




Article

Gaining Insights into Conceptual Models: A Graph-Theoretic Querying Approach

Danny Medvedev , Uri Shani  and Dov Dori * 

Faculty of Industrial Engineering and Management, Technion—Israel Institute of Technology, Haifa 3200003, Israel; med.danny@gmail.com (D.M.); s.uri@technion.ac.il (U.S.)

* Correspondence: dori@technion.ac.il

Featured Application: A capability to query and gain insights into complex OPM ISO 19450-based conceptual models using OPCloud by answering questions such as “what if”, cause-and-effect interactions, and gap analysis.

Abstract: Modern complex systems include products and services that comprise many interconnected pieces of integrated hardware and software, which are expected to serve humans interacting with them. As technology advances, expectations of a smooth, flawless system operation grow. Model-based systems engineering, an approach based on conceptual models, copes with this challenge. Models help construct formal system representations, visualize them, understand the design, simulate the system, and discover design flaws early on. Modeling tools can benefit tremendously from querying capabilities that enable gaining deep insights into system aspects that direct model observations do not reveal. Querying mechanisms can unveil and explain cause-and-effect phenomena, identify central components, and estimate impacts or risks associated with changes. Being connected networks of system elements, models can be effectively represented as graphs, to which queries are applied. Capitalizing on established graph-theoretic algorithms to solve a large variety of problems can elevate the modeling experience to new levels. To utilize this rich set of capabilities, one must convert the model into a graph and store it in a graph database with no significant loss of information. Applying the appropriate algorithms and translating the query response back to the original intelligible and meaningful diagrammatic and textual model representation is most valuable. We present and demonstrate a querying approach of converting Object-Process Methodology (OPM) ISO 19450 models into graphs, storing them in a Neo4J graph database, and performing queries that answer complex questions on various system aspects, providing key insights into the modeled system or phenomenon and helping to improve the system design.

Keywords: Object-Process Methodology (OPM); OPM ISO 19450; model-based system engineering (MBSE); conceptual modeling; graph databases; query languages



Citation: Medvedev, D.; Shani, U.; Dori, D. Gaining Insights into Conceptual Models: A Graph-Theoretic Querying Approach. *Appl. Sci.* **2021**, *11*, 765. <https://doi.org/10.3390/app11020765>

Received: 10 December 2020

Accepted: 5 January 2021

Published: 14 January 2021

Publisher’s Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Systems engineering, business processes, and workflows are few examples of domains that benefit greatly from conceptual modelling, which expresses them formally and visually. The increase in the complexity of systems, reflected in their models, is manifested in the ever-increasing number of components participating in the model’s structure and behavior and the sheer amount of interdependencies. This complexity renders simple inspection of the model insufficient for answering questions and queries related to relationships among the components, such as “what if”, explaining cause-and-effect interactions, and gap analysis. The high density of the model elements and their interconnections makes it difficult to holistically grasp and comprehend the entire system. The dynamic, operational aspects of the system represented by the model also become complex and hard to follow and predict. As a result, legacy systems, even those that are somehow modeled, are

often left unchanged, as refactoring them poses risks and might not be worth the effort. For evolving systems that undergo continuous changes, impact analysis is very important for decision-making and introducing changes to control flows. Tackling such complex tasks requires analytical queries. To avoid hazardous situations, organizations must model their newly developed or existing systems so they are amenable to being queried effectively and efficiently to enable judicious decision-making and risk assessment. Despite intensive search, we have not found literature on how to systematically query conceptual system models to gain technological, scientific, or business insights.

We present a methodology for querying a conceptual model of a (possibly very complex) system to answer questions related to issues such as the structural relations among system objects, possible execution scenarios, cause-and-effect, and “what if” questions about the behavior of the system.

The article is structured as follows: In Section 2, we briefly introduce Object-Process Methodology (OPM) ISO 19450, the conceptual modeling language used to represent the model to be queried. Section 3 contains a survey of the database and query approaches and types used in academia and industry. Adopting a graph-theoretic approach for model querying, in Section 4 we describe the conversion of an OPM model into a graph representation and its storage in a graph database. In Section 5, we demonstrate the power of OPM model querying in systems engineering and business processes. The implementation of various query types is elaborated in Section 6, and in Section 7 we conclude and propose future work.

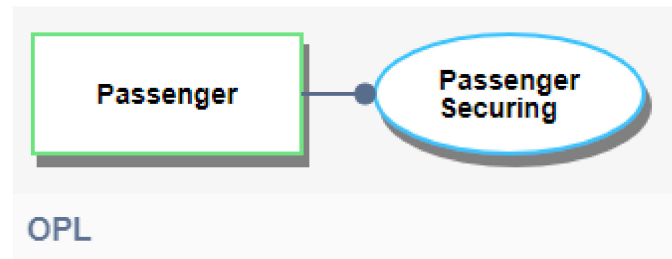
2. OPM—Object-Process Methodology

Object-Process Methodology (OPM) is a conceptual modeling language [1,2] recognized as ISO 19450 [3]. Based on a minimal universal ontology of stateful objects and processes that transform them, OPM supports modelling in a plethora of domains, including systems engineering, business processes, and biological processes. Each OPM model contains one or more hierarchically organized Object-Process Diagrams (OPDs). Each OPD contains OPM things—processes and objects—and links that connect them, each with its own type and semantics. Each OPM thing can be refined (in-zoomed or unfolded), creating a new, descendant OPD that describes the nested processes and related lower-level objects that are involved in these subprocesses.

The elements of an OPM model are OPM things and links, which connect the OPM things. A link can be procedural or structural. A procedural link connects two OPM things with opposite persistence, i.e., an object (rectangle) and a process (ellipse; see Figure 1). These links provide information about the system’s dynamic, time-dependent aspects, including generation and consumption of objects or a change in an object state. Structural links provide time-invariant information, such as aggregation-participation, between two or more OPM things with the same persistence, i.e., between objects or between processes (see Figure 2). OPM is bimodal—it is expressed simultaneously in both graphic diagrams, the OPD set, and in text, a subset of a natural language called Object-Process Language (OPL). As shown in Figures 1 and 2, the OPL sentences are visible below their corresponding OPDs. The textual OPL representation is generated on the fly in response to the modeler’s graphic interaction with OPCloud [4,5], the OPM modeling software. We implemented the graph-based querying and insight gaining extension within OPCloud.

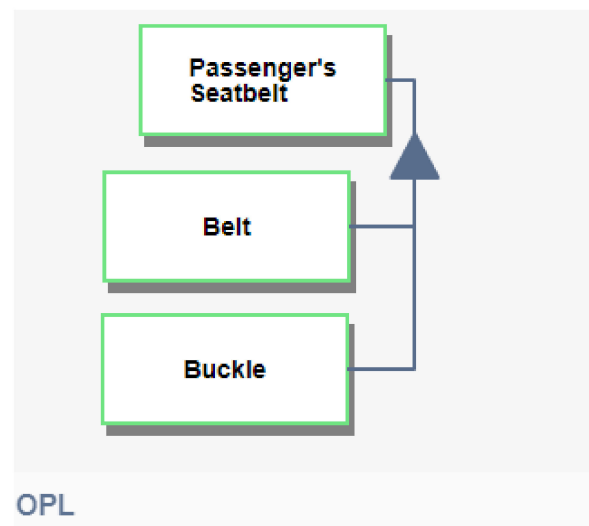
The multi-layered structure of an OPM model provides both a high-level view and an option for refinement, “deep-dive”, or drill-down, to specify additional details at increasing levels of detail as the modeler deems necessary to describe the system fully. The system that the model represents can be easily understood, since each OPD is not overly complicated, and the relations between any two OPDs in the model are direct refinement-abstractation relations, where one of the things in the descendant OPD is a refined (in-zoomed or unfolded) version of an OPM thing in the ancestor OPD. The entire OPM model is thus presented as a hierarchy of OPDs—the OPD tree. The entire model includes the union of the model facts expressed in all the OPDs of the model.

To enhance model understanding, the query capability is fully integrated into the two OPM modalities, so the query results are both graphically and textually, with each modality cognitively complementing the other.



Passenger handles Passenger Securing.

Figure 1. An Object-Process Methodology (OPM) model with a procedural (agent) link.



Passenger's Seatbelt consists of Belt and Buckle.

Figure 2. An OPM model with a structural (aggregation-participation) link.

3. Queries and Databases

3.1. Conceptual Model Querying

Conceptual modeling generates a representation of a system in some domain. A detailed model can help improve, develop, identify gaps, and optimize the represented system [6]. The goal of conceptual modeling is to establish a framework that facilitates understanding of the problem space, synthesis of possible solutions, and analysis of modeled solutions. Model querying is important, as it provides for information aggregating and filtering, enabling better model understanding and detection of flaws. Effective querying is required not only for humans but also for automated analysis systems.

Object-oriented modeling is supported by the Unified Modelling Language (UML) [XE "UML:Unified Modelling Language"]. A single kind of UML diagram, such as a class diagram, is not enough to define all the aspects of a specification. Therefore, it has 14 kinds of diagrams. UML includes a textual Object Constraint Language (OCL) [XE "OCL:Object Constraint Language"] [7], which can express detailed aspects about the modelled system. OCL was originally designed specifically for expressing constraints about a UML model. However, its ability to navigate the model and form collections of objects has led to

attempts to use it as a query language. OCL can be used as a basis for several different languages, and it has indeed served as a basis for the Query/Views/Transformations (QVT1) {XE “QVT:Query Views Transformations”} standard [8].

The Structured Query Language (SQL) has become synonymous for database querying that serves relational database management systems (RDBMSs), and it has many implementations. Object Query Language (OQL) XE {“OQL:Object Query Language”} [9] has been too complex, hindering full implementation of its semantics. Although there are many OQL variations, a fully accepted OQL equivalent to SQL for querying an object-oriented data model does not exist.

Prolog [10] is a declarative programming language based on the logic programming paradigm. Due to its rule-based approach, it has been successfully applied to implement query engines for query languages based on various data models, e.g., for querying RDF [11] data with SPARQL [12]. Prolog provides metamodel-independent facilities to implement query engines for many different modeling languages, such as UML and Event Driven Process (EPC) chains {XE “EPC:Event driven Process Chains”}.

OMG Systems Modeling Language (SysML) {XE “SysML:Systems Modeling Language”} [13] is used to build comprehensible models of engineering systems, but it lacks a formal basis, so compatibility checks cannot be performed on it directly. An approach that enables explicit knowledge representation and application of inference mechanisms is the Web Ontology Language (OWL). {XE “OWL:Web Ontology Language”} However, OWL targets formal knowledge representation and lacks modeling abilities. Therefore, Feldmann et al. [14] have proposed to combine systems modeling with semantic technologies, specifically OWL, in order to enhance the capability to analyze change influences. In their proposal, SysML enables creating an interdisciplinary system model with the required information, while OWL can be utilized for the formal compatibility check.

3.2. NOSQL Databases

NOSQL databases [15] are emerging alternatives to the most widely used SQL databases—RDBMSs. Commonly interpreted by developers as “not only SQL databases”, it does not completely replace SQL. Rather, it complements it in a way that enables the two to coexist. NOSQL solves scalability issues that relational databases suffer from due to current sheer data volumes and the resulting performance degradation. Unlike RDBMS, NOSQL is designed to scale up as the data volume grows. The NOSQL data model does not guarantee the ACID properties: Atomicity, Consistency, Isolation, and Durability that are fundamental in SQL databases. Instead, it guarantees the BASE properties: Basically Available, Soft State, and Eventual Consistency. NOSQL follows the Consistency, Availability, and Partition (CAP) {XE “CAP:Consistency, Availability, Partition”} tolerance theorem [16], which states that it is impossible for a distributed service to be consistent, available, and partition-tolerant at the same instant in time. Consistency means that all the copies of the data in the system appear the same to the outside observer at all times. Availability means that the system as a whole continues to operate even in cases of node failure. Partition-tolerance requires that the system continues to operate despite an arbitrary message loss, which can result from a crashed router or a broken network link, preventing communication between groups of nodes [17]. NOSQL databases are categorized into the following five types.

- 1. Key-value store:** The data consists of two parts: a string that represents the key and the actual data, referred to as value, creating a “key-value” pair. These stores are like hash tables, where the keys are used as indexes. The key-value stores offer significant performance and scalability advantages compared with traditional databases. They achieve these properties through a restricted application programming interface (API) {XE “API:Application Programming Interface”} that limits object retrieval: an object can only be retrieved by the (primary and only) key under which it was inserted [18].

- 2. Column store:** Column-store systems, such as BigTable, Cassandra, Amazon DynamoDB, and RIAK, completely partition a database vertically into a collection of individ-

ual columns that are stored separately. By storing each column separately on secondary storage, these column-based systems enable queries to readjust the attributes they need rather than having to read entire rows from that storage and then discard unneeded attributes once they are in memory [19].

3. Document stores: Document stores, such as MongoDB and CouchDB, are tuned to store unstructured (text) or semi-structured (XML, JSON) documents, which are usually hierarchal in nature. Each document consists of a set of keys and values, which are like the key-value databases. Each document in the database stores points to its fields using hashed pointers [20].

4. Object-oriented databases: Object-oriented databases, such as db4o, follow the object-oriented programming languages paradigm, in which the data or information to be stored is represented as an object. Thus, an object-oriented database can be considered a combination of principles from the object-oriented programming and database paradigms [21].

5. Graph databases: Graph databases, such as Neo4j, are built on graphs. As they provide the basis of this work, we elaborate on them next.

3.3. Graphs and Graph Databases

A graph consists of a set of nodes (vertices) connected by links (edges) [22]. Each node and each link contain metadata that describes its type and optionally additional properties. A set of vertices v connected by a set of edges e forms a mathematical structure for a graph $G = (v, e)$. Graph nodes can have properties. Nodes in the graph can be objects, and edges—relationships between the objects [23]. Graphs are used in multiple domains, such as computer science, where graph algorithms are a major research and application field. Graphs describe pivotal subjects, such as communication, flow of computation, and data organization.

