The analysis approach of ThreatGet

(version 21.04)

Korbinian Christl
Korbinian.Christl@ait.ac.at
Thorsten Tarrach
Thorsten.Tarrach@ait.ac.at

July 21, 2021

AIT Austrian Institute of Technology

Nowadays, almost all electronic devices include a communication interface that allows to interact with them, exchange data, or operate their services remotely. The trend toward increased interconnectivity simultaneously increases the vulnerability of these systems. Due to the high costs associated with comprehensive security analysis, many manufacturers neglect the safety aspect of a product in order to avoid costs. However, the importance of secure IT systems is growing, as the security of a system can also influence safety-critical aspects. Standard security analysis approaches are nowadays still mainly based on time-intensive and error-prone manual activities. In this paper, we present the formal concepts of the automatic threat and vulnerability analysis tool ThreatGet. Therefore, we introduce the concept of the Extended Data-Flow Diagram that is used to represent the system under investigation in an abstracted form, and we highlight the formal analysis language of the tool. This domain-specific language is used to formulate so-called anti-patterns. These anti-patterns that can be interpreted by the tool for an automatic security analysis of the system. Besides the language declaration, we present the entire semantic evaluation of the language during the analysis. Parts of the definitions and elaborations of the diagram model and the analysis language were developed in the context of the master thesis of Korbinian Christl, in cooperation with the University of Vienna.

Contents

1 Introduction

Over the past two decades, IT-systems, their areas of application and their purposes have significantly evolved. As a result, these systems have become increasingly complex and now offer cross-domain interfaces to exchange data and services.
This development has meant that everyday devices such as coffee machines and entire production lines in factories can be controlled and monitored from remote. Moreover, this development process is far from complete, as the efforts toward the autonomous driving show.

Besides the positive aspects that this development has brought, negative factors have also emerged. Due to the rapid development of new products and the competition on the market, manufacturers often omit security and safety audits of their products to save money (Gilchrist, 2017). However, as the number of electronic devices and their communication interfaces grows, the vulnerability of the systems increases (Jang-Jaccard & Nepal, 2014). Attacks by the "Mirai" botnet and the rise of malicious ransomware show the impact that insecure electronic devices can have (Mos & Chowdhury, 2020). As the automotive industry moves toward autonomous driving, the importance of vehicle security becomes even more apparent, as security deficiencies in this area can compromise passenger safety. This concern has been impressively demonstrated by Miller and Valasek (2015) in their paper on the "Jeep Cherokee" hack (Miller & Valasek, 2015).
With increasing customer awareness for security issues and rising interest in data privacy and security, manufacturers face the challenge of making their products not only smarter, but also safer and more secure.

The central issue in developing safe and secure products is that most security analysis methods are still predominantly based on error-prone and time-consuming manual activities. Therefore, high costs may be incurred, especially if an analysis has to be repeated due to changes to the system. Moreover, the quality of the results of such an analysis depends heavily on the knowledge and experience of the analyst (Krisper et al., 2020).
Another challenge stems from the fact that publicly available threat and vulnerability databases such as the "National Vulnerability Database" (NVD), "Common Vulnerabilities and Exposures" (CVE), "Common Vulnerabilities Enumeration" (CWE), and the "Common Attack Pattern Enumeration and Classification" (CAPEC) are enormous collections of entries that are almost daily updated, and no analyst can keep track of all the new entries (NVD, 2020; MITRE, 2019, 2020a, 2020b).
Finally we want to highlight the fact that compliance with security and safety standards such as the ISO/SAE 21434 (ISO/SAE, 2020) will become mandatory in the future. As a result, smart device manufacturers will be required to ensure the security of their products. However, there are probably not enough security experts to cover all areas and domains (Schmittner et al., 2020).
These challenges and circumstances indicate a high demand for an automated, repeatable, and objective solution to identify vulnerabilities and potential threats within a system. In addition, it should be applicable already during the planning- as well as the design-phase of a project to avoid the costs of an retrospective solution (Shostack, 2014).

In order to tackle these challenges, the Dependable Systems Engineering Group (DSE) at the Austrian Institute of Technology (AIT) has developed and maintains the ThreatGet tool. Furthermore, this tool is already commercially distributed in cooperation with LieberLieber and is also available for academic purposes on a test basis.
The concept of ThreatGet is based on two fundamental components comparable to other software/system-oriented analysis tools. Section 3 of discusses these tools and highlights further related research in this area.
On the one hand, the first component is called the *system-model*. It is a graphical representation of the system under study in an abstracted diagram. ThreatGet uses an extended form of the data-flow diagram (DFD), which is also discussed in this paper. Within this diagram, the user defines the structure of the system as well as all security and safety-related aspects (Hussain et al., 2014; Shostack, 2014).
On the other hand, the second component is often referred to as the *threat-model*. The term threat-model can be used to describe any type of knowledge collection that focuses on the definition and retention of threats or vulnerabilities. ThreatGet uses a proprietary analysis language that allows users to define threats in a domain-specific context that both humans and machines can interpret. In this way, ThreatGet provides a viable approach to managing threat intelligence and a sustainable approach to reusing the gathered knowledge.
In section 3, both the concept of the system-model and the threat model are presented and explained in detail. In addition to the formal definitions of the diagram and the language, their application is also explained using examples. Based on the language specification, it is explained in detail how its semantic evaluation is conducted based on the definition of the "Advanced Data-Flow Diagram" (ADFD). These specifications allow users to precisely understand how the expressions formulated in the analysis language are processed, how the analysis itself is conducted, and how the results are obtained.

Parts of this paper, especially Section 2, were written in cooperation with the University of Vienna as part of the master thesis of Korbinian Christl. The content of this chapter has been largely adopted but adjusted accordingly to the modifications of the analysis language.
This paper is a direct extension of the public tool documentation available on the public homepage[1]. Furthermore, it is considered a living document, which will be updated accordingly to the modifications of the analysis language or the diagram model used.

2 The Analysis Approach

The analysis approach of ThreatGet includes three components: an "Advanced Data-Flow Diagram" (ADFD) that serves as a system-model, a rule-based threat-model, and an analysis engine that compares the system-model with the threat-model. This section is divided into three subsections.

The first section outlines several preliminaries and notational conventions used in the following sections for the formal definition of the system-model and the threat-model.

[1] `https://www.threatget.com/`

The second section illustrates the formal definition of the system-model and threat-model. Firstly, the diagram components and the structure of a diagram are presented using a simplified example. This example is used continuously to illustrate the following formal concepts. Building on this, the formal representation of a diagram is presented. In addition, a content-specification is outlined. The content-specification includes all predefined stencils. Therefore, it can be used to validate the content of the defined diagram as well as the anti-patterns. It serves as a bridge between the system-model and the threat-model.

Afterward, the formal definition of the rule-based threat-model using the analysis language is elaborated. Each rule describes a potential threat in the form of an anti-pattern. Threats are defined and stored in a specially defined language, whose syntax and semantics are explained in detail.

2.1 Preliminaries and Notational Conventions

Several formal mathematical definitions are presented throughout the following sections. These definitions use the preliminaries and notational conventions presented in this section.

- A relation R over a set A is *transitive* if $\forall a, b, c \in A,\ (a, b) \in R \land (b, c) \in R \implies (a, c) \in R$ (Weisstein, 2021d).
- A relation R over a set A is *irreflexive* if $\forall a \in A, (a, a) \notin R$ (Weisstein, 2021b).
- A relation R over a set A is *antisymmetric* if $\forall a, b \in A, (a, b) \in R \implies (b, a) \notin R$ (Weisstein, 2021a).
- A function f is a *total* function if and only if it maps each value of the domain $(x \in X)$ to a value of the co-domain Y, denoted as $f : X \to Y$ (Black, 2007).
- A function $f : X \to Y$ is a *partial* function if f maps only values from a proper subset S of X to Y, denoted as $f : X \nrightarrow Y$ (Black, 2019).
- The result of a partial function $f : X \nrightarrow Y$ is *undefined* for a parameter $x \in X$ if the function does not provide any value $y \in Y$, such that $f(x) =$ *undefined* .
- The result of a partial function $f : X \nrightarrow Y$ is *defined* for a parameter $x \in X$ if the function does provide a value $y \in Y$, such that $f(x) = y$.
- An empty function f for a domain X always returns *undefined* for any parameter $x \in X$. $f(x) =$ *undefined*.
- A sequence seq is defined as an ordered set of entries e where the position of each entry is indicated by a number $seq(e_1 \dots e_i)$

- The *powerset* $\mathfrak{P}^A$ is defined as a set including subsets of the a set A (Weisstein, 2021c).

- Let f be a partial function. By $f[a := b]$ it is denoted that the function returns the value b for argument a and $f(x)$ for all other arguments x.

Table 1 contains the most important notational conventions used throughout the following sections. Generally, sets are labeled with uppercase letters, and their contained entries are labeled with lowercase letters.
The exact definition of the individual table entries is explained in the corresponding sections.

Concept	**Instance notation**	**Set notation**
Element identifiers	n	$\mathcal{N}$
Element types	l	$\mathcal{L}$
Asset identifiers	y	$\mathcal{Y}$
Asset types	z	$\mathcal{Z}$
Boundary identifiers	a	$\mathcal{A}$
Boundary types	b	$\mathcal{B}$
Connector identifiers	r	$\mathcal{R}$
Connector types	t	$\mathcal{T}$
Property keys	k	$\mathcal{K}$
Property values	v	$\mathcal{V}$
Flow	p	-
Sequence	$seq(\ldots)$	-
Functions	f, f_1	-

Table 1: Notational conventions

2.2 Model Concepts

This section presents the formal concepts of the system-model and the threat-model. Firstly, the "Advanced Data-Flow Diagram" (ADFD) and its components are discussed and illustrated using an example. Secondly, the content and structure of the diagram are formally defined and discussed. Thirdly, the definition of the rule-based threat-model and its language syntax is explained.

2.2.1 Diagram Components and Modeling Notation

A standard data-flow diagram does not offer enough semantic depth to meet the requirements of a comprehensive security analysis. This section presents an advanced version of the data-flow diagram (ADFD). Therefore, the individual diagram components, their modeling conventions, and their intended use is described in the following listing:

- **Element:** A standard data-flow diagram uses different diagram components to differentiate between external actors, processes, and data stores. However, the ADFD defines a single diagram component called "Element" with an assignable type. A type is predefined as a stencil (template) and describes the nature of an element, and contains its properties. This type-based approach allows both physical and logical system elements to be modeled and can be expanded as required. In addition, elements can be nested within each other. This hierarchical structure allows an additional semantic relationship besides the connectors (data-flows).

- **Connector:** A connector indicates an interaction between two diagram elements. A connector can be used to represent logical as well as physical data-flows. Connectors are always directed between a source and a target element. Similar to the elements, connectors are further specified by their assigned type.

- **Asset:** Assets describe logical or physical objects of value. An asset can be used to represent system-critical software or the movement of confidential data in a system. Assets are further specified by a predefined type.

- **Asset Connector:** In contrast to the previous connector, the asset connector is not used to display the relationships between two elements but rather to display the relationship between an asset and an element or connector. An asset connector is undirected, and an asset can be linked to several different elements and connectors. It is the only diagram component that cannot have an assigned type as it only represents the affiliation of an asset to an element or connector.

- **Boundary:** A boundary describes a separation between logically, physically, or legally separated system elements. A boundary has an assigned type, but a has no properties. Boundaries are located at the lowest hierarchical level of a diagram and can only be contained by other boundaries.

Properties are also predefined as stencils in the form of key-value pairs. Each property stencil consists of a unique key and multiple values where each value must be unique for the key. Boundary types have no assigned properties as they are not a real part of the system.
Figure 1 illustrates the graphical notation of the individual components.

