integration with other RDF-based sources.

B. Testing Non-Existence

Often, a rule should only apply if a graph is not embedded in certain other structures. A dedicated node type for non-

existence in the original graph could be introduced to address this issue.

ACKNOWLEDGMENT

This research is funded by Bilfinger Digital Next GmbH.

REFERENCES

[1] J. Bayer, A. Sinha, "Graph-Based Manipulation Rules for Piping and Instrumentation Diagrams," Leipzig Symposium on Visualization in Applications 2019.

[2] S. Grüner, P. Weber, U. Epple "Rule-based Engineering Using Declarative Graph Database Queries," 12th IEEE International Conference on Industrial Informatics (INDIN), 2014

[3] Schrr, Andy, and Felix Klar. "15 years of triple graph grammars." International Conference on Graph Transformation. Springer, Berlin, Heidelberg, 2008.

[4] J. Lehmann "Applications of Graph Rewriting and the Graph Rewriting Calculus"

[5] S. Siemer "Exploring the Apache Jena Framework", 2019.

[6] https://github.com/RDFLib/rdflib

[7] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The fujaba environment. In *Proceedings of the 22nd international conference on Software engineering*, pages 742–745, 2000.

[8] Shouvik Mani, Michael A Haddad, Dan Constantini, Willy Douhard, Qiwei Li, and Louis Poirier. Automatic digitization of engineering diagrams using deep learning and graph search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 176–177, 2020.

[9] Rohit Rahul, Shubham Paliwal, Monika Sharma, and Lovekesh Vig. Automatic information extraction from piping and instrumentation diagrams. *arXiv preprint arXiv:1901.11383*, 2019.

[10] Jukka K Nurminen, Kari Rainio, Jukka-Pekka Numminen, Timo Syrjänen, Niklas Paganus, and Karri Honkoila. Object detection in design diagrams with machine learning. In *International Conference on Computer Recognition Systems*, pages 27–36. Springer, 2019.

[11] Mahdi Rabbani, Reza Khoshkangini, HS Nagendraswamy, and Mauro Conti. Hand drawn optical circuit recognition. *Procedia Computer Science*, 84:41–48, 2016.

[12] R Lakshman Naika, R Dinesh, and S Prabhanjan. Handwritten electric circuit diagram recognition: An approach based on finite state machine. *International Journal of Machine Learning and Computing*, 9(3), 2019.

[13] Rachala Rohith Reddy and Mahesh Raveendranatha Panicker. Hand-drawn electrical circuit recognition using object detection and node recognition. *arXiv preprint arXiv:2106.11559*, 2021.

[14] David Eppstein. Subgraph isomorphism in planar graphs and related problems. In *Graph Algorithms and Applications*, pages 283–309, 2002.

[15] Carl Chapman, Kathryn T. Stolee. Exploring regular expression usage and context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 282–293, 2016.