Graphs can be stored in a graph database (GDB), which is geared towards storing and querying graph-structured data. GDBs use an index-free adjacency technique: every node has an implicit link to the adjacent node, providing for millions of records to be traversed quickly. In a GDB, the main emphasis is on the connections between data items, which provide schema-less and efficient storage of semi-structured data. GDBs serve in a variety of applications, such as social networking, recommendation software, bioinformatics, content management, security and access control, network, and cloud management.

GDB queries are expressed as graph traversals, making them much faster than relational database queries. They are scalable and whiteboard-friendly, allowing for expressiveness in data modeling by drawing nodes and relationships, and only then implementing the metadata related to the nodes and the relationships [24]. GDBs are ACID-compliant and offer rollback support. They usually require only a single query to retrieve linked data. In relational databases, however, extracting the same information would require complex queries that perform inefficient and expensive joins over multiple tables.

A GDB query language is not SQL, as it is designed to traverse a graph [25]. Unlike the case with SQL for relational DBMSs, as of 2018 no universal standard has been adopted as the GDB query language of choice. However, several Qs, such as Gremlin [26], SPARQL [12], and Cypher [27], are very popular; they are supported and distributed by multiple vendors and used by many organizations.

A notable Database as a Service (DBaaS) provider that uses a GDB is Neo4j. Since Neo4j is used in this research, we discuss it next. Neo4j is an open source, NOSQL native graph database [28]. Its initial development began in 2003, but it has been publicly available since 2007. The source code, written in Java and Scala, is available for free on GitHub. Considered a world-leading, high-performance graph database, Neo4j is a native GDB because it efficiently implements the property graph model down to the storage level. This means that the data is stored exactly as expected, and the database uses pointers to navigate and traverse the graph [29]. Neo4j is currently the top-rated graph database

management system [30], scoring almost twice as high as that of Microsoft Azure Cosmos DB, which ranks second (see Table 1).

Table 1. Graph database (DB)-engines ranking for December 2020¹ [30].

Rank December 2020	DBMS	Database Model	Score December 2020
1	Neo4j	Graph	54.64
2	Microsoft Azure Cosmos DB	Multi-model	33.54
3	ArangoDB	Multi-model	5.51
4	OrientDB	Multi-model	5.29
5	JanusGraph	Graph	2.65
6	Virtuoso	Multi-model	2.58
7	Amazon Neptune	Multi-model	2.51
8	GraphDB	Multi-model	2.05
9	FaunaDB	Multi-model	2.01
10	Dgraph	Graph	1.69

¹ Based on <https://db-engines.com/en/ranking/graph+dbms>.

The current Neo4j version 3.5.14 uses Cypher [27] as its query language. Cypher is a declarative, SQL-inspired language for describing patterns in graphs visually using an ascii-art syntax. Cypher allows stating what to select, insert, update, or delete from the data expressed by the graph, without requiring an exact description of how to do it.

Neo4j uses the Atomic, Consistent, Isolation, and Durable (ACID) model to ensure that the data is safe and stored consistently [31], making it reliable, highly available, and scalable. It offers a Representational State Transfer (REST) (XE “REST:Representational State Transfer”) interface, a coordinated set of architectural constraints that attempt to minimize latency and network communication [32], and a convenient Java API that can be embedded into JAR files. Neo4j is available in a GPL3-licensed open-source “community edition”, with online backup and high availability extensions licensed under the terms of the Affero General Public License [33]. Cypher, Neo4j’s graph query language, allows users to store and retrieve data from the GDB [34]. Incorporating the power and functionality of other standard data access languages, Cypher is easy to learn and understand. Fortune 500 companies, such as Adobe, Accenture, Cisco, Lufthansa, Telenor, and Mozilla, are only a small sample of those that use Neo4j. We use Neo4j as the GDB and Cypher as the query language for creating, modifying, and retrieving data.

4. Mapping an OPM Model to a Graph

Unlike an OPM model, which has several kinds of nodes and several kinds of links (edges), a graph has just one kind of node and one kind of edge. What differentiates between the elements is their properties—the metadata: each node and edge contain metadata that describes their properties. Merging the OPM models in Figures 1 and 2 produces the diagram on the left-hand side of Figure 3, for which the graph representation with five nodes and four edges is depicted on the right-hand side.

Each process in an OPM model can be refined—in-zoomed or unfolded—creating a new, more detailed OPD, with nested subprocesses and possibly additional objects that are expressed at a greater level of detail than its predecessor OPD. While these visual capabilities are critical for the human end user to understand the model, the multi-layered structure of an OPM model complicates the generation of a graph from the model, requiring pre- and post-processing, as described next.

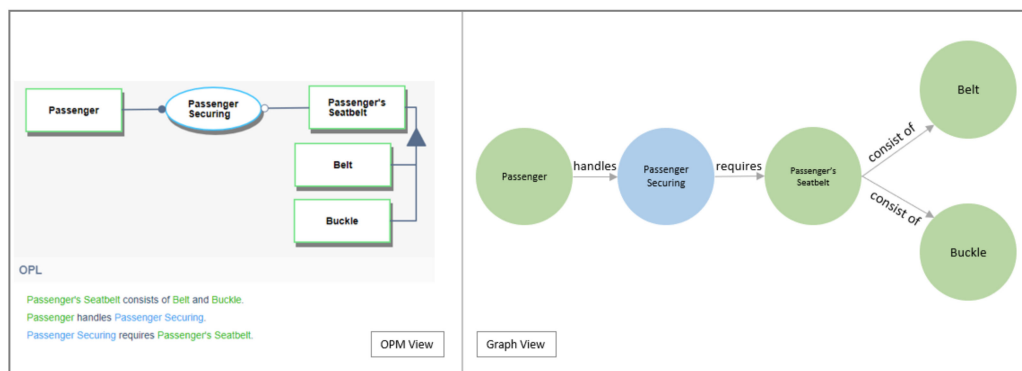


Figure 3. A graph representation (right) of the OPM model (left), resulting from merging the models in Figures 1 and 2.

4.1. Nested OPM Thing Flattening

Unlike the multi-layered hierarchical structure of an OPM model visual representation, in the GDB we use a single graph for the entire model, which mandates collapsing or flattening the OPM model hierarchy altogether. Prior to converting an OPM Model to a graph, we need to collapse the multi-layered OPM model representation into a single-layered OPM model, in which all the OPDs in the OPD tree are merged into a single, “flat” OPD. To perform this, we use a flattening algorithm, developed as part of this research. The Flattening algorithm merges all the OPDs into a single OPD without any loss of information.

During OPM model flattening, elements (things and links) that appear in an OPD without any relation to any one of the nested processes are relatively easy to merge; they are copied to the new flat, single-layered, single-OPD OPM model. However, any element that has a direct or indirect relation to a nested process requires extra processing by the Flattening algorithm. In some cases, the algorithm omits some of the links, and in other cases it adds new elements, as described next. Another approach that avoids flattening altogether can be an alternative option.

4.1.1. Stateful Object De-Stating

A stateful object is an object with two or more states such that it may transition amongst them during its lifetime, and which thus describe its possible situations (or values if the object is an attribute). As part of the Flattening Algorithm, we perform a “de-stating” process based on Algorithm 1, in which an object with n states is converted into n stateless objects that specialize the original stateful object. A pseudo-code of this algorithm is as follows.

Algorithm 1: Stateful Objects De-stating Algorithm

1. For each object with n states, where $n \geq 2$
 - 1.1. For each object state
 - 1.1.1. Generate a new stateless object, where the name of each such object is a concatenation of the state name followed by the name of the original stateful object.
 - 1.1.2. Migrate any state’s connected link to the new generated object
 - 1.1.3. Remove state from original object
 - 1.2. For each of the n newly generated objects
 - 1.2.1. Generate OPM Generalization-Specialization structural link connected to the original object

end

From a model complexity perspective, the number of model elements remains the same, as the object and its n states simply become $n + 1$ stateless objects. The number of logical links does not change, but the number of procedural links increases by n . To demonstrate this algorithm, the OPM model example on the left-hand side of Figure 4 contains one stateful object, **Passenger** (Names in bold letters are OPM things in the models in the figures), with two states: **unsecured** and **secured**, connected to the **Passenger Securing** process with an input–output link pair. Algorithm 1 generates two new stateless objects: **Unsecured Passenger** and **Secured Passenger**, where the input–output link is split into a consumption link from the newly created object **Unsecured Passenger** to the process **Passenger Securing** and a result link from **Passenger Securing** to the newly created object **Secured Passenger**. The original object **Passenger** has become stateless, and it is now linked with an OPM generalization–specialization link (the blank triangle) to each of the two new objects.

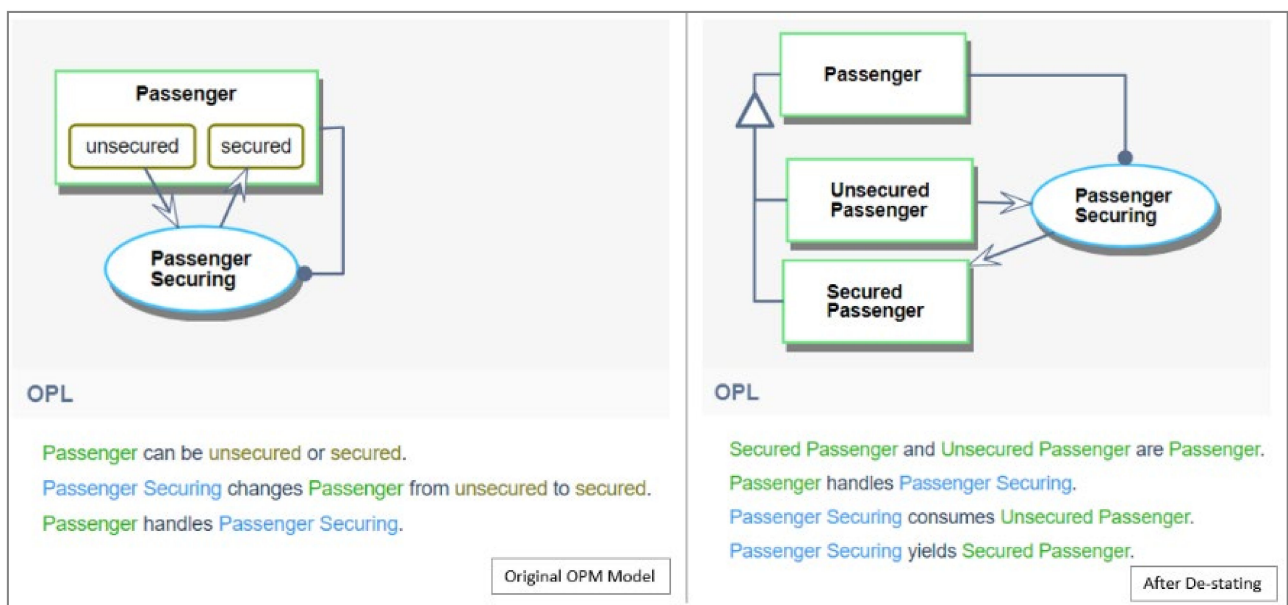


Figure 4. Example of stateful object de-stating. Left: original model with a stateful object. Right: de-stated object with two specializations and links correctly migrated.

4.1.2. Processing Links in Nested OPM Things

The OPM links connected to refined (in-zoomed or unfolded) OPM things can be direct, which means that they are incoming to or outgoing from the nested process. In Figure 5, the **unsecured** and **secured** states are linked directly to the subprocess **Seatbelt Pulling** and from the subprocess **Buckle Clicking**. Indirect relations are connected to the circumference of the enclosing, in-zoomed process. An OPM enabler is an object that enables the process occurrence but is not transformed by that process. In Figure 5, the objects **Passenger Seatbelt** (an instrument, i.e., a non-human enabler) and **Passenger** (an agent, i.e., a human enabler) are connected with OPM enabling links (an instrument link—the white lollipop—and an agent link—the black lollipop—respectively) to **Passenger Securing**. Additional attention needs to be paid to OPM things that appear in more than one OPD and at different detail levels.

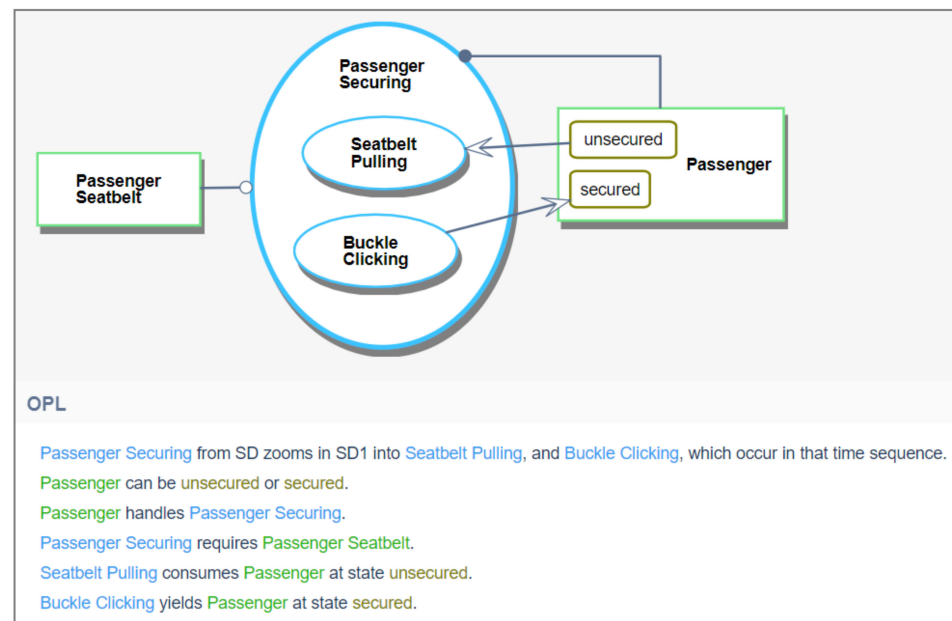


Figure 5. The in-zoomed process **Passenger Securing** with links to the nested processes and to the circumference.