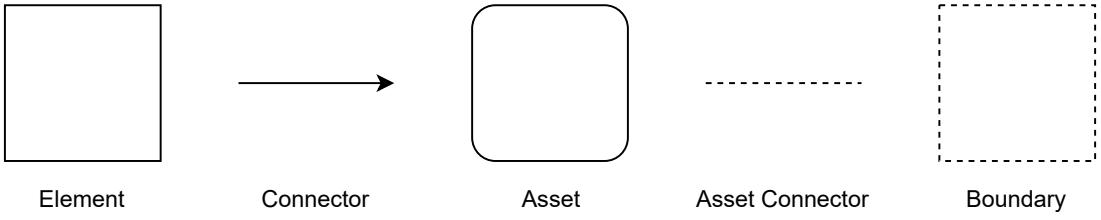

Figure 1: Graphical notation of the individual diagram components

2.2.2 Diagram Example

The previous section introduced the individual diagram components. This section demonstrates their intended use with a fictitious simplified example. The diagram shown in Figure 2 describes the request of confidential user data by a mobile phone user.

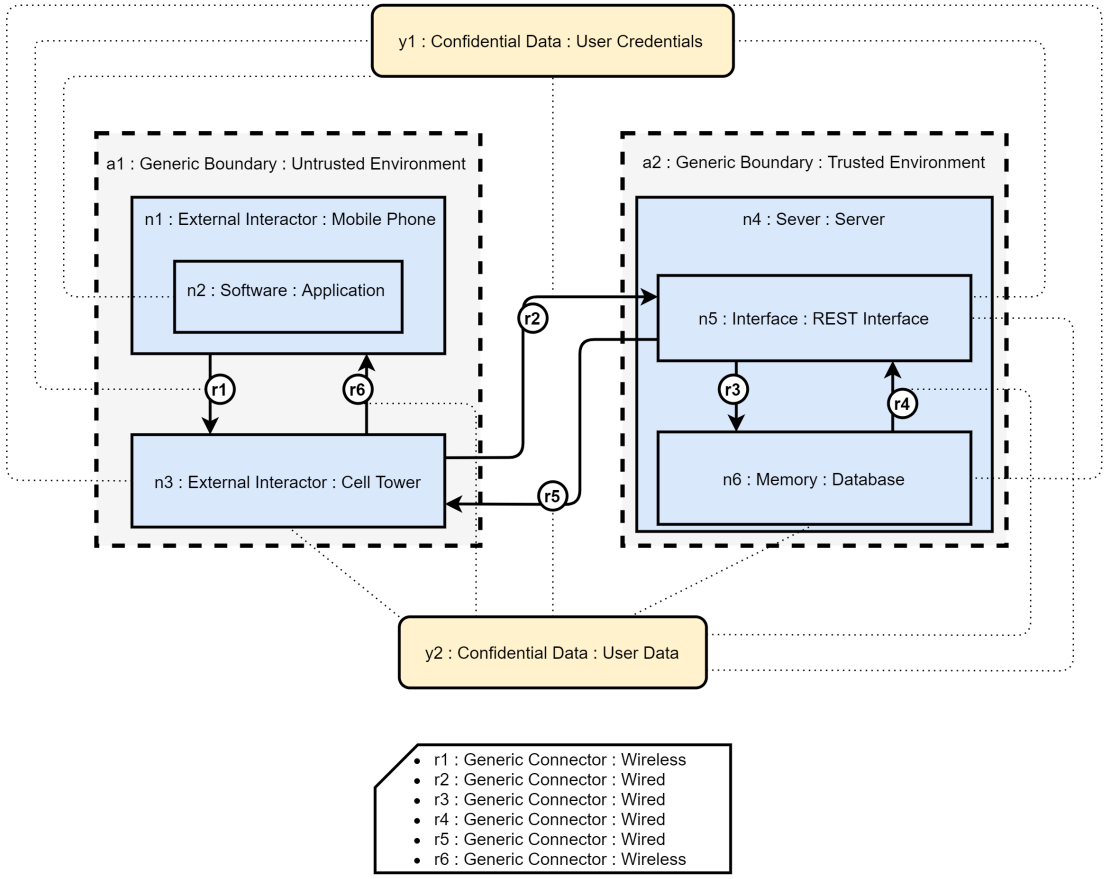

Figure 2: Mobile data request example diagram

The diagram describes the following sequence of events: A user requests data from an offered service by entering the credentials into the mobile device application. The credentials are sent to the nearest cell tower via a wireless connection. From this point, the data is forwarded to the specific server, which in turn validates the provided data and, if it is correct, sends the requested data back the same way.
All elements, connectors, boundaries, and assets are annotated according to the same scheme. This scheme has the following structure :
<Identifier>: <Top-Type><Sub-Type>.

Each diagram component has a unique identifier, which is used to distinguish the individual components in addition to the assigned types. Identifier and types are separated by a colon. The

"Top-Type" represents a generic group of more specific "Sub-Types". For example: Both assets within the diagram are of the "Top-Type: Confidential Data". However, the user enter the "User Credentials" and receives the "User Data". Both are "Sub-Types" of the "Top-Type: "Confidential Data".
The left side of the diagram shows the "Mobile Phone" (n1), the used "Application" (n2), and the closest "Cell Tower" (n3). These three elements are independent of the service offered on the right-hand side and are therefore in an "Untrusted Environment" boundary (a1). As described above, the user must first provide her login data. The credential information is shown within the diagram as an asset called "User Credentials" (y1). The asset component is linked to the connectors that transport the data asset and the elements that process the data. The link is visualized using asset connectors. The requested service is shown on the right side of the diagram. It consists of a "Server" (n4) which contains a "REST Interface" (n5) and a "Database" (n6). The "REST Interface" receives the user credentials, queries the requested data from the database, and sends it back. This data is also modeled as an asset named "User Data" (y2).

Identifier	**Type**	**Property Key**	**Value**
n1	Mobile Phone	OS	Android
n1	Mobile Phone	Vendor	Third Party
n2	Application	Vendor	Third Party
n4	Server	Vendor	Third Party
n5	REST Interface	Input Validation	Yes
n5	REST Interface	Input Sanitization	No
n6	Database	Encrypted	Yes
r1	Wireless	Protocol	HTTP
r2	Wired	Protocol	HTTP
r3	Wired	Protocol	HTTP
r4	Wired	Protocol	HTTP
r5	Wired	Protocol	HTTP
r6	Wireless	Protocol	HTTP
y1	User Credentials	Encrypted	No
y2	User Data	Encrypted	Yes

Table 2: Example Diagram assigned Properties

Table 2 displays the assigned properties of the individual diagram components. For example, the mobile phone operating system is Android, and the communication protocol of the connectors is HTTP.
In this case, only the confidential user data is encrypted but not the user credential asset. This condition could lead to the following threat: An attacker could intercept the credential data during the wireless transmission to the cell tower.

Moreover, the attacker could use them to impersonate the user and spoof the offered service to obtain the confidential data of the user. Subsequently, the attacker may be able to crack the

encryption and gain access to the data.
This example is intended to illustrate how the information of the system can be represented in a diagram and how it can be used to uncover potential threats.
The two following sections present the formal representation of the content-specification and the advanced data-flow diagram.

2.2.3 Formal Content-Specification

A modeled diagram consists of elements, connectors, assets, and boundaries. Each of these diagram components has an assigned type. The diagram component types and properties are predefined in stencils. The predefined stencils form the so-called content-specification. Its type determines the properties that can be set on an element, connector, or asset. The specification tells which properties are allowed for each type.

Furthermore, the types of the diagram components can be assigned in a two-level hierarchy. This means that the diagram components each have an top-type and a sub-type. To illustrate this, consider the previous diagram example. The "Mobile Phone" element contains an element called "Application". "Application" is a sub-type of the top type "Software". The top type "Software" contains several sub-types, for example: "Application", "Operating System" and "Firmware". Sub-types can be used to specify diagram components more precisely. Previously it was pointed out that the properties are assigned to the respective types. Sub-types always contain all properties of their top types, but can also have additional properties.
As stated before: the specification can be seen as the bridge between the system-model and the threat-model. A valid diagram can only display content that is defined in the specification, and an anti-pattern can only query what can be modeled. Section 2.2.7 presents the validation of a diagram using this specification and Section 2.2.11 the validation of an anti-pattern.

The specification S is defined as a tuple $S = \langle \mathcal{L}, \mathcal{Z}, \mathcal{B}, \mathcal{T}, \mathcal{K}, \mathcal{V}, \mathcal{H}, \mathcal{E}, \iota_C, \eta, \gamma \rangle$ where:

- $\mathcal{L}$ is a finite set of element types.
  All elements in a diagram can only have an assigned type $l \in \mathcal{L}$.

- $\mathcal{Z}$ is a finite set of asset types.
  All assets in a diagram can only have an assigned type $z \in \mathcal{Z}$.

- $\mathcal{B}$ is a finite set of boundary types.
  All boundaries in a diagram can only have an assigned type $b \in \mathcal{B}$.

- $\mathcal{T}$ is a finite set of element connector types.
  All connectors in a diagram can only have an assigned type $t \in \mathcal{T}$.

- $\mathcal{K}$ is a finite set of property keys.
  Elements, connectors, and assets in a diagram can have properties. $\mathcal{K}$ contains the predefined property keys.

- $\mathcal{V}$ is a finite set of property values.
  Each property key has a set of possible values. $\mathcal{V}$ contains all possible values that can be assigned to various property keys.

- $\iota_{\mathcal{L}} : \mathcal{L} \to \mathfrak{P}^{\mathcal{L}}$ assigns each "Top-Level" element component type its "Sub-Level" element component types.
  The function ι_C has an index C, which specifies the considered set of diagram component types. In this case only the element component types are considered. Moreover, the function $\iota_{\mathcal{L}}$ is *undefined* for each "Sub-Level" element component type since there are only two levels of hierarchy, mathematically expressed:
  $\forall l \in \mathcal{L}, \big((\exists l' \in \mathcal{L}, l \in \iota_{\mathcal{L}}(l')) \implies \iota_{\mathcal{L}}(l) = \textit{undefined}\big)$

- $\iota_{\mathcal{Z}} : \mathcal{Z} \to \mathfrak{P}^{\mathcal{Z}}$ assigns each "Top-Level" asset component type its "Sub-Level" asset component types. The definition of this function is similar to $\iota_{\mathcal{L}}$.

- $\iota_{\mathcal{B}} : \mathcal{B} \to \mathfrak{P}^{\mathcal{B}}$ assigns each "Top-Level" boundary component type its "Sub-Level" boundary component types. The definition of this function is similar to $\iota_{\mathcal{L}}$.

- $\iota_{\mathcal{T}} : \mathcal{T} \to \mathfrak{P}^{\mathcal{T}}$ assigns each "Top-Level" connector component type its "Sub-Level" connector component types. The definition of this function is similar to $\iota_{\mathcal{L}}$.

- η: $(\mathcal{L} \cup \mathcal{Z} \cup \mathcal{T}) \to \mathfrak{P}^{\mathcal{K}}$ assigns each element, asset or connector type a set of property keys.

- γ: $\mathcal{K} \to \mathfrak{P}^{\mathcal{V}}$ assigns each property key a set of possible values.

2.2.4 Example Content-Specification

This section shows a content-specification to which the example diagram from Section 2.2.2 corresponds. Normally, the order of the steps is reversed because the content-specification must already be defined before a diagram can be created. The specification for the example from Figure 2 is presented below:
$S = \langle \mathcal{L}, \mathcal{Z}, \mathcal{B}, \mathcal{T}, \mathcal{K}, \mathcal{V}, \mathcal{H}, \mathcal{E}, \iota_C, \eta, \gamma \rangle$

- $\mathcal{L} = \{$External Interactor, Mobile Phone, Software, Application, Cell Tower, Server, Interface, REST Interface, Memory, Database$\}$
  Contains the available element types.

- $\mathcal{Z} = \{$Confidential Data, User Credentials, User Data$\}$
  Contains the available asset types.

- $\mathcal{B} = \{$Generic Boundary, Untrusted Environment, Trusted Environment$\}$
  Contains the available boundary types.

- $\mathcal{T} = \{$Generic Connector, Wired, Wireless$\}$
  Contains the available connector types.

- $\mathcal{K}$ = {OS, Vendor, Input Validation, Input Sanitization,
  Encrypted, Protocol }
  Contains the available property keys for elements as well as for connectors.