A pseudo-code for the entire Flattening algorithm is listed in Algorithm 2.

Algorithm 2: OPM Model Flattening Algorithm

1. Clone all the OPM model elements to a new flattened OPD
2. Execute the Stateful Object De-Stating algorithm (Algorithm 1)
3. For each OPM Thing connected (incoming/outgoing link) to/from an in-zoomed or unfolded OPM Thing in the parent level OPD
 - 3.1. If the OPM Thing is not connected to any nested OPM Thing in any child level OPD
 - 3.1.1. Copy OPM Link from original OPM Thing to each nested OPM Thing
 - 3.1.2. Remove original OPM Link from OPM Thing to the in zoomed OPM Thing
 - 3.2. Else remove OPM Thing and OPM Link to the unfolded OPM Thing
4. For each nested OPM Thing in child level OPD
 - 4.1. In case OPM Thing is an Object
 - 4.1.1. Create a new OPM Unidirectional Tagged Link between the nested objects ordered by the graphical position within the unfolded object
 - 4.2. In case OPM Thing is Process
 - 4.2.1. Create a new OPM Invocation Link between the nested processes ordered by the graphical position within the unfolded process
 - 4.2.2. Create a new OPM Aggregation-Participation Link from the unfolded process to the nested processes

end

The complexity of this algorithm is linear, $O(n)$, by the number n of elements (things and links). The de-stating algorithm discussed above contributes at most m links, where m is the total number of states in the model. Likewise, k , the number of new links due to Step 4.1 of the algorithm, is the number of elements in refined OPDs. Thus $m < n$, and $k < n$, and the complexity of this algorithm is $O(n + m + k) = O(n)$.

To demonstrate this algorithm, suppose the OPD on the left-hand side of Figure 4 contains an additional instrument, **Passenger Seatbelt**, with an outgoing instrument link to the process **Passenger Securing**. The agent **Passenger** has two states: **unsecure** and **secure**,

connected with an input–output link pair. Figure 5 is a descendant OPD that contains two nested subprocesses inside **Passenger Securing: Seatbelt Pulling**, with a split input–output link pair, depicted as an incoming consumption link from the state **unsecure** of the **Passenger**, and **Buckle Clicking**, with an outgoing result link to the state **secure**. The OPD in Figure 6 is the result of applying the Flattening algorithm on this model.

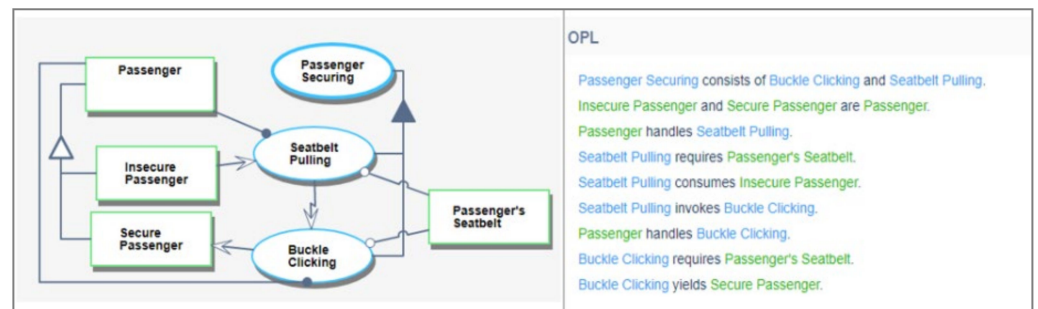


Figure 6. The result of applying the Flattening Algorithm on the **Passenger Securing** system.

Step 4.2 in the Flattening Algorithm adds two new agent links from the agent **Passenger** to the two subprocesses that were nested within the in-zoomed process and are now unfolded: **Seatbelt Pulling** and **Buckle Clicking**. The algorithm also adds two new instrument links from the instrument **Passenger Seatbelt** to these two subprocesses, replacing the original one from **Passenger Securing**.

Step 4.2 is demonstrated by (1) a new aggregation-participation link that is added from the originally in-zoomed and now unfolded **Passenger Securing** process to its two subprocesses, **Seatbelt Pulling** and **Buckle Clicking**; and (2) a new invocation link between the nested processes, based on the OPM timeline principle. This principle states that, within an in-zoomed process, a subprocess whose vertical position is higher is performed first. Applying this principle, within the original **Passenger Securing** process, **Seatbelt Pulling** is above **Buckle Clicking**, expressing the model fact that the former is performed before the latter. In the unfolded version of **Passenger Securing**, this model fact is expressed by the invocation link (the lightning-like arrow) from **Seatbelt Pulling** to **Buckle Clicking**.

4.2. From an OPM Model to a Cypher Query

We use the cloud based OPM modeling software environment OPCloud [4,5] as the implementation platform for representing and querying the OPM model. OPCloud is an Angular application, and it uses Rappid [35] for visualizing the OPM model from its internal representation. The GDB we use is Neo4j [28], which supports several querying facilities, including the Cypher language [27,34]. Using Cypher on a Neo4j-stored graph model enables generating, updating, and querying the models in the GDB. To generate the OPM model representation in the Neo4j GDB, we convert the OPM model to Cypher, from which we generate the graph representation directly in the GDB.

4.2.1. The Neo4j Cypher Query Language

Cypher is an easy-to-master pattern-matching language, enabling querying (by matching patterns in the graph), construction and modification of the graph, and specifying schemata for the graph. A Cypher statement is comprised of several phrases, starting with a graph-specific matching part, followed by optional SQL-like phrases to facilitate further post-processing of the matching outcome, and generating the result to be returned.

Cypher was originally intended to be used only with Neo4j, but in 2015 Cypher was opened up for more general use through the openCypher [36] project. The patterns follow the basic “triple” construct, whereby a graph is a collection of triples, each defining a link between two nodes and their properties. Like OPL, Object-Process Language, which is the textual representation of the OPM model, the Cypher query language is easy to read due to its graphics-oriented ascii-art syntax.

Each Cypher statement starts with a verb declaring its purpose: MATCH, CREATE, and MERGE, the triples-matching phrase, the optional follow-up SQL-like modification phrases, additional update and alteration phrases, and finally the optional RETURN part.

The pattern matching syntax, demonstrated in examples and algorithms in this paper, is of particular interest: Nodes are enclosed in round parentheses, e.g., (<node>), and links are enclosed in brackets, e.g., -[<link>]->. Basic triples are ()-() for a non-directional triple, ()->() for a right directional triple, and ()<-() for a left-directional triple.

Within the parentheses, each <node> or <link> consists of three optional parts: <var name>:<type><{property: 'value', ... }>. A variable name <var name> helps to refer to matched elements further in the statement, as well as in the RETURN phrase. The <type> specifies a classification of the element or link. In our case, these are OPM element types, such as process and object. Properties further enrich the element with a data record that can be used for matching and insertion of their values in statements.

A full example of a triple in the Cypher query language is:

```
(node1:TypeA {propertyKey: 'Value'})-[rel1:RelTypeA]->(node2:TypeB).
```

The variables node1, rel1, and node2 hold references to the respective elements of the triple and can be further used in optional follow-up phrases. If there is no use for a variable, it can be omitted, as we do in some of the examples in this paper to reduce clutter.

As all of the graph content comes from the logical model structure in OPCloud, we reference these values using a macro-like \$-NOTATION, such as \$OPM_THING_TYPE in Code 1.

4.2.2. Applying Cypher Queries on the OPM Model

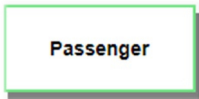

Each logical element in the model is an instance of an OPM element, which can be an OPM thing—an object or a process—or an OPM link. In the first phase of the model-to-graph conversion, we apply the OPM Model Flattening Algorithm (Algorithm 1, above). The flattened, single layered OPM model, expressed by a single OPD is equivalent to the entire original model, as it holds all the visual representations of the OPM model things and all the links that connect them, as expressed by the entire set of hierarchical organized original OPD tree.

Each graph node is associated with a specific OPM thing, and the nodes are linked by edges carrying the original OPM model link semantics. Code 1 is the Cypher code block of the query used for node generation, and Table 2 demonstrates the use of this query.

Code 1. Cypher query for node creation in the Neo4j graph database.

```
MERGE (:$OPM_THING_TYPE {name:'$OPM_THING_TEXT', opm_id: '$OPM_THING_ID'})
```

Table 2. Example of converting OPM things to Cypher nodes.

OPM Thing	Cypher Node Creation Query
	MERGE (:object {name: 'Passenger',opm_id: '1'})
	MERGE (:process {name: 'Passenger Securing',opm_id: '2'})

The OPM links in a graph are the edges that connect the nodes. Each OPM link is checked to identify all its sources and targets. Each “source”–“link”–“target” triple is represented in a customized object structure that contains the following metadata: {Source => {Type, Text, Id}, Link => {Type, Id}, Target => {Type, Text, Id}}. Each triple is unique in that it is based on the combined content of the three elements. Converting the OPM model to such triples can later be used for additional benefits, such as converting it

to the Resource Description Framework (RDF) [11,37] representation or any other data representation, and for new analysis methods.

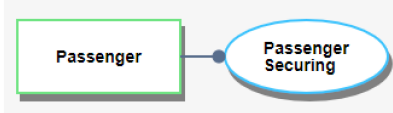
To generate a triple that contains the two nodes and one edge in a graph database, three “OPM Thing to Cypher” queries are required: one for the source element, one for the target element, and one to create the link between these two. The Cypher query to generate a single edge includes a search (MATCH) for the two connected nodes and a generating (MERGE) query to create a triple of the two identified nodes and a link between them, as presented in Code 2.

Code 2. Cypher query for relationship creation in a graph database

```
1: MATCH (n$INDEX:$S_OPM_THING_TYPE
{name:'$S_OPM_THING_TEXT',opm_id:'$S_OPM_THING_ID'})
2: MATCH (n($INDEX+1):$T_OPM_THING_TYPE
{name:'$T_OPM_THING_TEXT',opm_id:'$T_OPM_THING_ID'})
3: MERGE (n$INDEX)-[r:'$OPM_Link_Type' {opm_id: '$OPM_LINK_ID'}]->(n($INDEX+1))
```

An example of converting an OPM Link to a Cypher relationship is depicted in Table 3.

Table 3. Example of converting an OPM link to a Cypher query.

OPM Link	
Cypher relationship creation query	<pre>MATCH (n1:Object {name:'Passenger' , opm_id:'1' }) MATCH (n2:Process {name:'Passenger Securing' , opm_id:'2'}) MERGE (n1)-[r:'Agent' {opm_id: '3'}]->(n2)</pre>

5. Use Cases for OPM Queries

In this section, we present and discuss several use cases, in which querying an OPM model provides valuable insights about the represented system. These include cause-and-effect in model-based systems engineering, business process optimization using path finding, and thing criticality in product quality assurance using the *PageRank* algorithm.

5.1. Cause and Effect in Model-Based System Engineering

Complex engineering systems are comprised of subsystems with multiple processes and components with multiple interconnections and interdependencies. Therefore, any change, minute as it may seem, can cause unexpected behavior in any one of the other related components. The impacts of such change are often unknown and can potentially have an adverse effect on the system, so changes to the system must be considered carefully before they are applied. A change can also have a positive effect, yet it must be considered before it is applied. Risk and impact analyses are thus crucial to the viability of complex systems. An OPM model of such a complex system, converted to a graph in a GDB, can reveal and explain what went wrong or what can go wrong with objects that the processes along the path of execution transform (consume, create, or affect), what objects are required as enablers (agents or instruments) for these processes to happen, and what needs to be done to avoid failures.

5.1.1. Impact Analysis by Forward Chaining: The Case of the Aircraft Collision Avoidance System

Systems engineers often face the challenge of “What will be the impact of my change?” The larger and more complex the system, the more difficult it is to answer this question. A similar challenge exists when a system element fails, with the question being “What disruption will this failure cause to the operation of the system?” An OPM model of a system, converted to a graph representation, will often reveal a sub-graph directed

“downstream”, rooted from the affecting OPM thing, which includes the things connected to it. This information enables the systems engineer to evaluate the impacts and risks of planned and unplanned changes on the system’s health and performance.

Figure 7 is an OPM model of an **Airplanes Collision Avoiding System** with three levels of detail, created by iteratively zooming into the processes. The model describes the stateful object **Aircraft 1**. The **Airplanes Collision Avoiding** process changes the state of **Aircraft 1** from **close** (i.e., being near an obstacle) to **collision avoided**. The process consists of the subprocesses **Collision Detecting**, **Route Change Calculating**, and **Collision Avoiding**. The in-zoomed process **Collision Detecting** requires the instrument **Vicinity Screening Subsystem** to execute the process with the subprocesses nested within it.