- $\mathcal{V} = \{$ Unknown, Yes, No, HTTP, HTTPS, Third Party,
  Own Premise, Android, IOS$\}$
  Contains the available values which can be assigned to property keys.

- $\iota_{\mathcal{L}}$(External Interactor) = {Mobile Phone, Cell Tower }
  $\iota_{\mathcal{L}}$(Software) = {Application}
  $\iota_{\mathcal{L}}$(Server) = {}
  $\iota_{\mathcal{L}}$(Memory) = {Memory}
  $\iota_{\mathcal{L}}$(Interface) = {REST Interface}
  The partial function $\iota_{\mathcal{L}}$ describes the hierarchical levels of element components types $\mathcal{L}$. For the example it is defined that the element type "Software" is the "Top-Type" of the "Sub-Type": "Application". The "Top-Type" "Sensor" has no assigned "Sub-Types", but it is defined as a "Top-Type". Therefore it returns not *undefined* but the $\emptyset$.

- $\iota_{\mathcal{Z}}$(Confidential Data) = {User Credentials, User Data}
  The partial function $\iota_{\mathcal{Z}}$ describes the hierarchical levels of asset components types $\mathcal{Z}$. For the example it is defined that the asset type "Confidential Data" is the "Top-Type" of the "Sub-Types": "User Credentials" and "User Data".

- $\iota_{\mathcal{B}}$(Generic Boundary) = {Untrusted Environment, Trusted Environment}
  The partial function $\iota_{\mathcal{B}}$ describes the hierarchical levels of boundary components types $\mathcal{B}$.

- $\iota_{\mathcal{T}}$(Generic Connector) = {Wired, Wireless}
  The partial function $\iota_{\mathcal{T}}$ describes the hierarchical levels of connector components types $\mathcal{B}$.

- η(Mobile Phone) = {OS}, η(Mobile Phone) = {Vendor},
  η(Application) = {Vendor},
  η(Server) = {Vendor},
  η(REST Interface) = {Input Validation},
  η(REST Interface) = {Input Sanitization},
  η(Database) = {Encrypted},
  η(Wired) = {Protocol},
  η(Wireless) = {Protocol},
  η(User Credentials) = {Encrypted},
  η(Confidential User Data) = {Encrypted}
  The function η assigns each element, asset or connector type their available property keys.

For example each instance of an element with the assigned type "Database" has the property "Encrypted".

- $\gamma(\text{OS}) = \{\text{Unknown, Android, IOS}\}$,
  $\gamma(\text{Vendor}) = \{\text{Unknown, Third party, Own Premise}\}$,
  $\gamma(\text{Protocol}) = \{\text{Unknown, HTTP, HTTPS}\}$,
  $\gamma(\text{Encrypted}) = \{\text{Unknown, Yes, No}\}$,
  $\gamma(\text{Input Validation}) = \{\text{Unknown, Yes, No}\}$
  The function γ assigns each property key its available values.
  For example the property "Encrypted" can have exactly one value out of the set "{Unknown, Yes, No}".

2.2.5 Formal Diagram-Definition

Section 2.2.1 discussed the diagram components and Section 2.2.2 demonstrated their intended use. This section contains a formal definition of system-models that were previously only represented graphically. Since this definition does not refer to a specific diagram type, the analysis does support not only diagrams as presented in Section 2.2.2 but also other types of diagrams that can be represented in terms of the definition below.

The definition of a diagram D is defined as a tuple
$D = \langle \mathcal{N}, \mathcal{Y}, \mathcal{A}, \mathcal{R}, source, target, \lambda_C, \mu, \delta, \kappa, \rho \rangle$ where:

- $\mathcal{N}$ represents the elements in a defined diagram D.
  Each element in a diagram D has a unique identifier from $\mathcal{N}$.

- $\mathcal{Y}$ represents the assets in a defined diagram D.
  Each asset inside a diagram D has a unique identifier from $\mathcal{Y}$.

- $\mathcal{A}$ represents the boundaries in a defined diagram D.
  Each boundary inside a diagram D has a unique identifier from $\mathcal{A}$.

- $\mathcal{R}$ represents the connectors in a defined diagram D.
  Each connector in a diagram D has a unique identifier from R.The set R can also be $\emptyset$, as not all diagrams contain a connector.

- $source : \mathcal{R} \rightarrow \mathcal{N}$ is a total function that maps each connector to its *source* element. Each connector in a diagram must always have a starting point.

- $target : \mathcal{R} \rightarrow \mathcal{N}$ is a total function that maps each connector to its *target* element. Each connector in a diagram must always have an endpoint.

- $\lambda_{\mathcal{N}} : \mathcal{N} \rightarrow \mathcal{L}$ is a total function that can map each element identifier to an element type. The function λ_C has an index C which defines which set of diagram components is assigned a type. The set $\mathcal{N}$ contains all element identifiers which in turn can only be assigned element types from the set $\mathcal{L}$.

- $\lambda_{\mathcal{Y}} : \mathcal{Y} \rightarrow \mathcal{Z}$ is a total function that maps each asset identifier to an asset type. The definition of this function is similar to $\lambda_{\mathcal{N}}$.

- $\lambda_{\mathcal{A}} : \mathcal{A} \rightarrow \mathcal{B}$ is a total function that maps each boundary identifier to a boundary type. The definition of this function is similar to $\lambda_{\mathcal{N}}$.

- $\lambda_{\mathcal{R}} : \mathcal{R} \rightarrow \mathcal{T}$ is a total function that maps each connector identifier to connector type. The definition of this function is similar to $\lambda_{\mathcal{N}}$.

- $\mu : (\mathcal{N} \cup \mathcal{Y} \cup \mathcal{R}) \times \mathcal{K} \nrightarrow \mathcal{V}$ is a partial function that maps an element, asset or a connector identifier and a property key to a certain value.

As described before, each element, asset, or connector has an assigned type. Each type can contain multiple properties. Each property has multiple potential values. The function μ defines the selected value for a property key. It is a partial function as not every property is defined for every component.

- $\delta \subseteq \mathcal{N} \times \mathcal{N}$ is a transitive, irreflexive, asymmetric relation between elements in D. This indicates that one element is contained by another element. The tuple (n_1, n_2) indicates that n_1 is the *parent* of n_2. So to say n_1 *contains* n_2 and n_2 *is contained by* n_1.

- $\kappa \subseteq \mathcal{A} \times (\mathcal{A} \cup \mathcal{N})$ is a transitive, irreflexive, asymmetric relation between boundaries and elements in D. That indicates that one element is located inside a boundary or a boundary is contained by another boundary. The tuple (b_1, n_1) indicates that the boundary with the identifier b_1 contains the element with the identifier n_1. An element cannot contain a boundary. However, a boundary can contain other boundaries.

- $\rho \subseteq (\mathcal{N} \cup \mathcal{R}) \times \mathcal{Y}$ is a relation between elements and connectors on the one hand and assets on the other hand. This relation indicates which asset is held by an element or connector. Multiple elements and connectors can hold the same asset to represent the movement of the asset through the system.

2.2.6 Example Diagram-Definition

To provide a better understanding of this definition, the example in Section 2.2.2 is transferred to: $D = \langle \mathcal{N}, \mathcal{Y}, \mathcal{A}, \mathcal{R}, source, target, \lambda_C, \mu, \delta, \kappa, \rho \rangle$:

- $\mathcal{N} = \{n_1, n_2, n_3, n_4, n_5, n_6\}$
  Contains all the unique element identifiers.

- $\mathcal{Y} = \{y_1, y_2\}$
  Contains all the unique asset identifiers.

- $\mathcal{A} = \{a_1, a_2\}$
  Contains all the unique boundary identifiers.

- $\mathcal{R} = \{r_1, r_2, r_3, r_4, r_5, r_6\}$
  Contains all the unique connector identifiers.

- $source(r_1) = n_2, source(r_2) = n_4, source(r_3) = n_5, source(r_4) = n_6, source(r_5) = n_5, source(r_6) = n_3$
  The *source* function maps each connector identifier to an element identifier since each connector needs a starting point.

- $source(r_1) = n_3, source(r_2) = n_5, source(r_3) = n_6, source(r_4) = n_5, source(r_5) = n_3, source(r_6) = n_2$
  The *target* function maps each connector identifier to an element identifier since each connector needs an end point.

  .

- $\lambda_{\mathcal{N}}(n_1) =$ Mobile Phone, $\lambda_{\mathcal{N}}(n_2) =$ Application,
  $\lambda_{\mathcal{N}}(n_3) =$ Cell Tower, $\lambda_{\mathcal{N}}(n_4) =$ Server,
  $\lambda_{\mathcal{N}}(n_5) =$ REST Interface, $\lambda_{\mathcal{N}}(n_6) =$ Database

  The function $\lambda_{\mathcal{N}}$ maps each element instance to a type from $\mathcal{L}$. For example, the element with the identifier n_1 has the assigned type "Mobile Phone".

- $\lambda_{\mathcal{Y}}(y_1) =$ User Credentials, $\lambda_{\mathcal{Y}}(y_2) =$ Confidential User Data
  The function $\lambda_{\mathcal{Y}}$ maps each asset instance to a type from $\mathcal{Z}$. For example, the asset with the identifier y_1 has the assigned type "User Credentials".

- $\lambda_{\mathcal{A}}(a_1) =$ Untrusted Environment, $\lambda_{\mathcal{A}}(a_2) =$ Trusted Environment
  The function $\lambda_{\mathcal{A}}$ maps each boundary instance to a type from $\mathcal{B}$. For example, the boundary with the identifier a_1 has the assigned type "Untrusted Environment".

- $\lambda_{\mathcal{R}}(r_1) =$ Wireless, $\lambda_{\mathcal{R}}(r_2) =$ Wired,
  $\lambda_{\mathcal{R}}(r_3) =$ Wired $\lambda_{\mathcal{R}}(r_4) =$ Wired

$\lambda_{\mathcal{R}}(r_5) = \text{Wired}$ $\lambda_{\mathcal{R}}(r_6) = \text{Wireless}$
The function $\lambda_{\mathcal{R}}$ maps each connector instance to a type from $\mathcal{T}$. For example, the connector with the identifier r_1 has the assigned type "Wireless".

- $\mu(n_1, \text{Operating System}) = \text{Android}$, $\mu(n_1, \text{Vendor}) = \text{Third Party}$,
$\mu(n_2, \text{Vendor}) = \text{Third Party}$, $\mu(n_4, \text{Vendor}) = \text{Third Party}$,
$\mu(n_5, \text{Input Validation}) = \text{Yes}$, $\mu(n_5, \text{Input Sanitization}) = \text{No}$,
$\mu(n_6, \text{Encrypted}) = \text{Yes}$,
$\mu(r_1, \text{Protocol}) = \text{HTTP}$, $\mu(r_2, \text{Protocol}) = \text{HTTP}$,
$\mu(r_3, \text{Protocol}) = \text{HTTP}$, $\mu(r_4, \text{Protocol}) = \text{HTTP}$,
$\mu(r_5, \text{Protocol}) = \text{HTTP}$, $\mu(r_6, \text{Protocol}) = \text{HTTP}$,
$\mu(y_1, \text{Encrypted}) = \text{No}$, $\mu(y_2, \text{Encrypted}) = \text{Yes}$

  The function μ maps for each element, asset, or connector instance a property key to a concrete value. The function μ is only valid if the assigned element, asset, or connector type allows the property key and the property value is allowed for the specific property key

- $\delta = \{(n_1, n_2), (n_4, n_5), (n_4, n_6)\}$
The relation δ describes which element contains another element or which element is contained by another one. For example the element with the identifier n_1 contains the element n_2. Inside the diagram example, this refers to the "Mobile Phone", which contains the "Application" element.

- $\kappa = \{(a_1, n_1), (a_1, n_2), (a_1, n_3), (a_2, n_4), (a_2, n_5), (a_2, n_6)\}$
The relation κ describes which boundary contains another boundary or element. For Example the boundary a_1 contains the elements n_1, n_2 and n_3.