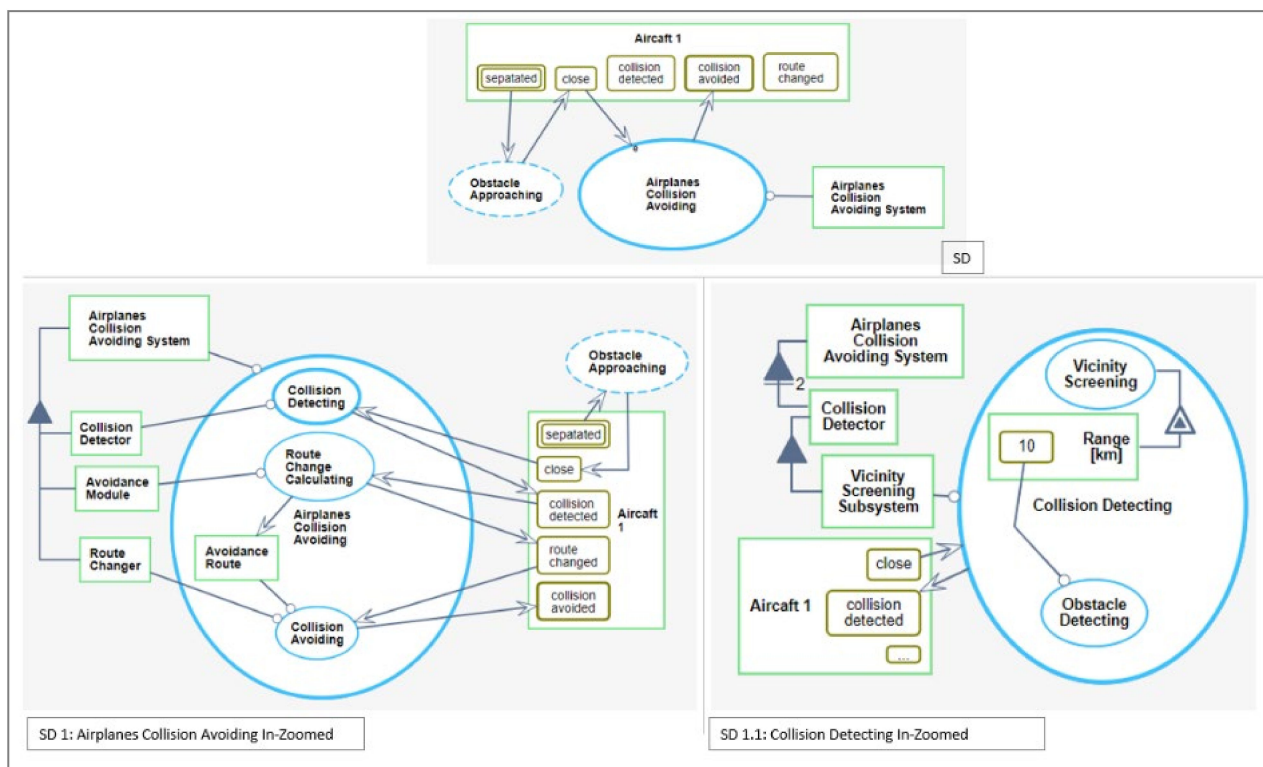


Figure 7. An **Airplanes Collision Avoiding System**. Top: The system diagram, SD; bottom left: SD1—first detail level; bottom right: SD1.1—second detail level.

Suppose there is a scheduled maintenance task of upgrading the **Vicinity Screening Subsystem**, which is specified in Figure 7 at the second detail level, SD 1.1: **Collision Detecting In-Zoomed**. The team responsible for executing this task does not have the full picture of the possible impact of a failure of this system. Hence, their risk assessment of the system’s operation will be very limited, as from the viewpoint of SD1.1, it seems that removing the **Vicinity Screening Subsystem** would impact only **Vicinity Screening** and **Obstacle Detecting**.

Executing a “Directional” query, which is one of the queries described below, collectively named “Neighborhood OPM Queries”, and filtering only the “Outgoing” relationships from the **Vicinity Screening Subsystem** as the graph root node will provide all the OPM things that are impacted due to element failure or disruption. The Cypher query to retrieve the data is as follows:

```
1: MATCH p=(source:$START_NODE_TYPE {name:'$START_NODE_TEXT'})-[*1..$MAX_DEPTH]->(child)
2: RETURN p
```

The result of the query presented in Figure 8 provides additional elements to consider beyond the obvious ones, **Vicinity Screening** and **Obstacle Detecting**. The result includes

elements from all the model levels and the relationships among them, clearly showing that failure or disruption of this subsystem has a major impact, as it might prevent the system from being able to avoid an obstacle. The three objects, **Collision Detected Aircraft 1**, **Route Change Aircraft 1**, and **Collision Avoided Aircraft 1**, result from de-stating (see Section 4.1.1) of the 5-state object (object with five states) **Aircraft 1**.

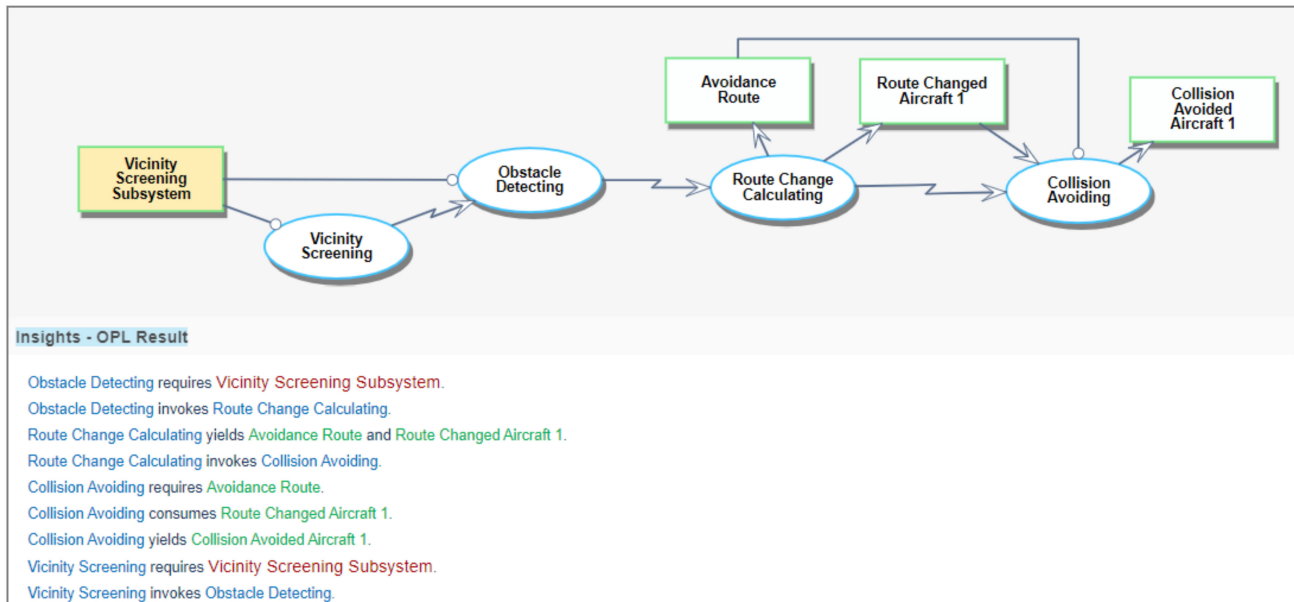


Figure 8. Neighborhood OPM query result of an airplane collision avoidance model.

System monitoring and controlling capability is an important factor in ensuring system stability and health, as it provides for continuous validation, verification, and incident mitigation. Cyber–physical systems are prime examples, in which the sensors and indicators that constitute their “cyber” part, are commonplace, and they monitor and control the physical subsystems, especially the critical ones. Mordecai and Dori [38] demonstrated how the 1979 Three Mile Island 2 (TMI-2) nuclear accident could be mitigated using OPM as a conceptual modeling framework. A graph representation of the OPM model, which includes mainly procedural links, reveals an upstream sub-graph that explains step-by-step what went wrong, leading to the eventual core meltdown. The explanatory power of the graphical representation is enhanced by examining the OPL paragraph that describes the accident causes. In cyber–physical systems, such a description helps determine the cyber–physical gap—the gap between the physical and the cyber realities—leading to an improved understanding of the system vulnerabilities and how to tackle them.

5.1.2. Possible Causes by Backward Chaining: The Case of the Three Mile Island Nuclear Accident

The accident in the second reactor of TMI-2 was caused due to many reasons, which started eleven hours before the meltdown. To demonstrate the OPM query capability, we focus on the **Pilot Operated Relief Valve (PORV)** component in the steam generating subsystem. According to the TMI control room operators’ understanding, there were only two states that the **PORV** can be at: **closed** or **open**. The same is true for the instruments of the control room, which indicate only if the **PORV** is **open** or **closed**. The post-incident review of this accident shows that the **PORV** was in a “**stuck-open**” state—a state that had never before been considered as possible. The operators were not aware of the cooling water pouring out through the **stuck-open PORV**. Because of that, the operators did not identify the loss of coolant accident that caused the core to overheat and initiated the nuclear fuel pellets’ cladding to rupture, eventually leading to core melting.

A fault-aware model, described in [38] as the second version of implementing the “Cyber Physical Gap Aware Modeling and Engineering” (CPGAME), is a model that enhances the original one with potential faults of each component. The second of the three model versions describes only the physical failures and system identification for those failures. Designers and engineers usually avoid this kind of modeling, as it involves failed component states, putting in question its reliability. In our case, the “melted” state of the reactor core was not considered.

Figure 9 shows the process **Steam Generating** in-zoomed with the related objects, including the process **Meltdown**, which changes the state of the stateful object **Reactor Core** from its initial state **cooled** to **melted**.

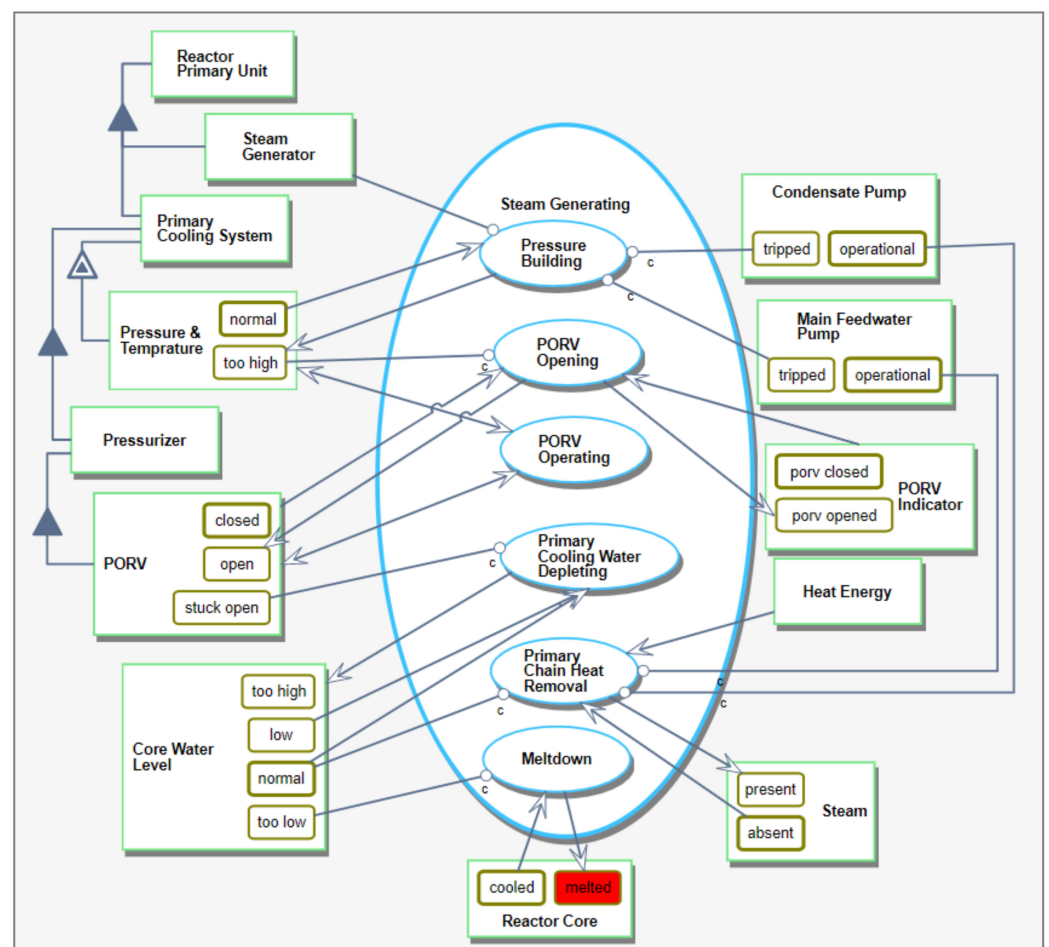


Figure 9. OPD of the fault-aware **Steam Generating** process in-zoomed.

Mitigation of such accidents could be achieved by reviewing all the processes, objects, and object states of the failure aware OPM model, and following the chain of events that leads to a change in the system’s health into an instable or even a disastrous state, as modeled in Figure 9. We execute the Directional query within the Neighborhood OPM Queries, placing the state **melted** of the **Reactor Core** object as the graph’s root node and specifying “Incoming” for the “Direction” property of the “Neighborhood Queries” (see Section 6.2), thus asking for all the incoming links. The query result in Figure 10 shows all the elements that contributed to the meltdown as part of the **Steam Generating** process. The **PORV** at state **Stuck Open** and **Core Water Level** at state **too low** are the failure-aware elements that trigger the processes that eventually caused the accident of the nuclear fuel pellet meltdown. This OPM model was generated postmortem, so it focuses on the physical failure that caused the meltdown. Modeling a system with as many as possible failure modes and undesired states, along with possible ways to avoid them, such as adding more

sensors and polling them where it is known that the actual state is critical, is bound to yield an OPM model that is much more complex than the current one. This is where the ability to query the OPM model becomes of paramount importance, as it helps focusing on analysis of possible failures and the contribution of additional components to the mitigation of those failures, ensuring that the system is made more reliable and robust.