- $\rho = \{(r_1, y_1), (n_3, y_1), (r_2, y_1), (n5, y_1), (r_3, y_1), (n_6, y_1)$
$(r_6, y_2), (n_3, y_2), (r_5, y_2), (n5, y_2), (r_4, y_2), (n_6, y_2)\}$
The relation ρ describes which element or connector holds an asset. An example tuple (n_6, y_2) from Figure 2 indicates that the element "Database" n_6 *holds* the "User Data" asset y_2.

2.2.7 Relation between Content-Specification and Diagram-Definition

This section defines the conformance relation between the diagram $D = \langle \mathcal{N}, \mathcal{Y}, \mathcal{A}, \mathcal{R}, source, target, \lambda_C, \mu, \delta, \kappa, \rho \rangle$ and the content-specification $S = \langle \mathcal{L}, \mathcal{Z}, \mathcal{B}, \mathcal{T}, \mathcal{K}, \mathcal{V}, \mathcal{H}, \mathcal{E}, \iota_C, \eta, \gamma \rangle$.

The expression $D \models S$ means that a diagram D conforms to the specification S. The symbol $\models$ describes a *conformance relation* and says that D *conforms to* S. $D \models S$ holds iff all of the following conditions hold:

- $|\mathcal{N}| > 0$
  D contains at least one element.

- $|\mathcal{R}| = 0 \vee (|\mathcal{R}| > 0 \rightarrow |\mathcal{N}| \geq 2)$
  D contains no connector, or if it does, it also contains at least two elements.

- $|\mathcal{Y}| = 0 \vee (|\mathcal{Y}| > 0 \rightarrow (|\mathcal{N}| \geq 1 \vee |\mathcal{Y}| \geq 1))$
  D contains no asset or if it does it also contains at least one element or one connector, because an asset must be held by an element or connector.

- $\forall n \in \mathcal{N}, \lambda_{\mathcal{N}}(n) \in \mathcal{L}$
  Each element n in D must have an assigned type from set $\mathcal{L}$.

- $\forall y \in \mathcal{Y}, \lambda_{\mathcal{Y}}(y) \in \mathcal{Z}$
  Each asset y in D must have an assigned type from set $\mathcal{Z}$.

- $\forall a \in \mathcal{A}, \lambda_{\mathcal{A}}(a) \in \mathcal{B}$
  Each boundary a in D must have an assigned type from set $\mathcal{B}$.

- $\forall r \in \mathcal{R}, \lambda_{\mathcal{R}}(r) \in \mathcal{T}$
  Each connector r in D must have an assigned type from set $\mathcal{T}$.

- $\forall n \in \mathcal{N}, k \in \mathcal{K}, \mu(n,k)$ is *defined* $\Leftrightarrow k \in \eta(\lambda(n))$
  If an element instance n has an assigned property key k, then k must be valid for the assigned type of n.

- $\forall y \in \mathcal{Y}, k \in \mathcal{K}, \mu(y,k)$ is *defined* $\Leftrightarrow k \in \eta(\chi(y))$
  If an asset instance y has an assigned property key k, then k must be valid for the assigned type of n.

- $\forall r \in \mathcal{R}, k \in \mathcal{K}, \mu(r,k)$ is *defined* $\Leftrightarrow k \in \eta(\tau(r))$
  If a connector instance r has an assigned property key k, then k must be valid for the assigned type of r.

- $\forall j \in (\mathcal{N} \cup \mathcal{Y} \cup \mathcal{R}), k \in \mathcal{K}, \mu(j,k)$ is *defined* $\Leftrightarrow \mu(j,k) \in \gamma(k)$
  If an element, asset, or connector j has a property key k, then the assigned value for the property k must be in the set of the possible values for k.

Every diagram D, such that $D \models S$, can be checked against the threat-model, formalized in the next section.

2.2.8 The Rule-based Threat-Model

The following sections describe the rule-based threat-model. A rule consists of a title, a description, an assigned threat type, an impact estimation, a likelihood estimation, and, most importantly, the so-called anti-pattern.
The anti-pattern is at the heart of every rule. It expresses threats in a human as well as machine-readable language. Each anti-pattern describes an undesirable condition inside a system.

2.2.9 Rule Syntax

Figure 3 and Figure 4 display the full syntax of the context-free anti-pattern grammar. The notation of the grammar was inspired by the "Extended Backus–Naur form" (EBNF), where each line represents a production rule of the syntax. Each production rule, in the following referred to as "term", consists of terminal and non-terminal tokens. A terminal is marked in red and describes an immutable part of the syntax. The non-terminal tokens of the syntax are marked in blue. Each non-terminal must be replaced by the associated term. The syntax differentiates between "patterns" and "filters". A pattern relates to a diagram component such as the element-pattern (elPat).
A pattern examines whether an element, asset, boundary, or connector exists within the diagram or not. An assigned type can further restrict each pattern except the flow-pattern. The assigned type of a diagram component can be examined using a type-filter (typeFil). The type-filter as well as most of the other filters are optional.
A pattern can but does not necessarily have additional filters assigned. A filter specifies additional conditions that a pattern must meet in order to correspond to a threat. For example, the property-filter (propFil) can be applied to an element-pattern, connector-pattern (conPat) or asset-pattern (assetPat) and verifies whether the component has a specific property with a specific value. The meaning and intended use of all patterns and filters are presented in Section 2.2.11.
Some lines of the syntax contain the following black symbols " | ", " + " and question mark " ? ". The black vertical line describes a logical OR. The red vertical bar is a terminal symbol of the language itself. The plus indicates that the symbols enclosed in black brackets must appear at least once. However, they may also occur multiple times. All red brackets are part of the syntax itself. The question mark indicates that the preceding symbols can occur but do not need to.

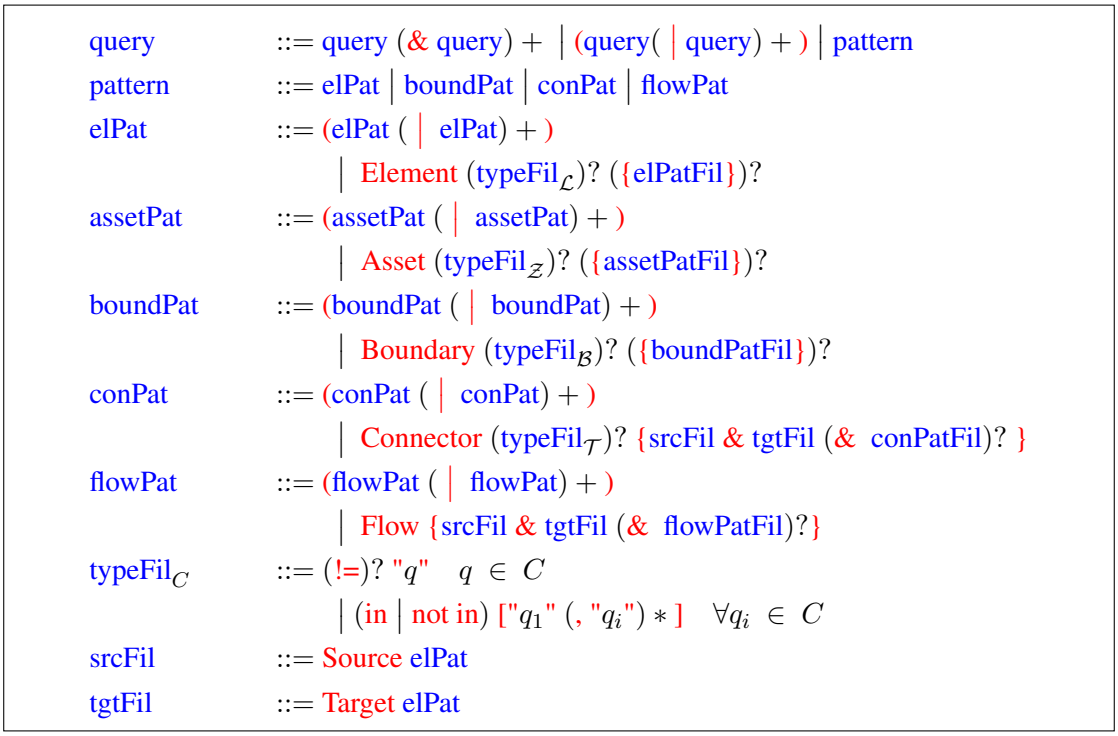

Figure 3: Syntax of Patterns

elPatFil ::= elPatFil (& elPatFil) + | (elPatFil(| elPatFil) +)
| propFil | assetFil | elRelFil | conFil | flowFil

assetPatFil ::= assetPatFil (& assetPatFil)+
| (assetPatFil(| assetPatFil) +) | propFil

boundPatFil ::= boundPatFil (& boundPatFil)+
| (boundPatFil(| boundPatFil) +) | boundRelFil

conPatFil ::= conPatFil (& conPatFil) + | (conPatFil(| conPatFil) +)
| propFil | assetFil | conCrossesFil

flowPatFil ::= flowPatFil (& flowPatFil) + | (flowPatFil(| flowPatFil) +)
| includesFil | flowCrossesFil

tvFil ::= "k" (= | !=)"v" $k \in \mathcal{K}, v \in \mathcal{V}$
| "k" (in | not in) ["v_1" (, "v_i") $*$] $k \in \mathcal{K}, \forall v_i \in \mathcal{V}$

assetFil ::= Holds assetPat

elRelFil ::= Contains (no)? elPat
| (Not)? Contained by (elPat | boundPat)

boundRelFil ::= Contains (no)? (elPat | boundPat)
| (Not)? Contained by boundPat

conFil ::= Has (no)? Connector ("conType")?
{(srcFil | tgtFil) (& conPatFil)?}

flowFil ::=Has (no)? Flow {(srcFil | tgtFil) (& flowPatFil)?}

conCrossesFil ::= Crosses (elPat | boundPat)

flowCrossesFil ::= Crosses (elPat | boundPat)

includesFil ::= Includes (no | only)? (elPat | conPat)

Figure 4: Syntax of Filters

2.2.10 Flow Definition

Before the syntax is explained in the next section, the concept of *flows* is introduced in this section. A flow in a diagram is a finite alternating sequence of elements and connectors between a start point (element) and an endpoint (element). It should also be mentioned that these two points do not have to be directly connected to each other. In addition, a flow cannot contain loops so that an element can appear several times in the flow sequence. In order to prevent infinite loops, each connector and connector must be unique within a flow sequence.
A flow $p = seq(\mathcal{N} \cup \mathcal{R})$ has the following structure $p = (n_1, r_1, n_2, \ldots, r_{i-1}, n_i)$. The following functions can be applied to a flow:

- $\mathsf{pSource}(p)$ is a function that takes a flow sequence p as an argument and returns the first element identifier in the sequence:
  $\mathsf{pSource}(n_1, r_1 \ldots r_{i-1}, n_i) = n_1$.

- $\mathsf{pTarget}(p)$ is a function that takes a flow sequence p as argument and returns the last element identifier in the sequence:
  $\mathsf{pTarget}(n_1, r_1 \ldots r_{i-1}, n_i) = n_i$.

- $\mathsf{elements}(p)$ is a function that takes a flow sequence p as argument and returns the set of all element identifiers in the flow:
  $\mathsf{elements}(n_1, r_1 \ldots r_{i-1}, n_i) = \{n_1 \ldots n_i\}$.

- $\mathsf{connectors}(p)$ is a function that takes a flow p as argument and returns the set of all connector identifiers in the flow:
  $\mathsf{connectors}(n_1, r_1 \ldots r_{i-1}, n_i) = \{r_1 \ldots r_{i-1}\}$.

The function $\mathsf{flows}(D, src, tgt)$ takes three arguments: a diagram instance D, a source element src and target element tgt identifier. The function returns a set of all possible flow sequences between the source element, and the target element within the diagram. Each of these sequences represents a single flow p in D. At this point it should be emphasized that mathematically a flow can be compared with a path between two elements. The formal definition of $\mathsf{flows}(D, src, tgt) =$:

$$\{(n_1, r_1 \ldots r_{i-1}, n_i) \mid (\forall 1 \leq j < i,\ source(r_j) = n_j \wedge\ target(r_j) = n_{j+1} \tag{1}$$
$$\wedge\ (\nexists\ 1 \leq j < k < i,\ (n_j = n_k)\} \tag{2}$$

Line (1) states that all connectors within a flow always link two consecutive elements and the target of connector r_j is always the source of the connector r_{j+1}.
The second line (2) expresses that each element must be unique in the sequence.
The corresponding algorithm of this function is presented in Section 2.3.2.