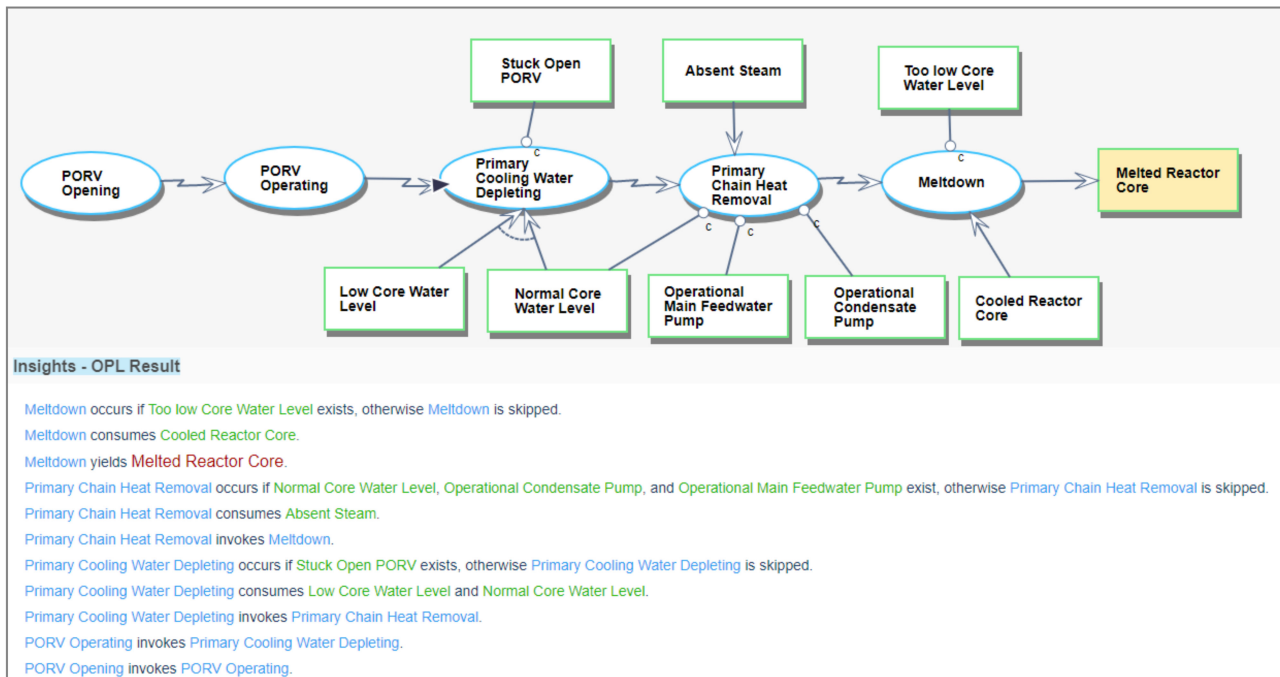


Figure 10. Neighborhood OPM query result of the fault-aware **Steam Generating** model.

Classical shortest-path algorithms [39], including the class of travelling salesman problems, commonly used for network routing problems, are challenging classical combinatorial optimization problems [40,41]. A shortest-path query on conceptual models can reveal the path that requires the minimal effort or cost, for example by following weighted edges that contribute the least to the total of the objective parameter. Yet, conceptual modeling is usually done for purposes of communication, documentation, and visualization of the system architecture, function, structure, and behavior, while modeling for the sake of gaining insights is rare.

5.2. Business Process Optimization Using Path Finding: The Case of the IT Support Workflow

To explore and demonstrate possible path finding capabilities in an OPM model, we generated a workflow OPM model of handling issues by an internal Information Technology (IT) {XE “IT:Information Technology”} support organization and its customers within a large enterprise. In large enterprises, IT support is a service organization in and of itself, often comprised of multiple units. These are sub-organizations staffed by professionals with unique support skill sets, such as experts in a High-Performance Computing environment for design customers, networking solutions for office and data centers, notebooks, and enterprise applications.

To get an IT support service, the customer (an enterprise employee) must generate an IT service ticket, which should be addressed to the person with the relevant support skill. The ticket lifecycle includes one or more support levels that are involved in the ticket resolution, where Level 1 and Level 2 are basic support units, often located in emerging markets. It is expected that most of the tickets will be resolved at these levels. Level 3 support teams are usually engineering teams, which, in many cases, are co-located with

the customers. To reduce Level 3 engineering teams' overhead and context switching, a customer must first go through Level 1 and Level 2 support.

Customers who are not always aware what is the adequate support skill level they need for their current incident or request have difficulty creating a service ticket and prefer direct contact with a Level 3 support team member. Since most customers are located next to Level 3 teams, direct interaction with someone familiar sounds easier than going into the seemingly long route of ticket issuing. They use email or instant messaging methods to ask for the relevant support skill, expecting that the contacted IT person will be able to solve their problem quicker, without having to issue a service ticket. To deal with this situation, the enterprise IT organization created a web application via an "IT Portal", with support categories that are determined based on the customer's issue or request and on the required IT support skill. The requested service is categorized and routed to the relevant support skill. For example, if a customer selected an issue classified as belonging to the Wi-Fi category, the system will generate a ticket and route it to the office networking team. The system diagram (SD), the top-level OPD of the OPM system model in Figure 11, describes the agent **End-User** who **owns** an **Incident** with the initial state **unresolved**. The **Incident Resolving** process changes it to **resolved** using the instrument **Incident Resolving System**.

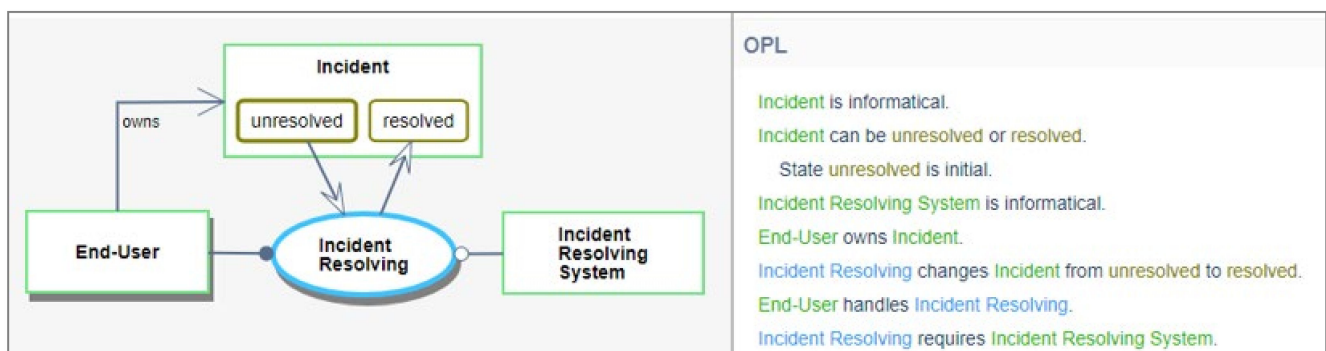


Figure 11. Incident resolving system OPM model—system diagram (SD).

The in-zoomed process **Incident Resolving** in Figure 12 exposes the process **IT Contacting**, which yields **Contact Method** at one of the states **IT Portal** or **IT Personnel**. If the state is **IT Portal**, the **Automatic IT Skill Classifying** process is executed using the instrument **Incident Resolving System**.

If the **Contact Method** is **IT Personnel**, **Manual IT Skill Classifying** is performed. This process is in-zoomed in Figure 13, exposing three subprocesses: **IT Personnel Contacting**, **Issue Reviewing**, and **Required IT Skill Classifying**. Via the **IT Personal Contacting** process, the **End-User** agent changes **Level 3 IT Personnel** from being at state **issue-unaware** to the issue-aware state. The process also invokes **Issue Reviewing**, which changes **Level 3 IT Personnel** from issue-aware to **issue-understood**. The last process, **Required IT Skill Classifying**, changes the stateful object **Required IT Skill** to state **classified**.

Both scenarios lead to a change of the stateful informational object **Required IT Skill** from its initial state **unclassified** to **classified**. This state is connected to the **Incident Ticket Creating** process with an instrument condition link, which triggers **Incident Ticket Creating**. With the help of the instrument **Incident Resolving System**, **Incident Ticket** is created in its initial state **open**. The process **Incident Ticket Resolving**, handled by **Level 1 and 2 Support Agents**, changes the state of the informational object **Incident Ticket** from **open** to **closed** and the **Incident** object from **unresolved** to **resolved**.

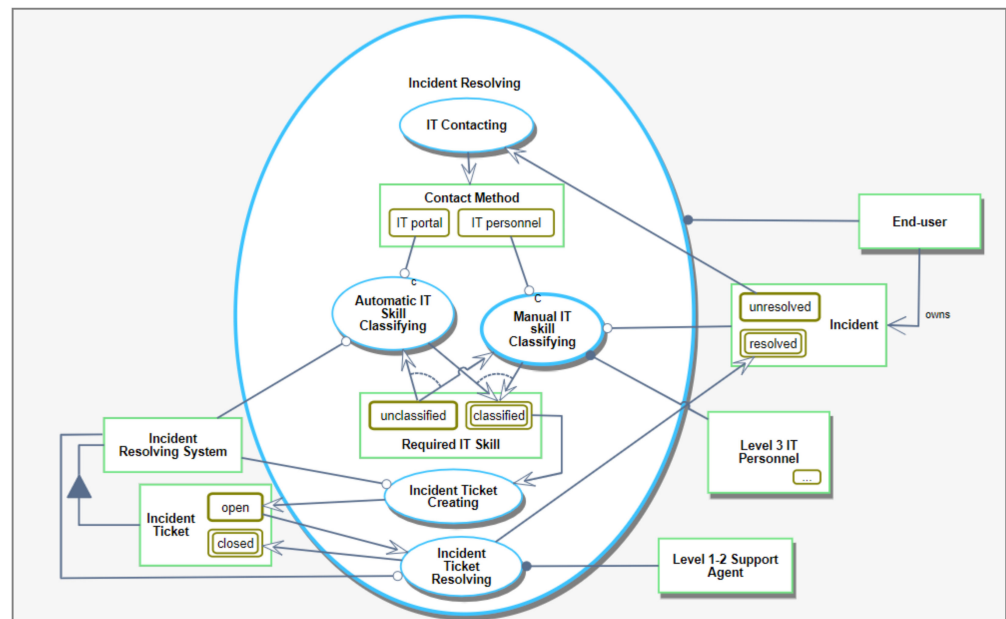


Figure 12. SD1 of the Incident Resolving System model, in which Incident Resolving is in-zoomed.

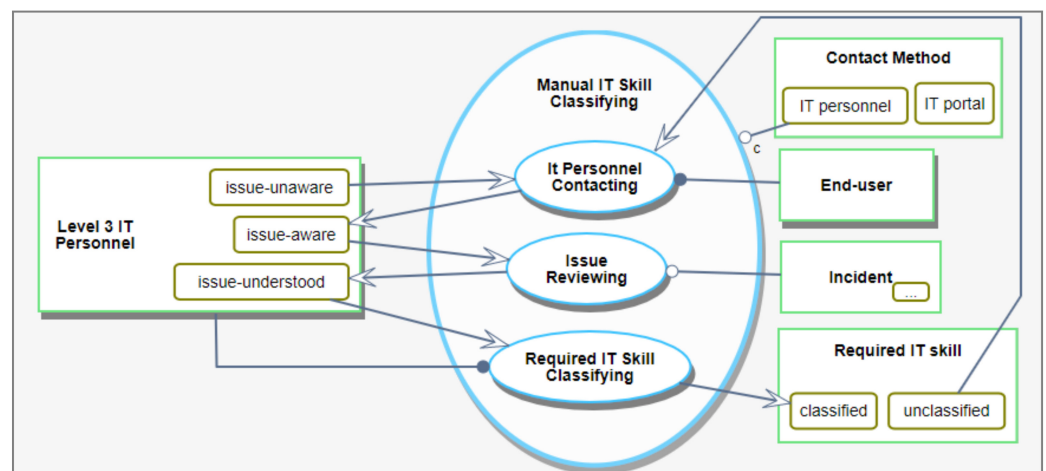


Figure 13. SD1.1 of the Incident Resolving System model, in which Manual IT Skill Classifying is in-zoomed.

Having completed our model design of the system, we can query it. We want to find the shortest path—the path with the minimal number of hops (edges), which involves as few sub-processes as possible and still gets the same result of resolving an incident for the end user. This is also the path that passes through the minimal number of OPM things. In Figure 14, we see the query result of the shortest path query execution as one of the “Path Finding” OPM queries. The path starts from the source **Unresolved Incident** and ends at the target **Resolved Incident**. The result shows that the shortest path passes through **IT Portal Contact Method**. This answer to the query clearly indicates that using the IT portal works better (faster) than contacting an IT staff member, even though customers tend to think intuitively that approaching a staff member directly via email or messaging would be faster.

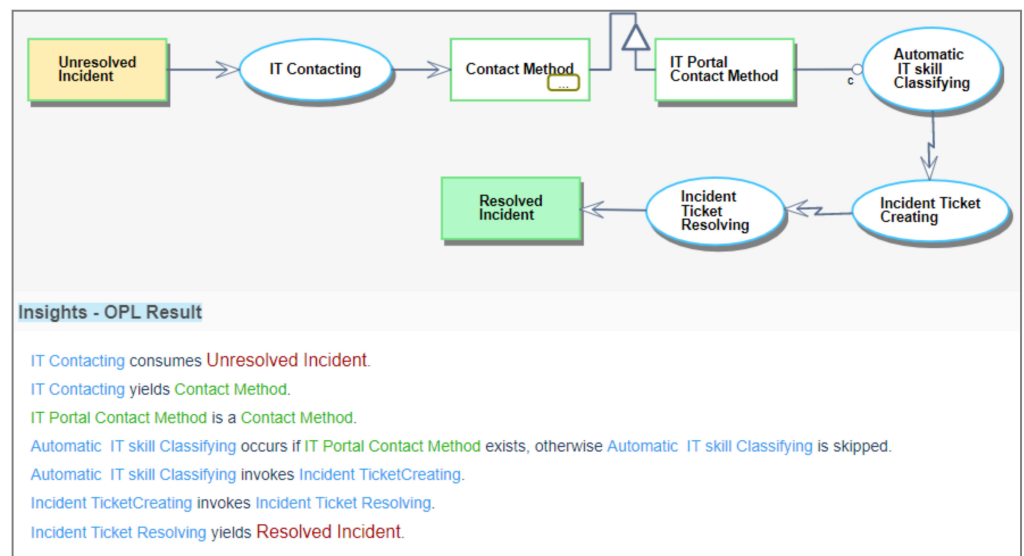


Figure 14. Shortest path OPM query result of the Incident Resolving OPM model.

The longest path query, which is another query in the collection of “Path Finding” OPM queries, generates the result in Figure 15. This solution includes processes and objects that were not part of the shortest path query result. One of these processes is **IT Personnel Contacting**. The insight we can gain from combining both queries is that using the **IT Portal** not only provides better service to the end users, as it contains the minimal number of processes; it also benefits the IT engineering teams who provide Level 3 support, as their productivity can grow thanks to reducing unnecessary overheads and time spent on classifying issues and the distractions caused by unnecessary context switching.

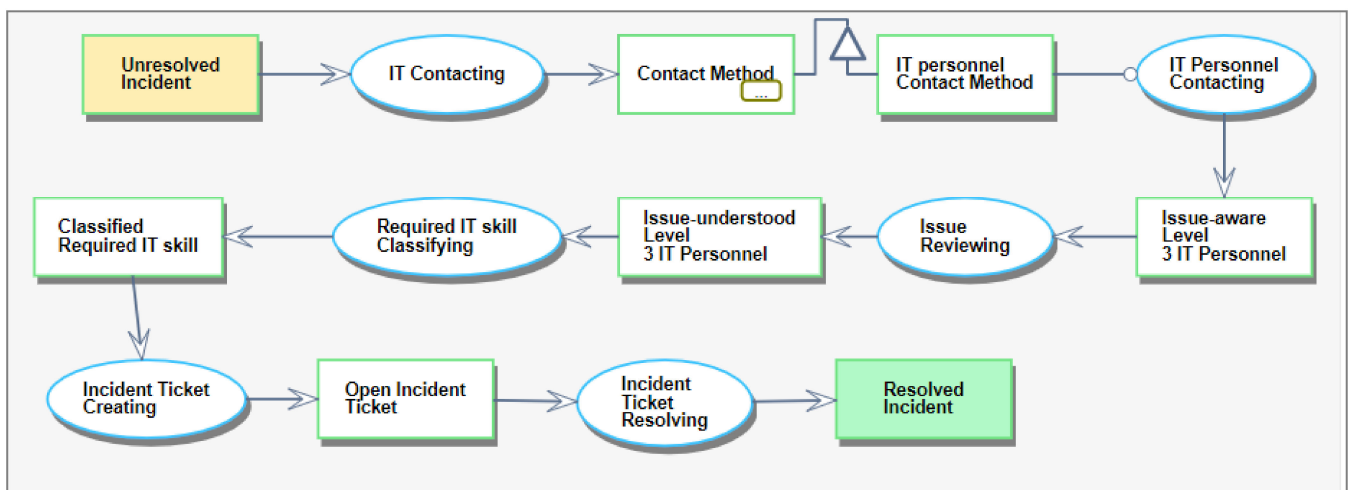


Figure 15. Longest path OPM query result of the Incident Resolving model.

6. OPM Query Implementation

In this section, we discuss the implementation of each of the OPM query collections, including a GDB update approach, web service in OPcloud, path finding query set, neighborhood query set, and centrality query.

6.1. Path Finding Queries Implementation

Researched as far back as the 19th century [42], path finding in graphs is one of the classical graph problems. It gained prominence in the early 1950s in the context of “alternate routing”, i.e., finding a second shortest route when the shortest route gets blocked.

6.1.1. Path Finding UI Implementation

Once the user enters the “Path Finding Queries” section in the “OPM Insights” feature of OPCloud, a dialog pops up, requesting to select a specific path query. Each selection presents a list of properties or parameters to be defined for the query execution. Figure 16 shows an example for the shortest path query, where the display, source, and target nodes need to be specified.

Shortest Path Query

✕

The query will identify the shortest path between two OPM things

Query Selection

Query Type: Shortest Path ▼

Query Properties

Flat Model Visibility Flat Model Visibility ▼

Source Source ▼

Target Target ▼

Run Query

Figure 16. Shortest path query dialog window.

Clicking the “Run Query” button executes a method in the GDB Angular service, which calls additional methods based on the properties specified for that query in the dialog window. A main part of the flattening preprocessing is de-stating. When selecting an OPM state as the source and/or the destination of a query (such as the shortest path), we need to update the GDB Angular service to refer to the corresponding stateless object that represents that state instead of the original state. The query execution progress is displayed dynamically, and when the execution is complete, the dialog window closes automatically. The dialog window includes a checkbox to indicate whether or not to display the flat model in the background. In case of an error, a message in bold red font is displayed.

Figure 17a is an example of executing a “Longest Path” query on an OPM model of the processes (legs) for a flight from Tel Aviv to San Francisco, which can be either direct or with a connection in New York. If the selected flat model visibility option is “Show”, the entire OPM model in the flattened OPD appears in grey, as shown in Figure 17b with the OPM things and links resulting from the query execution highlighted and positioned at the top of the model, while the rest of the elements are positioned below them. The selected source and target things are highlighted by distinct yellow and green colors, respectively. If the selected option of “Flat Model Visibility” is “Hide”, the result of the query, depicted in Figure 17c, will display an OPD with only the relevant OPM things and links, ordered according to the resulting path, positioned in a snake form. In both cases, the Object-Process Language (OPL) paragraph includes only OPL sentences that are part of the query result, and these OPL sentences are ordered according to the resulting route. The query source and target OPM things are highlighted in the OPL by larger font size, different color, and bold style. This enables reading the “story” of traversing the path.

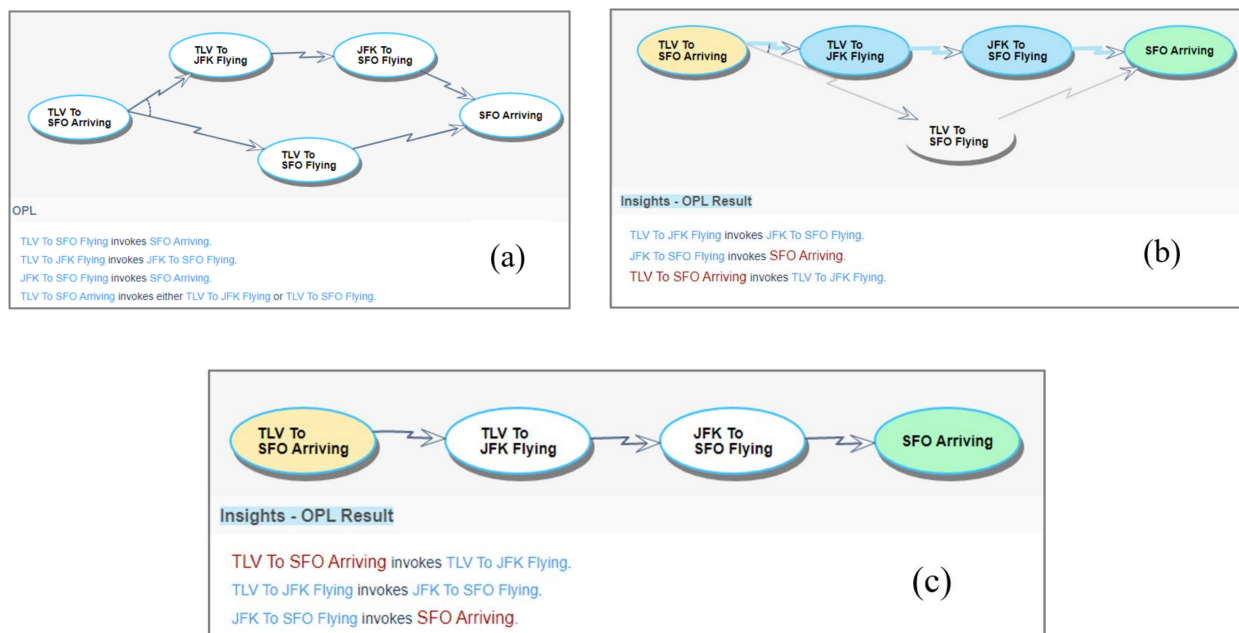


Figure 17. Display view options in a longest path query result: (a) original model; (b) with flat model shown; (c) with flat model hidden.

6.1.2. Shortest Path Implementation

Dijkstra published his shortest path algorithm in 1956 [39] while trying to come up with something to show off the new ARMAC computers. He needed to find a problem and a solution that people not familiar with computing can understand, so he designed what is now known as Dijkstra’s algorithm. There are two options to execute the shortest path in Neo4j. One is the native way, directly, using the Cypher language [43], and the other is to use procedures from Neo4j Labs Graph Algorithms library [44]. Since the library was not developed by Neo4j, it is not officially supported by Neo4j. Therefore, our implementation decision was to use the native Cypher method. Code 3 presents the Cypher query for the shortest path.

Code 3. Cypher query used to get the shortest path result

```

1: MATCH (source: $S_OPM_THING_TYPE {name:'$S_OPM_THING_TEXT'}),
   (target: $T_OPM_THING_TYPE {name:'$T_OPM_THING_TEXT'}),
2: p = shortestpath((source)-[*]->(target))
3: RETURN p

```

The query finds the shortest path between the specified source and target based on the OPM thing type (object or process) and its name (the text of the OPM Thing), assuming the OPM thing names in the model are unique.

6.1.3. Longest Path Implementation

Unlike the shortest path query, Cypher does not have a subroutine for the longest path. To identify the longest path in a graph between two nodes, we generate all the possible paths from the selected source to the selected target, sort them by length in descending order, and pick the top of the list as the longest path.

OPM models are not guaranteed to be acyclic, so in order to avoid loops, a WHERE clause is added to the query to make sure that no path leads back to the source. Code 4 presents the Cypher query used to get the longest path.

Code 4. Cypher query used to get the longest path result

```

1: MATCH p = (source: $S_OPM_THING_TYPE {name:' S_$OPM_THING_TEXT'}) -[*]->
(target: $ T_OPM_THING_TYPE {name:'$ T_OPM_THING_TEXT'})
2: WITH p, RANGE (1, length(p)) AS z
3: WHERE ALL (i IN z WHERE nodes(p)[i] <> source)
4: WITH p
5: ORDER BY length(p) DESC
6: RETURN
7: LIMIT 1

```

6.2. Neighborhood Queries

We developed a set of OPM queries for finding a graph neighborhood—a subgraph which includes a collection of nodes and links that are reachable from a root node in at most a given number of hops (the neighborhood radius or depth). These queries use simple native Cypher queries with an option to filter the link type and limit the maximal depth level. In this section, we present and demonstrate these neighborhood type queries.

6.2.1. Neighborhood UI Implementation

Once the user enters the “Neighborhood Queries” section in the “OPM Insights” feature of OPCloud, the dialog window shown in Figure 18 pops up, requesting the user to choose between a “Non-directional” or a “Directional” query type. Each query type selection generates a list of query properties required for the query execution.

Figure 18. Directional query dialog window.

The “Direction” property is presented only when the “Directional” OPM Query type is selected. The possible options for this query are “Outgoing” and “Incoming”, representing a link direction from or to the root node, respectively. The “Max Depth” property indicates the number of hops from the root node that the graph search query considers. For example, if the selected “Max Depth” value is “1”, the subgraph that the query will return will be the root node and all its neighbors that are directly connected with arcs leading to them. “Only Procedural” is a Boolean property that enables the user to filter the vertices by specifying the OPM procedural links connecting them, which can be one of three kinds [3]:

1. Transforming link, which connects a transformee (an object that the process transforms) or one of its states, with a process to model object transformation, namely generation, consumption, or state change of that object as a result of the process performance;

2. Enabling link, which connects an enabler, i.e., an agent or an instrument, or one of their states, with a process to model an enabling presence for that process; and

3. Control link, which is a transforming or an enabling link with an event or condition control modifier, marked next to the link end by the letter e or c, respectively, that adds semantics of an execution control mechanism to model an event that initiates the linked process or a condition for performing the linked process.

The “Root” property identifies the root (starting) node of the subgraph. The values for this property are generated dynamically, based on the existing OPM model entities, i.e., process, object, or state.

The main process in the OPM model in Figure 19 is **Piano Music Playing**. It is handled by the agent **Piano Player** and results in **Piano Music**. **Piano Music** is connected to the **Music Listening** process, handled by the agent **Audience**. The semantics of the condition link, denoted by the letter c next to the consumption link from **Music Listening**, is that if **Piano Music** exists, it is consumed by **Music Listening**. **Piano Music Playing** requires the instrument **Piano**, which consists of **Keyboard**, **Hammer Set**, **Damper Set**, and **String Set**. The aggregation-participation (whole-part, or “consists-of”) relationship is an OPM fundamental structural link, while the other relationships in the OPM model are procedural—they connect an object and a process.