2.2.11 Relation between Analysis Language and Content-Specification

This section depicts the conformance relation between the rule anti-pattern content and the content-specification $S = \langle \mathcal{L}, \mathcal{Z}, \mathcal{B}, \mathcal{T}, \mathcal{K}, \mathcal{V}, \mathcal{H}, \mathcal{E}, \iota_C, \eta, \gamma \rangle$. Therefore, the conformance relation is denoted as $term \models S$, where $term$ refers to a rule in the syntax.

In some cases the conformance relation requires a context, denoted as $\models_c$, where c is the context. The context $c \subseteq (\mathcal{L} \cup \mathcal{Z} \cup \mathcal{T} \cup \mathcal{B})$ is a subset of element types, asset types, boundary types or connector types which are under consideration. Since a path pattern (pathPat) cannot have an assigned type, it cannot be assigned to the context c.

By $term \models S \Leftrightarrow term' \models_c S$ it is denoted that $term$ conforms to a specification S iff $term'$ conforms to S under context c. Not all terms require a context for their conformance relation, in particular the conformance relation for query does not require a context. nil is defined as a special replacement for optional parameters which are omitted. Since nil is not part of the specification S, it never conforms to it: $\text{nil} \not\models S$

Since most of the pattern and filter conformance relations differ only marginally, only the first is described in detail in order to avoid repetition.

The query term is the root or starting point of the syntax. This is a non-terminal symbol which can be logically linked (& |) multiple times with itself or can be replaced by a pattern term. It is the only term that can be used to connect different patterns.

$$\bullet\ \text{query}_1\ \&\ \text{query}_2 \models S \Leftrightarrow \text{query}_1 \models S \wedge \text{query}_2 \models S \quad (1)$$

$$\bullet\ (\text{query}_1 \mid \text{query}_2) \models S \Leftrightarrow \text{query}_1 \models S \wedge \text{query}_2 \models S \quad (2)$$

Line (1) describes how multiple query terms are linked with a logical AND (&). Both sub-terms have to conform to the content-specification to form a valid rule query.
The second line (2) shows two logically OR (|) connected query terms. Although the two query terms are linked with a logical OR on the left, the conformance relationships to the right are linked with an AND ($\wedge$). The entire term conforms to the specification S only if the first query_1 term AND ($\wedge$) the second query_2 term conform to S.

The **element-pattern** elPat is used to examine the elements of a diagram.

$$\bullet\ (\text{elPat}_1 \mid \text{elPat}_2) \models S \Leftrightarrow \text{elPat}_1 \models S \wedge \text{elPat}_2 \models S \quad (1)$$

$$\bullet\ \text{Element}\ (\text{typeFil})?\ (\{\text{elPatFil}\})? \models S \Leftrightarrow \quad (2)$$

$$(\text{typeFil} \models_{\mathcal{L}} S \vee \text{typeFil} = \text{nil}) \wedge (\text{elPatFil} = \text{nil} \vee \text{elPatFil} \models_c S) \quad (3)$$

The first line (1) of every pattern term describes that the same pattern can occur several times if it is linked by a logical OR (|). Each sub-pattern has to conform to the specification S. Therefore they are connected by a logical AND ($\wedge$).
Line (2) presents the complete element-pattern. Each pattern starts with its corresponding key-

word (Element, Asset, Boundary, Connector, Flow). It should be noted that the type-filter (typefil) and the element-pattern-filters (elPatFil) definition are marked with a question mark. Therefore, they can be omitted. If they are omitted, these terms are replaced by nil.
(3) The provided type-filter (typefil) is either omitted and thus nil or must conform to the specification S. The context of the type-filter evaluation is $\mathcal{L}$ since an element-pattern is used to analyze all element-components within the diagram D. If an element-pattern has additional filters (elPatFil), these have to conform to the specification S under the consideration of the context c. This context contains either the set of element-types ($c \subseteq \mathcal{L}$) defined within the provided type-filter (typefil) or nil if the type-filter was omitted.
The conformance relation of the asset-pattern (assetPat) and boundary-pattern (boundPat) is similar to this one.

The **asset-pattern** (assetPat) is used to examine the assets of a system. It is the only pattern that cannot stand alone but is used in the so-called asset-filter (assetFil).

- $(\text{assetPat}_1 \mid \text{assetPat}_2) \models S \Leftrightarrow \text{assetPat}_1 \models S \wedge \text{assetPat}_2 \models S$
- $\text{Asset}\ (\text{typeFil})?\ (\{\text{assetPatFil}\})? \models S \Leftrightarrow$
  $(\text{typeFil} \models_{\mathcal{Z}} S \vee \text{typeFil} = \text{nil}) \wedge (\text{assetPatFil} = \text{nil} \vee \text{assetPatFil} \models_c S)$

The **boundary-pattern** (boundPat) examines the defined boundary components of a diagram.

- $(\text{boundPat}_1 \mid \text{boundPat}_2) \models S \Leftrightarrow \text{boundPat}_1 \models S \wedge \text{boundPat}_2 \models S$
- $\text{Boundary}\ (\text{typeFil})?\ (\{\text{boundPatFil}\})? \models S \Leftrightarrow$
  $(\text{typeFil} \models_{\mathcal{B}} S \vee \text{typeFil} = \text{nil}) \wedge (\text{boundPatFil} = \text{nil} \vee \text{boundPatFil} \models_c S)$

The **connector-pattern** (conPat) is used to investigate the connectors.

- $(\text{conPat}_1 \mid \text{conPat}_2) \models S \Leftrightarrow \text{conPat}_1 \models S \wedge \text{conPat}_2 \models S$ (1)
- $\text{Connector}\ (\text{typeFil})?\ \{\text{srcFil}\ \&\ \text{tgtFil}\ (\&\ \text{conPatFil})?\} \models S \Leftrightarrow$ (2)
  $(\text{typeFil} \models_{\mathcal{T}} S \vee \text{typeFil} = \text{nil}) \wedge \text{srcFil} \models S \wedge \text{tgtFil} \models S \wedge$ (3)
  $(\text{conPatFil} = \text{nil} \vee \text{conPatFil} \models_c S)$ (4)

Line (2) depicts the complete connector-pattern. Compared to the previous patterns this pattern includes a mandatory source-filter (srcFil) and target-filter (tgtFil).
(3) In order that the connector-pattern conforms to the specification S, both the source-filter and the target-filter have to conform to S.

The **flow-pattern** (flowPat) is used to evaluate these flow sequences. Section 2.2.10 introduced the concept of flows.

- (flowPat$_1$ | flowPat$_2$) $\models S \Leftrightarrow$ flowPat$_1$ $\models S \wedge$ flowPat$_2$ $\models S$ (1)
- Flow {srcFil & tgtFil (& flowPatFil)?} $\models S \Leftrightarrow$ (2)

  srcFil $\models S \wedge$ tgtFil $\models S \wedge$ (flowPatFil $=$ nil $\vee$ flowPatFil $\models S$) (3)

(2) The flow-pattern is the only pattern that contains no type specification as a flow consists of various elements and connectors. Similar to the connector-pattern (conPat), the flow-pattern always includes a source-filter (srcFil) and a target-filter (tgtFil).
(3) The source-filter as well as the target-filter have to conform to S. The validation of the additional filters applied to the flow-pattern requires no additional context c, as a flow cannot have an assigned type.

The **type-filter** (typeFil) is used to examine elements, assets, boundaries and connectors according to their assigned type. Therefore, the evaluation of the type-filter also has a context c which contains the set of diagram component types, depending on the pattern that holds the type-filter.

- : (!=)? "q" $\models_c S \Leftrightarrow q \in c$ (1)
- : (in | not in)? ["q_1" , ... "q_i"] $\models_c S \Leftrightarrow \forall q_i \in c$ (2)

The context c is either the set of element-types ($\mathcal{L}$), asset-types ($\mathcal{Z}$), boundary-types ($\mathcal{B}$), or connector-types ($\mathcal{T}$). Depending on the previous pattern.
Line (1) describes the simple version of the type-filter. In this case it is checked whether the diagram component under investigation has or has exactly not the assigned type defined by q. The type-filter conforms to the specification S if the specified type is contained by the type set inside the context c.
Line (2) depicts an advanced version of the type-filter.In this case it is checked whether the diagram component has one of or has exactly none of the specified types defined by $q_1 \ \ldots \ q_i$. The type-filter conforms to the specification S if all the specified types $q_1 \ \ldots \ q_i$ are contained by the type set inside the context c.

The **source-filter** (srcFil) is used within the connector-pattern and flow-pattern to specify the source element. The **target-filter** (tgtFil) is used to specify the target element.

- Source elPat $\models_c S \Leftrightarrow$ elPat $\models S$ (1)
- Target elPat $\models_c S \Leftrightarrow$ elPat $\models S$ (2)

(1-2) The source-filter as well as the target-filter conforms to the specification if the defined element-pattern (elPat) conforms to S.

The term elPatFil describes all filters which can be assigned to an element-pattern. An element-pattern can be filtered according to its properties (propFil), its assets (assetFil), its rela-

tionships to other elements and boundaries (elRelFil) and its incoming and outgoing connectors (conFil) as well as flows (flowFil).

$$\bullet\ \text{elPatFil}_1\ \&\ \text{elPatFil}_2\ \models_c S \Leftrightarrow \text{elPatFil}_1\ \models_c S \wedge \text{elPatFil}_2\ \models_c S \quad (1)$$

$$\bullet\ (\text{elPatFil}_1\ |\ \text{elPatFil}_2)\ \models_c S \Leftrightarrow \text{elPatFil}_1\ \models_c S \wedge \text{elPatFil}_2\ \models_c S \quad (2)$$

Each pattern of the syntax can be assigned multiple filters. These filters are connected either using a logical AND (&) or a logical OR (|). The conformance relation of the following filters is similar to this one.
Line (1) depicts two filters that are logically connected by an AND (&). In this case, each of the filters has to conform to the specification S.
Line (2) depicts two filters that are logically connected by an OR (|). Similar to the line (1), each of the filters has to conform to the specification S. Since it is not a matter of the semantic evaluation of the language, both conformance relations are connected in line 4 with an AND ($\wedge$).

The asset-specific filters are defined using the term assetPatFil. An asset-pattern can only be filtered according to its properties propFil.

$$\bullet\ \text{assetPatFil}_1\ \&\ \text{assetPatFil}_2\ \models_c S \Leftrightarrow \text{assetPatFil}_1\ \models_c S \wedge \text{assetPatFil}_2\ \models_c S$$

$$\bullet\ (\text{assetPatFil}_1\ |\ \text{assetPatFil}_2)\ \models_c S \Leftrightarrow \text{assetPatFil}_1\ \models_c S \wedge \text{assetPatFil}_2\ \models_c S$$

The filters of the boundary-pattern are specified using the boundPatFil term. Similar to the asset-filters, a boundary-pattern has only one filter which refers to the relationship between the boundary and other elements boundRelFil.

$$\bullet\ \text{boundPatFil}_1\ \&\ \text{boundPatFil}_2\ \models_c S \Leftrightarrow \text{boundPatFil}_1\ \models_c S \wedge \text{boundPatFil}_2\ \models_c S$$

$$\bullet\ (\text{boundPatFil}_1\ |\ \text{boundPatFil}_2)\ \models_c S \Leftrightarrow \text{boundPatFil}_1\ \models_c S \wedge \text{boundPatFil}_2\ \models_c S$$

The connector-specific filters are defined using the term conPatFil. A connector-pattern can be filtered according to its assigned properties (propFil), its assets (assetFil) and whether it crosses an element or boundary (conCrossesFil).

$$\bullet\ \text{conPatFil}_1\ \&\ \text{conPatFil}_2\ \models_c S \Leftrightarrow \text{conPatFil}_1\ \models_c S \wedge \text{conPatFil}_2\ \models_c S$$

$$\bullet\ (\text{conPatFil}_1\ |\ \text{conPatFil}_2)\ \models_c S \Leftrightarrow \text{conPatFil}_1\ \models_c S \wedge \text{conPatFil}_2\ \models_c S$$

The flow-related filters are given by the term flowPatFil. A flow-pattern can be filtered according to its included elements and connectors (includesFil) and whether it crosses a boundary or element (flowCrossesFil).

$$\bullet\ \text{flowPatFil}_1\ \&\ \text{flowPatFil}_2\ \models S \Leftrightarrow \text{flowPatFil}_1\ \models S \wedge \text{flowPatFil}_2\ \models S$$

$$\bullet\ (\text{flowPatFil}_1\ |\ \text{flowPatFil}_2)\ \models S \Leftrightarrow \text{flowPatFil}_1\ \models S \wedge \text{flowPatFil}_2\ \models S$$

The **property-filter** propFil is used to examine elements, assets and connectors according to their assigned properties.

- "k" (= | !=) "v" $\models_c S \Leftrightarrow$ (1)

  $k \in \mathcal{K} \wedge (c = \text{nil} \vee \forall 1 \leq j \leq i, k \in \eta(c_j)) \wedge v \in \gamma(k)$ (2)
- "k" (in | not in) ["v_1" , ... "v_i"] $\models_c S \Leftrightarrow$ (3)

  $k \in \mathcal{K} \wedge (c = \text{nil} \vee \forall 1 \leq j \leq i, k \in \eta(c_j)) \wedge \forall 1 \leq j \leq i, v_j \in \gamma(k)$ (4)

The first two lines (1-2) describe the first variant of this filter. This property-filter conforms to S if k is part of the property key set $\mathcal{K}$. Moreover, v has to be a valid value for the key k determined by the function $\gamma(k)$. If the context c is not nil, then the property key k must be valid for each type inside the type set stored in context c determined by the function $\eta(c)$.
The second variant of this filter is described in lines (3-4). Similar to the first variant, the property key k must be part of $\mathcal{K}$, and if the context c is not nil, then it must be a valid key for each component type inside the context. Unlike the first variant, this one deals with a set of values $\{v_1, v_2 \dots v_i\}$, which must all be valid values for k.

The **asset-filter** (assetFil) determines whether an element holds an asset or a connector transports an asset. The filter can be applied to an element-pattern (elPat) or a connector-pattern (conPat), since these are the only diagram components which can be related to an asset.

- Holds assetPat $\models_c S \Leftrightarrow$ assetPat $\models S$ (1)

(1) The asset-filter conforms to the specification if the defined asset-pattern (assetPat) conforms to S.

The **element-relation-filter** (elRelFil) can only be used in an element-pattern (elPat) and examines the relationships between an element and other elements and boundaries. The context c is either nil or an element type (elType).

- Contains (no)? elPat $\models_c S \Leftrightarrow$ elPat $\models S$ (1)
- (Not)? Contained by elPat $\models_c S \Leftrightarrow$ elPat $\models S$ (2)
- (Not)? Contained by boundPat $\models_c S \Leftrightarrow$ boundPat $\models S$ (3)

(1) The first variant of this filter is valid if the defined element-pattern (elPat) conforms S.
The conformance relation of line (2) is similar to the first variant in line (1).
Line (3) conforms to the specification if the boundary-pattern (boundPat) does.

The **boundary-relation-filter** (boundRelFil) is the only filter that can be assigned to a boundary-pattern (boundPat). This filter is similar to the element-relation-filter, but in this case from the

perspective of a boundary. The context c is either nil or a set of boundary types (boundType).

- Contains (no)? elPat $\models_c S \Leftrightarrow$ elPat $\models S$
- Contains (no)? boundPat $\models_c S \Leftrightarrow$ boundPat $\models S$
- (Not)? Contained by boundPat $\models_c S \Leftrightarrow$ boundPat $\models S$

The conformance relation of this filter is similar to the element-relation-filter (elRelfil).

The **connector-filter** (conFil) can only be assigned to an element-pattern (elPat). This filter is very similar to the connector-pattern, but it has either a source-filter (srcFil) or a target-filter (tgtFil). The element-pattern which contains this filter forms the counterpart of the defined source or target-filter.

$$\begin{aligned}
&\text{• Has (No)? Connector (typeFil)? \{(srcFil} \mid \text{tgtFil) (\& conPatFil)?\}} \models_c S \Leftrightarrow && (1)\\
&\quad (\text{typeFil} \models_{\mathcal{T}} S \lor \text{typeFil} = \text{nil}) \land && (2)\\
&\quad (\text{srcFil} \models S \lor \text{tgtFil} \models S) \land && (3)\\
&\quad (\text{conPatFil} = \text{nil} \lor \text{conPatFil} \models_c S) && (4)
\end{aligned}$$

Line (1) depicts the full syntax of the filter. The syntax is similar to the connector-pattern (conPat), except that this filter requires either a source-filter or a target-filter.
(3) The specified source or target-filter must conform to the specification S. The other is nil and does therefore not conform to the specification.

The **flow-filter** (flowFil) is similar to the flow-pattern (flowPat) but it only requires either the source (srcFil) or the target-filter (tgtFil). This filter can only be assigned to an element-pattern (elPat). The conformance relation of this filter is similar to the connector-filter (conFil).

$$\begin{aligned}
&\text{• Has (No)? Flow \{(srcFil} \mid \text{tgtFil) (\& flowPatFil)?\}} \models_c S \Leftrightarrow\\
&\quad (\text{srcFil} \models S \lor \text{tgtFil} \models S) \land (\text{flowPatFil} = \text{nil} \lor \text{flowPatFil} \models S)
\end{aligned}$$

The **connector-crosses-filter** (conCrossesFil) can only be assigned to a connector-pattern (conPat) or a connector-filter (conFil). It is used to check if a connector crosses a boundary or element.

$$\begin{aligned}
&\text{• Crosses elPat} \models_c S \Leftrightarrow \text{elPat} \models S && (1)\\
&\text{• Crosses boundPat} \models_c S \Leftrightarrow \text{boundPat} \models S && (2)
\end{aligned}$$

(1) The first variant of this filter is valid if the defined element-pattern (elPat) conforms S.
Line (2) conforms to the specification if the boundary-pattern (boundPat) does.

The **flow-crosses-filter** (flowCrossesFil) can only be assigned to a flow-pattern (flowPat) or a connector-filter (flowFil). The conformance relation of this filter is similar to the connector-

crosses-filter (conCrossesFil). It is used to check if one of the connectors within the flow crosses a boundary or element.

- Crosses elPat $\models_c S \Leftrightarrow$ elPat $\models S$
- Crosses boundPat $\models_c S \Leftrightarrow$ boundPat $\models S$

The **includes-filter** (includesFil) can only be applied to a flow-pattern (flowPat) or a flow-filter (flowFil). It is used to check if a flow contains a specific element or connector.

- Includes (no | only)? elPat $\models_c S \Leftrightarrow$ elPat $\models S$ (1)
- Includes (no | only)? conPat $\models_c S \Leftrightarrow$ conPat $\models S$ (2)

(1) The first variant of this filter is valid if the defined element-pattern (elPat) conforms to S.
(2) The second variant of this filter is valid if the defined connector-pattern (conPat) conforms to S.

2.3 Method Concept

The previous sections presented the formal system-model and threat-model. The following sections discuss the semantic evaluation of the language and its grammar. The first part elaborates the evaluation of the individual patterns and applied filters. Section 2.3.2 presents the flows(D, src, tgt) algorithm.

2.3.1 Semantic Evaluation of Anti-Patterns

In contrast to the previous section, the semantic evaluation on the basis of a concrete diagram is examined, and not whether an anti-pattern is syntactically correct. $D = \langle \mathcal{N}, \mathcal{Y}, \mathcal{A}, \mathcal{R}, source, target, \lambda_C, \mu, \delta, \kappa, \rho \rangle$. Therefore, the context c has to be redefined.
In this case, the context $c \in (\mathcal{N} \cup \mathcal{Y} \cup \mathcal{A} \cup \mathcal{R} \cup \{seq(\mathcal{N} \cup \mathcal{R})\})$ contains an element, asset, boundary or connector identifier or a flow sequence which is under consideration of the evaluation.
Only for the type-filter (typeFil) has a special context, denoted as capital C. This special context is necessary because the type filter considers all diagram components according to the assigned type. In this case the context C contains either the set of all element identifiers ($\mathcal{N}$), asset identifiers ($\mathcal{Y}$), boundary identifiers ($\mathcal{A}$) or connector identifiers ($\mathcal{R}$).

Let $[\![term]\!]_D$, $[\![term]\!]_{D,c} or [\![term]\!]_C$ be the evaluation function of a term with respect to a diagram D and an optional context c, type-filter context C. The evaluation function follows a recursive manner, where each evaluation results in a set of tuples. Each tuple has the form (x, M). Where x represents the element, asset, boundary, connector identifier, or flow sequence, which is under investigation. M is a set containing the additionally affected elements, assets, boundaries, connector identifiers, and flow sequences that result from the applied filters. For the top-level term query it is guaranteed that M also contains x itself. This is not true for the evaluation of some of the sub-terms. x is relevant only as an intermediate result for the recursive calls to $[\![]\!]$. It is not a relevant result of the analysis, which only asks for the affected elements M. The symbol "_" is used to indicate that any value could be put at this position of the tuple.

The query term is the entry point of each rule query evaluation. Each query term is replaced by one of the pattern terms.

- $[\![\text{query}_1 \ \& \ \text{query}_2]\!] =$ (1)
  $\{(\_, M_1 \cup M_2) \mid (x_1, M_1) \in [\![\text{query}_1]\!]_D \ \wedge (x_2, M_2) \in [\![\text{query}_2]\!]_D\}$ (2)
- $[\![(\text{query}_1 \ \mid \ \text{query}_2)]\!]_D = [\![\text{query}_1]\!]_D \ \cup \ [\![\text{query}_2]\!]_D$ (3)

(1) The evaluation of several query terms, which are connected by a logical AND (&), requires that each of these patterns does not return an empty result. If one of the terms returns an empty set, the end result is the empty set. If all terms return a non-empty set, the end result is the cross-product of the results, meaning all combinations of affected elements are returned. Since query is the top-level entry point of the evaluation function, the first member of the returned tuple (_) is irrelevant.

(2) If multiple query terms are linked by a logical OR (|), the end result of the evaluation is the union of the resulting sets of tuples. A logical OR requires that only one of the two evaluations return a non-empty set. If both return an empty set, then the end result is the empty set. If one evaluation returns a non-empty set and the other does not, then the end result is the non-empty set. If both evaluations return a non-empty set, then the end result is the union of both sets. In this case, the cross product is not needed as the result of each query can independently realize the threat.

The **element-pattern** (elPat) allows the description of an anti-pattern which examines all elements in the diagram. During the analysis all elements $\mathcal{N}$ in the diagram D are compared to the element-pattern. All elements which conform to the pattern are returned.