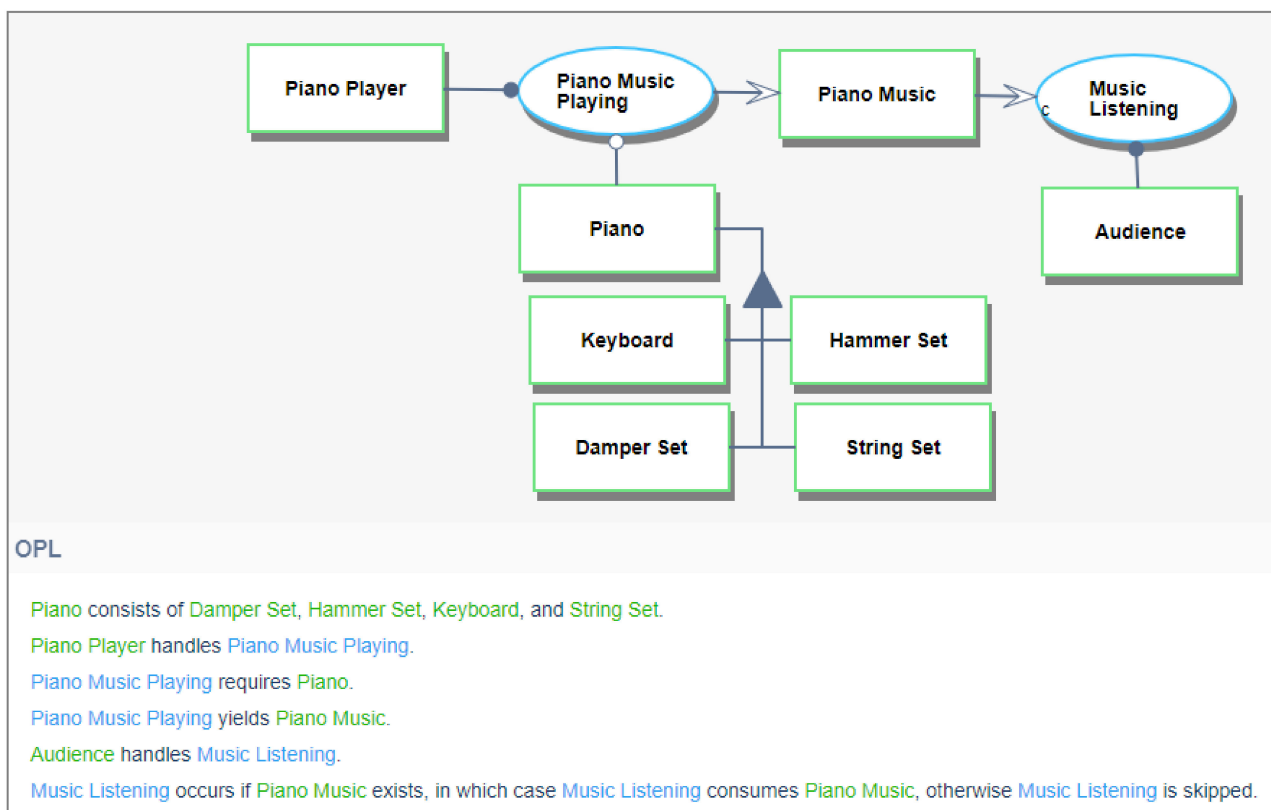


Figure 19. Piano Music Playing OPM model.

Executing a non-directional query starting from the **Piano Music Playing** process with a maximal depth of 3, while filtering for procedural links only, generates the result presented in Figure 20. Due to the procedural links filter, the parts of **Piano** are not included in the result. Since the maximal depth is 3, **Audience** is included in the query result as part of the sub-path {**Piano Music Playing** [HOP1], **Piano Music** [HOP2], **Music Listening** [HOP3], **Audience**}. The root node is highlighted in both the diagram and the OPL sentences.

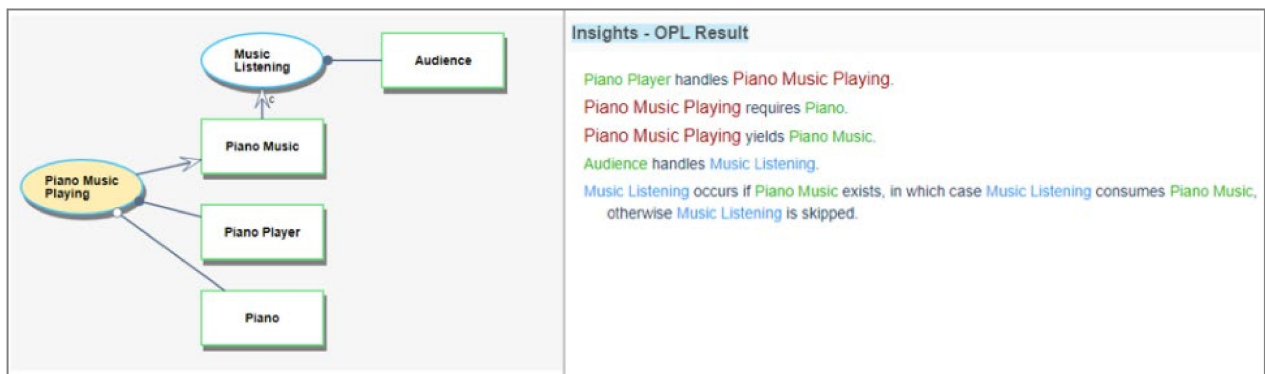


Figure 20. Result of a unidirectional neighborhood query in the **Piano Music Playing** OPM model.

6.2.2. Neighborhood Queries Implementation

All the queries in the neighborhood category are native, noncomplex Cypher queries. The main difference between the three query types in this category, listed in Code 5, Code 6, and Code 7, is the link direction between the start node, which is provided as input from the user, and the other elements in the OPM model. The max depth parameter, also provided by the user, is part of the relationship search configuration.

Code 5. Unidirectional query.

```
1: MATCH p = (source: $ROOT_NODE_TYPE {name:$ROOT_NODE_TEXT'})
  -[*1..$MAX_DEPTH]-(child)
2: RETURN p
```

Code 6. Directional query—outgoing links.

```
1: MATCH p = (source: $ROOT_NODE_TYPE
  {name:$ROOT_NODE_TEXT'})-[*1..$MAX_DEPTH]->(child)
2: RETURN p
```

Code 7. Directional query—incoming links.

```
1: MATCH p = (child) - [*1..$MAX_DEPTH] -> (source: $ROOT_NODE_TYPE
  {name:$ROOT_NODE_TEXT'})
2: RETURN p
```

Code 8 is the conditional statement that is added to the query prior the “RETURN” directive if the user wants to specify that there is a need to filter the OPM procedural links.

Code 8. Conditional statement in neighborhood queries to filter procedural links

```
WHERE ALL (r IN relationships(p) WHERE type(r) = "OpmProceduralRelation")
```

6.3. Centrality Queries

A complete OPM system model contains OPM things (objects and processes), each with its own influence level on other OPM things in the operation system. Identification of the most influential things in the system helps to understand system behavior and predict the behavior of its subsystems. It also helps to identify “hot spots” and critical elements in the system information flow.

Centrality queries is a set of OPM queries that provide a list of influential nodes—a collection of nodes and their direct influence value. The query is based on the *PageRank* algorithm [45,46], which measures the transitive influence or connectivity of nodes. *PageRank* is a variant of the Eigenvector Centrality algorithm [47]. In this section we present and demonstrate these centrality-type queries.

6.3.1. Centrality UI Implementation

Upon entering the “Centrality Queries” section in the “OPM Insights” feature of OPCLoud, a dialog window shown in Figure 21 opens. The “Flat Model Visibility” property in this view has the optional values “Hide”, which will present only the OPM things that have the highest rank, and “Show”, which will show the flat model all grayed out, except for the highest ranking OPM things, which are colored. The “Link Type” property determines whether to calculate the node rank based on procedural links only, fundamental links only, or any type of link. The “Number of Items” property limits the list of ranked nodes in descending order based on the rank level. The “Thing Type” property determines whether to filter out all but processes, all but object and states or no filter at all for the rank consideration.

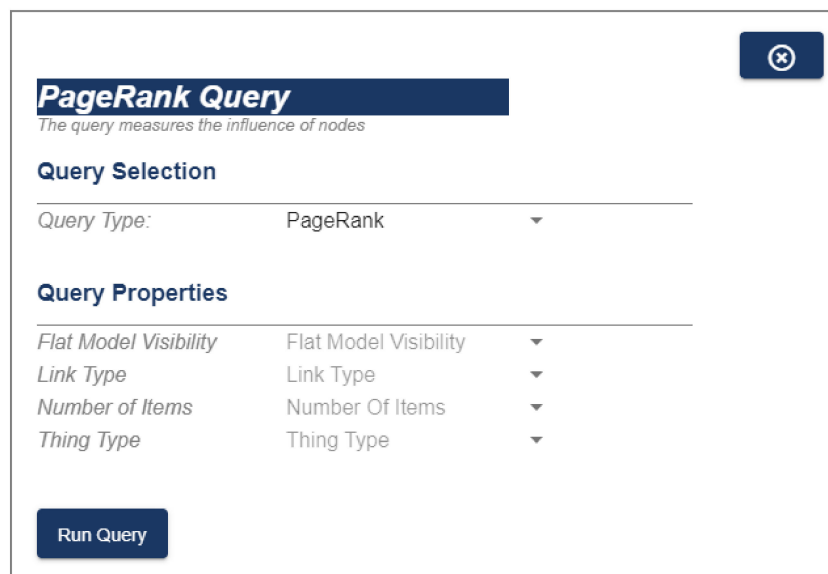


Figure 21. PageRank query dialog window.

Figure 22 is an OPM model in which a **Coffee Making** process, handled by the agent **Barista**, results in the object **Coffee**. **Coffee** is consumed either by **Coffee Taking Away** or by **Coffee Serving**, handled by the agent **Waiter**. Both processes affect the object **Customer**.

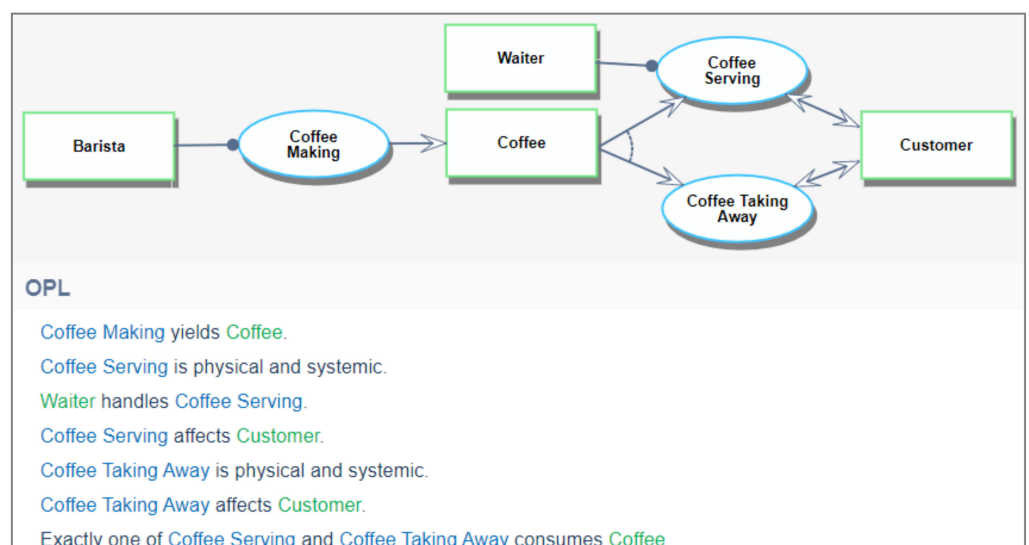


Figure 22. Coffee Making OPM model.

Figure 23a is a graph representation of the OPM model in Figure 22. Each of the highlighted graph nodes, **Coffee Serving** and **Customer**, have the maximum number of incoming links, which is 2. As *PageRank* measures node influence, the number of links is not the only factor in the calculation.

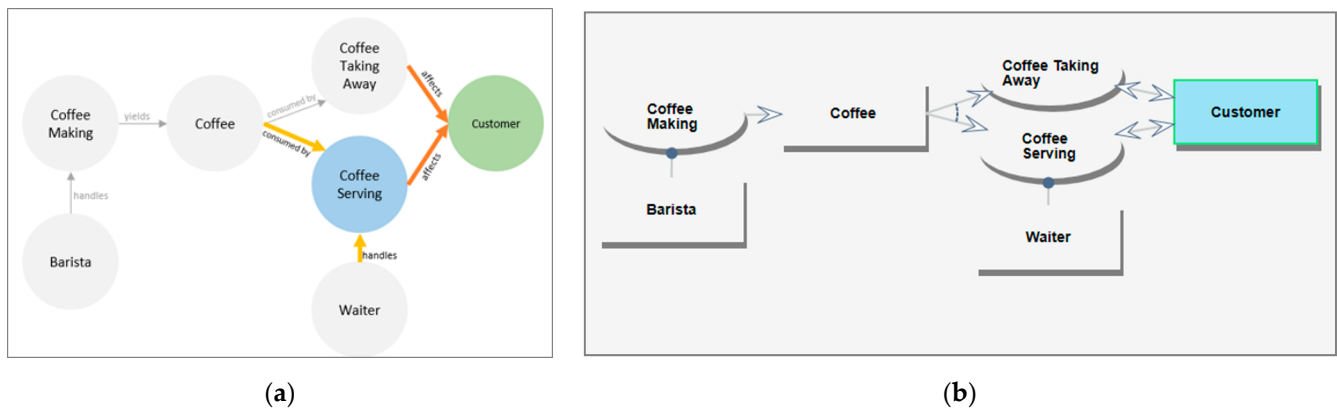


Figure 23. Coffee Providing OPM model: (a) graph representation of the model; (b) *PageRank* OPM query result on the model.

The influence rank of the source nodes contributes to the influence rank of the target nodes. Figure 23b is the *PageRank* query result showing that the most influential OPM thing in this system is the object **Customer**.

6.3.2. *PageRank* Query Implementation

PageRank [45,46] is an algorithm used to rank web pages in Google’s search results, counts the number and quality of links to a web page, and estimates how important the page is. The underlying assumption is that pages of importance are more likely to receive more links than other pages. *PageRank* measures the transitive, or directional, influence of nodes. Other centrality algorithms measure the direct influence of a node, whereas *PageRank* considers the influence of a node’s neighbors and their neighbors. *PageRank*’s formula [46] is

$$PageRank(A) = (1 - d) + d \left(\sum_{i=1}^n \frac{PageRank(T_i)}{C(T_i)} \right),$$

where d is a damping factor, which can be set between 0 and 1—usually set to 0.85—and $C(A)$ is the number of links going out of A . The assumption here is that A has n incoming links T_i .