- $[\![(\text{elPat}_1 \mid \text{elPat}_2)]\!]_D = [\![\text{elPat}_1]\!]_D \cup [\![\text{elPat}_2]\!]_D$ (1)
- $[\![\text{Element}]\!]_D = \{(n, \{n\}) \mid n \in \mathcal{N}\}$ (2)
- $[\![\text{Element typeFil}]\!]_D = \{(n, \{n\}) \mid n \in \mathcal{N} \wedge (n, \_) \in [\![\text{typeFil}]\!]_{\mathcal{N}}\}$ (3)
- $[\![\text{Element } \{\text{elPatFil}\}]\!]_D =$ (4)

  $\{(n, \{n\} \cup M) \mid n \in \mathcal{N} \wedge (n, M) \in [\![\text{elPatFil}]\!]_{D,n}\}$ (5)
- $[\![\text{Element typeFil } \{\text{elPatFil}\}]\!]_D =$ (6)

  $\{(n, \{n\} \cup M) \mid n \in \mathcal{N} \wedge (n, \_) \in [\![\text{typeFil}]\!]_{\mathcal{N}} \wedge$ (7)

  $(n, M) \in [\![\text{elPatFil}]\!]_{D,n}\}$ (8)

The first line (1) of every pattern term describes that the same pattern can occur several times if it is linked by a logical OR (|). The evaluation of each pattern returns a set of tuples or an empty set. If all evaluations return an empty set, the empty set is returned. If one of the evaluations returns a non-empty set, this set is returned. If multiple evaluations return a non-empty set, the union of the sets is returned as each pattern can independently realize the threat.
Line (2) displays the simplest form of the element-pattern, which does not specify any type or additional filters. Thus, this rule matches all elements and returns a set of tuples, whereby the number of tuples corresponds to the number of elements in the diagram since each tuple contains one of the elements. There are no additional affected components, therefore $\{n\}$ is returned.
(3) The resulting set contains only tuples whose elements correspond to the specified type-filter (typeFil). The evaluation of the type-filter takes place under the consideration of the context capital C, which contains all element identifiers contained in D.
(4-5) This element-pattern includes one or more additional filters (elPatFil). The context of the filter evaluation is the element identifier n. Some of the element-pattern-filters can affect additional diagram components which are contained in M. By design $[\![\text{elPatFil}]\!]_{D,n}$ returns $\{(n, M)\}$ if the element n passes the filter. If n does not pass the filter, the empty set will be returned.
(6-8) Returns only elements that match the specified type-filter (typeFil) and satisfy the additional filters. Similar to the lines (4-5), the resulting tuples contain the affected elements M.

The **asset-pattern** (assetPat) allows the description of an anti-pattern which examines all assets inside the diagram. All assets $\mathcal{Y}$ inside the diagram D are compared to the defined pattern.

The evaluation is similar to that of the element-pattern (elPat).

- $[\![(\text{assetPat}_1 \mid \text{assetPat}_2)]\!]_D = [\![\text{assetPat}_1]\!]_D \cup [\![\text{assetPat}_2]\!]_D$
- $[\![\text{Asset}]\!]_D = \{(y, \{y\}) \mid y \in \mathcal{Y}\}$
- $[\![\text{Asset typeFil}]\!]_D = \{(y, \{y\}) \mid y \in \mathcal{Y} \wedge (y, \_) \in [\![\text{typeFil}]\!]_{\mathcal{Y}}\}$
- $[\![\text{Asset } \{\text{assetPatFil}\}]\!]_D =$
  $\{(y, \{y\} \cup M) \mid y \in \mathcal{Y} \wedge (y, M) \in [\![\text{assetPatFil}]\!]_{D,y}\}$
- $[\![\text{Asset typeFil } \{\text{assetPatFil}\}]\!]_D =$
  $\{(y, \{y\} \cup M) \mid y \in \mathcal{Y} \wedge (y, \_) \in [\![\text{typeFil}]\!]_{\mathcal{Y}} \wedge (y, M) \in [\![\text{assetPatFil}]\!]_{D,y}\}$

The **boundary-pattern** (boundPat) allows the description of an anti-pattern which examines all boundaries inside the diagram. All boundaries $\mathcal{A}$ inside the diagram D are compared to the defined pattern. The evaluation is similar to that of the element-pattern (elPat).

- $[\![(\text{boundPat}_1 \mid \text{boundPat}_2)]\!]_D = [\![\text{boundPat}_1]\!]_D \cup [\![\text{boundPat}_2]\!]_D$
- $[\![\text{Boundary}]\!]_D = \{(a, \{a\}) \mid a \in \mathcal{A}\}$
- $[\![\text{Boundary typeFil}]\!]_D = \{(a, \{a\}) \mid a \in \mathcal{A} \wedge (a, \_) \in [\![\text{typeFil}]\!]_{\mathcal{A}}\}$
- $[\![\text{Boundary } \{\text{boundPatFil}\}]\!]_D =$
  $\{(a, \{a\} \cup M) \mid a \in \mathcal{A} \wedge (a, M) \in [\![\text{boundPatFil}]\!]_{D,a}\}$
- $[\![\text{Boundary typeFil } \{\text{boundPatFil}\}]\!]_D =$
  $\{(a, \{a\} \cup M) \mid a \in \mathcal{A} \wedge (a, \_) \in [\![\text{typeFil}]\!]_{\mathcal{A}} \wedge (a, M) \in [\![\text{boundPatFil}]\!]_{D,a}\}$

The **connector-pattern** (conPat) allows the description of an anti-pattern which examines all connectors inside the diagram. All connectors $\mathcal{R}$ inside the diagram D are compared to the defined pattern.

- $[\![(\text{conPat}_1 \mid \text{conPat}_2)]\!]_D = [\![\text{conPat}_1]\!]_D \cup [\![\text{conPat}_2]\!]_D$ (1)
- $[\![\text{Connector } (\{\text{srcFil } \& \text{ tgtFil}\}]\!]_D =$ (2)

  $\{(r, \{r\} \cup M_1 \cup M_2) \mid r \in \mathcal{R}$ (3)

  $\wedge \exists n_1, (n_1, M_1) \in [\![\text{srcFil}]\!]_D \wedge \exists n_2, (n_2, M_2) \in [\![\text{tgtFil}]\!]_D$ (4)

  $\wedge\, source(r) = n_1 \wedge target(r) = n_2\}$ (5)
- $[\![\text{Connector typeFil } (\{\text{srcFil } \& \text{ tgtFil}\}]\!]_D =$ (6)

  $\{(r, \{r\} \cup M_1 \cup M_2) \mid r \in \mathcal{R} \wedge (r, \_) \in [\![\text{typeFil}]\!]_{\mathcal{R}}$ (7)

  $\wedge \exists n_1, (n_1, M_1) \in [\![\text{srcFil}]\!]_D \wedge \exists n_2, (n_2, M_2) \in [\![\text{tgtFil}]\!]_D$ (8)

  $\wedge\, source(r) = n_1 \wedge target(r) = n_2\}$ (9)
- $[\![\text{Connector typeFil } (\{\text{srcFil } \& \text{ tgtFil } \& \text{ conPatFil}\}]\!]_D =$ (10)

  $\{(r, \{r\} \cup M_1 \cup M_2 \cup M_3) \mid r \in \mathcal{R} \wedge (r, \_) \in [\![\text{typeFil}]\!]_{\mathcal{R}}$ (11)

  $\wedge \exists n_1, (n_1, M_1) \in [\![\text{srcFil}]\!]_D \wedge \exists n_2, (n_2, M_2) \in [\![\text{tgtFil}]\!]_D$ (12)

  $\wedge\, source(r) = n_1 \wedge target(r) = n_2$ (13)

  $\wedge\, (r, M_3) \in [\![\text{conPatFil}]\!]_{D,r}\}$ (14)

Lines (2-5) describe a connector-pattern without specified type-filter and no additional filters except the mandatory source and target-filter (srcFil,tgtFil). Each resulting tuple consists of the connector r, the union of r, M_1 and M_2, where M_1 is a set that contains the source element n_1, and its affected components and M_2 is a set that contains the target element n_2 and its affected components.
Lines (6-9) describe a connector-pattern variant with a specified type-filter. In addition to the lines (2-5), the connector identifier r has to be inside the resulting set of the type-filter (typeFilter) evaluation.
Lines (10-14) depict a connector-pattern variant with additional connector filters. The context of the filter evaluation is the connector identifier r. The additional affected components are contained in set M_3, which is also part of the union of each resulting tuple.

The **flow-pattern** flowPat allows the description of an anti-pattern which examines a defined flow sequence. In section 2.2.10, the concept of flows has been introduced. The function $\mathsf{flows}(D, src, tgt)$ returns a set of all possible flow sequences between a source element and target element in diagram D.

- $[\![(\text{flowPat}_1 \mid \text{flowPat}_2)]\!]_D = [\![\text{flowPat}_1]\!]_D \cup [\![\text{flowPat}_2]\!]_D$ (1)
- $[\![\text{Flow } \{\text{srcFil \& tgtFil}\}]\!]_D =$ (2)

  $\{(p, \{p\} \cup M_1 \cup M_2) \mid \exists n_1, (n_1, M_1) \in [\![\text{srcFil}]\!]_D$ (3)

  $\wedge \exists n_2, (n_2, M_2) \in [\![\text{tgtFil}]\!]_D \wedge \exists p, p \in flows(D, n_1, n_2)$ (4)

  $\wedge\, pSource(p) = n_1 \wedge pTarget(p) = n_2\}$ (5)
- $[\![\text{Flow } \{\text{srcFil \& tgtFil \& flowPatFil}\}]\!]_D =$ (6)

  $\{(p, \{p\} \cup M_1 \cup M_2 \cup M_3) \mid$ (7)

  $\exists n_1, (n_1, M_1) \in [\![\text{srcFil}]\!]_D$ (8)

  $\wedge \exists n_2, (n_2, M_2) \in [\![\text{tgtFil}]\!]_D \wedge \exists p, p \in flows(D, n_1, n_2)$ (9)

  $\wedge\, pSource(p) = n_1 \wedge pTarget(p) = n_2$ (10)

  $\wedge\, (p, M_3) \in [\![\text{flowPatFil}]\!]_{G,p}\}$ (11)

Lines (2-5) describe a flow-pattern with no additional filter, except the mandatory source and target-filter (srcFil,tgtFil). Each resulting tuple consists of the flow sequence p and the union of p, M_1 and M_2 where M_1 is a set that contains the source element n_1 and its affected components and M_2 is a set that contains the target element n_2 and its affected components. The flow sequence p must exist in the result set of the function $\mathsf{flows}(D, n_1, n_2)$ where n_1 is the source element and n_2 is the target element.
(6-11) This flow-pattern has additional flow-filters. The context of the filter evaluation is the flow sequence p. The additional affected components are contained in the set M_3, which is also part of the union of each resulting tuple.

The type filter (typeFil) is used to examine the diagram components according to their assigned type. This filter can be applied to the element-pattern (elPat), asset-pattern (assetPat), boundary-pattern (boundPat) and connector-pattern (conPat). The evaluation of this filter takes place under the consideration of the special context capital C. This context contains the respective set of diagram component identifiers to which this filter is to be applied according to the previous pattern.

- $[\![\text{: "}q\text{"}]\!]_C = \{(c, \emptyset) \mid \lambda_C(c) = q \vee \lambda_C(c) \in \iota_C(q)\}$ (1)
- $[\![\text{!= "}q\text{"}]\!]_C = \{(c, \emptyset) \mid \lambda_C(c) \neq q \wedge \lambda_C(c) \notin \iota_C(q)\}$ (2)
- $[\![\text{: IN ["}q\text{", } \ldots, \text{"}q_i\text{"}]\!]_C =$ (3)

  $\{(c, \emptyset) \mid \lambda_C(c) \in \{q \ldots q_i\} \vee \lambda_C(c) \in \{\iota_C(q) \ldots \iota_C(q_i)\}\}$ (4)
- $[\![\text{: NOT IN ["}q\text{", } \ldots, \text{"}q_i\text{"}]\!]_C =$ (5)

  $\{(c, \emptyset) \mid \lambda_C(c) \notin \{q \ldots q_i\} \wedge \lambda_C(c) \notin \{\iota_C(q) \ldots \iota_C(q_i)\}\}$ (6)

The context capital C contains either all element, asset, boundary or connector identifiers. The functions λ_C and ι_C consider the set of identifiers specified inside the context capital C. The variable q (or variables $q \ldots q_i$) defines the specified type which should be examined.