Unlike other OPM queries, *PageRank* is not only a retrieval query that follows a graph generation query; it requires changing the graph structure for the retrieval query preconditions. While OPM has two types of OPM things, objects, and processes, and two types of OPM links, procedural and fundamental, *PageRank* can be executed only on a single type of nodes and on a single type of links. In case the user needs to execute the *PageRank* on any OPM thing type or any link type, the conversion of the OPM model to a graph has to be modified accordingly. To achieve “any” OPM thing, an overwrite of the OPM thing type (object or process) to a generic value of “THING” is included. Similarly, an overwrite of the OPM link type to a generic value “LINK” is included in case the end user specifies “any” link type. The result of the retrieval query contains the original ids of the OPM things and links, and therefore the result is transformed back to the original OPM thing or link type. Code 9 presents the Cypher query used to calculate and obtain the *PageRank*.

Code 9. Cypher query used to get the *PageRank* result

```
1: CALL algo.pageRank.stream (“$OPM_THING_TYPE“, “$OPM_LINK_TYPE“, {iterations:20})
2: YIELD nodeId, score
3: MATCH (node) WHERE id(node)=nodeId
4: RETURN algo.getNodeById(nodeId),score
5: ORDER BY score DESC
6: LIMIT $NUMBER_OF_ITEMS
```

7. Conclusions and Future Work

The goal of conceptual modeling, the core of model-based systems engineering (MBSE), is to establish a framework that facilitates understanding of the problem space, synthesis of possible solutions, and analyses of identified solutions. Conceptual modeling creates a coherent representation of a system and its operating domain, including interactions with other systems and with the system’s environment [48]. Indeed, MBSE brings much value to systems, with focus on their early lifecycle stages. Yet, when deep analyses that rely on the mathematical representation of the model are required, MBSE lacks rigorous, effective methods. In this work, we show a way to realize the potential of graph models [49] to benefit MBSE. Graph models are configurations of nodes and connections that occur in a plethora of applications. Graphs can represent both diverse physical networks, such as electrical circuits, roadways, or organic molecules, and interactions that might occur in ecosystems, databases, and flow of control in computer programs.

Much research has been done on graph theory applications and graph query optimizations. There is also rich literature on conceptual modeling and its applications in MBSE methods, but research seeking to merge the two fields is scarce. Combining conceptual models with graphs can be highly effective for knowledge representation, as such a combination can provide valuable answers to complex questions and insights into the modeled system. This work is innovative in that it capitalizes on the synergy emanating from the integration of OPM-based conceptual models and graph databases (GDBs), specifically Neo4j.

OPM enables rich formal knowledge representation in both diagrams and text, while Neo4j is based on the formality and robustness of graph theory. Put together, they synergistically make a powerful tool for knowledge management and systems analysis. We demonstrate that a single query or a combination of a few queries on a graph extracted from a conceptual OPM model can provide insights and conclusions that are not obvious or even impossible to draw using just a visual inspection of a non-trivial model. We present examples of querying system models from various disparate domains, including nuclear plants, household, IT, and air safety, showing the value of queries to gain a better understanding of new systems or to perform post-mortem analyses to explain past disasters, such as the Three Mile Island accident, to enhance these systems’ robustness and avoid similar mishaps in the future.

We present an integrated solution whereby the GDB capability is fully immersed in the OPM conceptual modeling tool OPCloud, presenting to the modeler a familiar view of OPM diagrams and the complementary textual OPL sentences in a subset of English or many other natural languages, catering to the dual channel processing multimedia assumption [50] and enhancing the user experience.

In future work, we would like to examine the value of querying as a means to explain and answer questions about the model in the context of a large knowledge base that can be linked with the graph representation of the model, be it general or domain specific. This approach extends querying beyond the limited set of concepts defined by each model in isolation. A common definition of an artificial intelligence (AI) system is a system that is able “to correctly interpret external data, to learn from such data, and to use those learnings to achieve specific goals and tasks through flexible adaptation” [51]. In the spirit of this definition, our goal is to use the querying approach presented in this work as a key component in AI applications. Specifically, we plan to use the graph representation of an

OPM model as a key ingredient in UniKOM—Universal Knowledge Management—an innovative system idea that uses deep learning and language models to convert free natural language text to an OPM model.

Author Contributions: Conceptualization, D.D., D.M., and U.S.; methodology, D.D. and D.M.; software, D.M.; validation, D.M., U.S. and D.D.; complexity analysis, D.M. and U.S.; writing—original draft preparation, D.M.; writing—review and editing, D.D. and U.S.; visualization, D.M.; supervision, D.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Dori, D. *Object-Process Methodology—A Holistic Systems Paradigm*; Springer: Berlin/Heidelberg, Germany, 2002.
2. Dori, D. *Model-Based Systems Engineering with OPM and SysML*; Springer: New York, NY, USA, 2016.
3. ISO. ISO/PAS 19450:2015: Automation systems and integration—Object-Process Methodology. Available online: <https://www.iso.org/obp/ui/#iso:std:iso:pas:19450:ed-1:v1:en> (accessed on 12 January 2021).
4. OPCloud. Available online: <https://opcloud-trial.firebaseio.com/> (accessed on 12 October 2020).
5. Dori, D.; Jbara, A.; Levi, N.; Wengrowicz, N. Object-process methodology, OPM ISO 19450—OPCloud and the evolution. *PPI SyEN* **2018**, *61*, 6–17.
6. Wand, Y.; Weber, R. Research commentary: Information systems and conceptual modeling—A research agenda. *Inf. Syst. Res.* **2002**, *13*, 363–376. [CrossRef]
7. Akehurst, D.H.; Behzad, B. On querying UML data models with OCL. In Proceedings of the International Conference on the Unified Modeling Language, Toronto, ON, Canada, 1–5 October 2001.
8. Kurtev, I. State of the art of QVT: A model transformation language standard. In Proceedings of the International Symposium on Applications of Graph Transformations with Industrial Relevance, Berlin, Germany, 10–12 October 2007.
9. Dirk, B.; Berler, M.; Eastman, J.; Gamerman, S.; Jordan, D.; Springer, A.; Strickland, H.; Wade, D. *The Object Database Standard: ODMG 2.0*; Morgan Kaufmann Publishers: Los Altos, CA, USA, 1997.
10. Chimiak-Opoka, J.; Felderer, M.; Lenz, C.; Lange, C. Querying UML models using OCL and Prolog: A performance study. In Proceedings of the IEEE International Conference on Software Testing Verification and Validation Workshop, Lillehammer, Norway, 9–11 April 2008.
11. Allemang, D.; Hendler, J. RDF-The basis of the Semantic Web. In *Semantic Web for the Working Ontologist*, 2nd ed.; Morgan Kaufmann: Burlington, MA, USA, 2011; pp. 27–50.
12. Pérez, J.; Arenas, M.; Gutierrez, C. Semantics and complexity of SPARQL. In *International Semantic Web Conference*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 30–43.
13. Holt, J.; Perry, S. *SysML for Systems Engineering*; IET: London, UK, 2008.
14. Feldman, S.; Kernschmidt, K.; Vogel-Heuser, B. Combining a SysML-based modeling approach and semantic technologies for analyzing change influences in manufacturing plant models. *Proced. Cirp.* **2014**, *17*, 451–456. [CrossRef]
15. Ameya, N.; Poriya, A.; Dikshay, P. Type of NoSQL databases and its comparison with relational databases. *Int. J. Appl. Infor. Syst.* **2013**, *5*, 16–19.
16. Shim, S.S. The CAP theorem’s growing impact. *Compute* **2012**, *45*, 21–22. [CrossRef]
17. Manoj, V. Comparative study of nosql document, column store databases and evaluation of cassandra. *Int. J. Database Manag. Syst.* **2014**, *6*, 1.
18. Escrava, R.; Wong, B.; Sirer, E.G. HyperDex: A distributed, searchable key-value store. In Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication, Helsinki, Finland, 13–17 August 2012.
19. Abadi, D.; Boncz, P.; Harizopoulos, S.; Idreos, S.; Madden, S. The design and implementation of modern column-oriented database systems. *Foundat. Trends Databases* **2013**, *5*, 197–280. [CrossRef]
20. Vatika, S.; Meenu, D. SQL and NoSQL databases. *Int. J. Adv. Res. Comput. Sci. Softw. Eng.* **2012**, *2*, 8.
21. Alzahrani, H. Evolution of object-oriented database systems. *Glob. J. Comput. Sci. Technol.* **2016**, *16*, 1–4.
22. Wilson, R.J. *Introduction to Graph Theory*; Pearson Education: Delhi, India, 1979.
23. Kaveh, A. Introduction to graph theory and algebraic graph theory. In *Optimal Analysis of Structures by Concepts of Symmetry and Regularity*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 15–35.

24. Romero, J.C.; García-Valdez, M. Using a graph based database to support collaborative interactive evolutionary systems. In *Recent Advances on Hybrid Approaches for Designing Intelligent Systems*; Springer: Cham, Switzerland, 2014; pp. 581–591.
25. Güting, R.H. GraphDB: Modeling and querying graphs in databases. In Proceedings of the 20th International Conference on Very Large Data Bases, Santiago de Chile, Chile, 12–15 September 1994.
26. Rodriguez, M.A. The gremlin graph traversal machine and language (invited talk). In Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, 25–30 October 2015.
27. Holzschuher, F.; René, P. Performance of graph query languages: Comparison of cypher, gremlin and native access in Neo4j. In Proceedings of the Joint EDBT/ICDT 2013 Workshops, Genoa, Italy, 18–23 March 2013.
28. Neo4j. Neo4j Graph Database. Available online: <https://neo4j.com/developer/graph-database/> (accessed on 12 January 2021).
29. Baton, J.; Van Bruggen, R. *Learning Neo4j 3. x: Effective Data Modeling, Performance Tuning and Data Visualization Techniques in Neo4j*; Packt Publishing Ltd: Birmingham, UK, 2017.
30. DB-Engines Ranking. Available online: <https://db-engines.com/en/ranking/graph+dbms> (accessed on 12 December 2020).
31. Neo4j. ACID VS Base Consistency. Available online: <https://neo4j.com/blog/acid-vs-base-consistency-models-explained/> (accessed on 12 January 2021).
32. Fielding, R.T.; Taylor, R.N. Principled design of the modern Web architecture. *TOIT* **2002**, *2*, 115–150. [CrossRef]
33. Wikipedia. Neo4j. Available online: <https://en.wikipedia.org/wiki/Neo4j> (accessed on 12 January 2021).
34. Neo4j. Neo4j Cypher. Available online: <https://neo4j.com/developer/cypher/> (accessed on 12 January 2021).
35. JointJS. Rappid Joint. Available online: <https://resources.jointjs.com/docs/rappid/v3.1/index.html> (accessed on 12 January 2021).
36. openCypher. What Is OpenCypher? Available online: <http://www.opencypher.org/> (accessed on 12 January 2021).
37. Cyganiak, R.; Wood, D.; Lanthaler, M. RDF 1.1 concepts and abstract syntax. In *W3C Recommendation*; W3C: Beijing, China, 2014.
38. Mordecai, Y.; Dori, D. Minding the cyber-physical gap: Model-based analysis and mitigation of systemic perception-induced failure. *Sensors* **2017**, *17*, 1644. [CrossRef]
39. Dreyfus, S.E. An appraisal of some shortest-path algorithms. *Oper. Res.* **1969**, *17*, 395–412. [CrossRef]
40. Adewole, A.; Otubamowo, K.; Egunjobi, T. A comparative study of simulated annealing and genetic algorithm for solving the travelling salesman problem. *IJAIS* **2012**, *4*, 6–12.
41. Sahib, S.J.; Saraswat, P.; Singh, K.; Sharma, J.; Majumdar, R.; Chowdhary, S. Travelling salesman problem optimization using genetic algorithm. In Proceedings of the Amity International Conference on Artificial Intelligence (AICAI), Dubai, UAE, 4–6 February 2019.
42. Berge, C. *The Theory of Graphs and Their Applications*; Wiley: Hoboken, NJ, USA, 1962.
43. Neo4j. [Cypher Manual] Shortest Path Planning—Version 3.5. Available online: <https://neo4j.com/docs/cypher-manual/current/execution-plans/shortestpath-planning/> (accessed on 3 December 2019).
44. Neo4j Labs. [Graph Algorithms] The Shortest Path Algorithm. Available online: <https://neo4j.com/docs/graph-algorithms/current/labs-algorithms/shortest-path/> (accessed on 3 December 2019).
45. Neo4j. Neo4j PageRank. Available online: <https://neo4j.com/docs/graph-algorithms/current/algorithms/page-rank/> (accessed on 4 January 2020).
46. Langville, A.N.; Meyer, C.D. Deeper inside PageRank. *Internet Math.* **2004**, *1*, 335–380. [CrossRef]
47. Bonacich, P. Power and centrality: A family of measures. *Am. J. Sociol.* **1987**, *92*, 1170–1182. [CrossRef]
48. Topper, S.J.; Nathaniel, H.C. Model-based systems engineering in support of complex systems development. *Johns Hopkins APL Techn. Digest* **2013**, *32*, 419–432.
49. Gross, J.L.; Yellen, J. *Graph Theory and Its Applications*; CRC Press: Boca Raton, FL, USA, 2005.
50. Mayer, R.E. *Multimedia Learning Third Edition*; Cambridge University Press: Cambridge, UK, 2021.
51. Kaplan, A.M.; Haenlein, M. Siri, Siri, in my hand: Who’s the fairest in the land? On the interpretations, illustrations, and implications of artificial intelligence. *Bus. Horiz.* **2019**, *62*, 15–25. [CrossRef]