(1) The first version of this filter checks whether the assigned type of the diagram component matches the specified type q. Due to the fact that the types are defined in a two level hierarchy and each diagram component has both a "Top-Type" and a "sub-type", the component must either have exactly ($\lambda_C(c)$=q) the specified type (q) or be a "Sub-Type" of it ($\lambda_C(c) \in \iota_C(q)$). The returned set of the evaluation consists of tuples that contain the diagram component c that fulfills the type-filter and the empty-set $\emptyset$ since no additional diagram components can be affected by this filter.
Line (2) displays the negated version of the first version (1). Therefore, the component must not have ($\lambda_C(c) \neq q$) the specified type (q) AND is not allowed to be a "Sub-Type" of it ($\lambda_C(c) \notin \iota_C(q)$).
(3-4) The third version of the type-filter is similar to the first version (1). However, in this case it is examined whether the assigned type of diagram component is contained within a set of possible types ($q_1 \dots q_i$). Therefore, the assigned type must be in the set ($\lambda_C(c) \in \{q \dots q_i\}$), OR it is in the set of the "Sub-Types" ($\lambda_C(c) \in \{\iota_C(q) \dots \iota_C(q_i)\}$).
The last version of this filter (5-6) is the negated form of version three (3-4). Furthermore, it is similar to the second version (2). Therefore, it is defined that the tuples of the resulting set can not contain diagram components that have an assigned type that is included inside the type-filter ($\lambda_C(c) \notin \{q \dots q_i\}$), AND the type must not be in the set of the "Sub-Types" ($\lambda_C(c) \notin \{\iota_C(q) \dots \iota_C(q_i)\}$).

The **source-filter** (srcFil) is used to define the source element of a connector or flow-pattern. The **target-filter** (tgtFil) is used to define the target element.

$$[\![\text{Source elPat}]\!]_D = \{(n, M) \mid n \in N, (n, M) \in [\![\text{elPat}]\!]_D\} \quad (1)$$
$$[\![\text{Target elPat}]\!]_D = \{(n, M) \mid n \in N, (n, M) \in [\![\text{elPat}]\!]_D\} \quad (2)$$

(1-2) The resulting set includes only tuples that contain an element n that conforms the specified element-pattern (elPat), a set M that contains the additional affected components. This applies to the source-filter as well as the target-filter.

The elPatFil term defines the additional filters of an element-pattern. Therefore, the context c can only be an element identifier. Assignable filters are: propFil, assetFil, elRelFil, conFil and flowFil.

- $[\![\text{elPatFil}_1 \ \& \ \text{elPatFil}_2]\!]_{D,c} =$ (1)
  $\{(c, M_1 \cup M_2) \mid (c, M_1) \in [\![\text{elPatFil}_1]\!]_{D,c} \wedge (c, M_2) \in [\![\text{elPatFil}_2]\!]_{D,c}\}$ (2)
- $[\![(\text{elPatFil}_1 \mid \text{elPatFil}_2)]\!]_{D,c} = [\![\text{elPatFil}_1]\!]_{D,c} \cup [\![\text{elPatFil}_2]\!]_{D,c}$ (3)

(1-2) If a pattern has multiple additional filters that are linked by a logical AND (&), each one of them must be fulfilled by the component in the context c. If one of the terms returns an empty set, the end result is the empty set. If all terms return a non-empty set, the end result is the cross-product of all the results, meaning all combinations of affected elements (M_1 and M_2) are returned.

(3) If the filters are linked by a logical OR (|), only one of them has to be met by the component in the context c. The resulting set is the union of all evaluation sets as each entry of the evaluations independently fulfills the filter.

The assetPatFil term defines the additional filters of an asset-pattern. Therefore, the context c can only be an asset identifier. The only assignable filter is the property-filter (propFil).

- $[\![\text{assetPatFil}_1 \ \& \ \text{assetPatFil}_2]\!]_{D,c} =$ (1)

  $\{(c, M_1 \cup M_2) \mid (c, M_1) \in [\![\text{assetPatFil}_1]\!]_{D,c} \wedge (c, M_2) \in [\![\text{assetPatFil}_2]\!]_{D,c}\}$ (2)
- $[\![(\text{assetPatFil}_1 \mid \text{assetPatFil}_2)]\!]_{D,c} = [\![\text{assetPatFil}_1]\!]_{D,c} \cup [\![\text{assetPatFil}_2]\!]_{D,c}$ (3)

The boundPatFil term defines the additional filters of a boundary-pattern. Therefore, the context c can only be a boundary identifier. The only assignable filter is the boundary-relation-filter boundRelFil.

- $[\![\text{boundPatFil}_1 \ \& \ \text{boundPatFil}_2]\!]_{D,c} =$ (1)

  $\{(c, M_1 \cup M_2) \mid (c, M_1) \in [\![\text{boundPatFil}_1]\!]_{D,c} \wedge (c, M_2) \in [\![\text{boundPatFil}_2]\!]_{D,c}\}$ (2)
- $[\![(\text{boundPatFil}_1 \mid \text{boundPatFil}_2)]\!]_{D,c} = [\![\text{boundPatFil}_1]\!]_{D,c} \cup [\![\text{boundPatFil}_2]\!]_{D,c}$ (3)

The conPatFil term defines the additional filters of a connector-pattern. Therefore, the context c can only be a connector. Assignable filters are: propFil, assetFil, conCrossesFil.

- $[\![\text{conPatFil}_1 \ \& \ \text{conPatFil}_2]\!]_{D,c} =$ (1)

  $\{(c, M_1 \cup M_2) \mid (c, M_1) \in [\![\text{conPatFil}_1]\!]_{D,c} \wedge (c, M_2) \in [\![\text{conPatFil}_2]\!]_{D,c}\}$ (2)
- $[\![(\text{conPatFil}_1 \mid \text{conPatFil}_2)]\!]_{D,c} = [\![\text{conPatFil}_1]\!]_{D,c} \cup [\![\text{conPatFil}_2]\!]_{D,c}$ (3)

The flowPatFil term defines the additional filters of a flow-pattern (flowPat). Therefore, the context c can only be a flow (p). Assignable filters are: flowCrossesFil, includesFil.

- $[\![\text{flowPatFil}_1 \ \& \ \text{flowPatFil}_2]\!]_{D,c} =$ (1)

  $\{(c, M_1 \cup M_2) \mid (c, M_1) \in [\![\text{flowPatFil}_1]\!]_{D,c} \wedge (c, M_2) \in [\![\text{flowPatFil}_2]\!]_{D,c}\}$ (2)
- $[\![(\text{flowPatFil}_1 \mid \text{flowPatFil}_2)]\!]_{D,c} = [\![\text{flowPatFil}_1]\!]_{D,c} \cup [\![\text{flowPatFil}_2]\!]_{D,c}$ (3)

The **property-filter** (propFil) investigates the assigned properties of the component given by the context c. The context can either be an element, an asset or a connector identifier.

$$\bullet \ [\![\text{"}k\text{" = "}v\text{"}]\!]_{D,c} = \begin{cases} \{(c,\{c\})\} & \text{if } \mu(c,k) \equiv v \\ \emptyset & \text{if } \mu(c,k) \not\equiv v \vee \mu(c,k) = \textit{undefined} \end{cases} \quad (1)$$

$$\bullet \ [\![\text{"}k\text{" != "}v\text{"}]\!]_{D,c} = \begin{cases} \{(c,\{c\})\} & \text{if } \mu(c,k) \not\equiv v \\ \emptyset & \text{if } \mu(c,k) \equiv v \vee \mu(c,k) = \textit{undefined} \end{cases} \quad (2)$$

$$\bullet \ [\![\text{"}k\text{" in ["}v_1\text{", } \ldots \text{"}v_i\text{"]}]\!]_{D,c} = \quad (3)$$

$$\begin{cases} \{(c,\{c\})\} & \text{if } \mu(c,k) \in \{v_1, \ldots v_i\} \\ \emptyset & \text{if } \mu(c,k) \notin \{v_1, \ldots v_i\} \vee \mu(c,k) = \textit{undefined} \end{cases} \quad (4)$$

$$\bullet \ [\![\text{"}k\text{" in ["}v_1\text{", } \ldots \text{"}v_i\text{"]}]\!]_{D,c} = \quad (5)$$

$$\begin{cases} \{(c,\{c\})\} & \text{if } \mu(c,k) \notin \{v_1, \ldots v_i\} \\ \emptyset & \text{if } \mu(c,k) \in \{v_1, \ldots v_i\} \vee \mu(c,k) = \textit{undefined} \end{cases} \quad (6)$$

(1) Returns the provided context c if the function $\mu(c,k)$ returns the same value as defined in v. If the result of the function $\mu(c,k)$ does not match the value v, the empty set is returned. Since the function μ is a partial function, it is also possible that the property key k is not defined for the component in context c and is therefore *undefined*. If k is *undefined* for the context c, the empty set is returned.
(2) Returns the provided context c if the function $\mu(c,k)$ does not return the same value as defined in v or the partial function is not defined for the property k and the context c. An empty set is returned if the result of $\mu(c,k)$ matches the value v.
(3-4) Is similar to the first variant of the property-filter, except that in this case a set of possible values $(v_1 \ldots v_i)$ is specified. The context c is returned if the result of the partial function $\mu(c,k)$ is in the set of values or is *undefined*. The empty set is returned if the result of the function $\mu(c,k)$ is not in the values $(v_1 \ldots v_i)$.
(5-6) Represents the negated form of the variant from lines (3-4). In this case, the context c is returned if the result of the partial function $\mu(c,k)$ is not within the set of the specified values or it is *undefined*. An empty set is returned if the result of $\mu(c,k)$ is part of the values set $v_1 \ldots v_i$.

The **asset-filter** (assetFil) examines whether an element or connector is linked to an asset defined by the asset-pattern (assetPat). This filter can only be assigned to an element or a connector-pattern. For this reason, context c can only contain one element or connector identifier.

$$\bullet \ [\![\text{Holds assetPat}]\!]_{D,c} = \{(c,M) \mid \exists y, (y,M) \in [\![\text{assetPat}]\!]_D \wedge (c,y) \in \rho\} \quad (1)$$

(1) The diagram D contains asset y, which corresponds to the specified asset-pattern (assetPat). The relation ρ contains all relationships between elements, connectors and assets. Therefore, the tuple (c,y) must be contained in ρ for this filter to apply. The resulting set includes only tuples that contain the context c that is related to the asset y, a set M that contains the affected asset.

The **element-relation-filter** elRelFil examines the relations of an element component with other elements and boundaries. This filter can only be applied to an element-pattern (elPat).

Therefore, the context c can only be an element identifier n.

- $[\![\text{Contains elPat}]\!]_{D,c} = \{(c, M) \mid \exists n, (n, M) \in [\![\text{elPat}]\!]_D \wedge (c, n) \in \delta\}$ (1)
- $[\![\text{Contains no elPat}]\!]_{D,c} = \{(c, \{c\}) \mid [\![\text{Contains elPat}]\!]_{D,c} = \emptyset\}$ (2)
- $[\![\text{Contained by elPat}]\!]_{D,c} = \{(c, M) \mid \exists n, (n, M) \in [\![\text{elPat}]\!]_D \wedge (n, c) \in \delta\}$ (3)
- $[\![\text{Not Contained by elPat}]\!]_{D,c} = \{(c, \{c\}) \mid [\![\text{Contained By elPat}]\!]_{D,c} = \emptyset\}$ (4)
- $[\![\text{Contained by boundPat}]\!]_{D,c} = \{(c, M) \mid \exists a, (a, M) \in [\![\text{boundPat}]\!]_D \wedge (a, c) \in \kappa\}$ (5)
- $[\![\text{Not Contained by boundPat}]\!]_{D,c} = \{(c, \{c\}) \mid [\![\text{Contained By boundPat}]\!]_{D,c} = \emptyset\}$ (6)

(1) Examines if the element in context c contains another element n. This element has to conform to the defined element-pattern (elPat), and the tuple (c, n) must be part of the relation δ which defines all element-element relations. The resulting tuples contain the context c, the set M which contains the element n, and the additional affected diagram components.
(2) Represents the negated form of (1). Therefore, the resulting set of (1) has to be the empty set. As a non-existing element cannot be part of the result, only the element identifier in context c is returned.
(3) Examines if the element in context c is contained by another element n. This element has to conform the defined element-pattern (elPat) and the tuple (n, c) must be part of the relation δ.
(4) Represents the negated form of (3). Therefore, the resulting set of (3) is not allowed to contain a tuple which includes the context c.
(5) Checks if the element in context c is contained by a boundary a. This boundary has to conform to the defined boundary-pattern (boundPat), and the tuple (a, c) must be part of the relation κ which defines all boundary-element relations. The resulting tuples contain the context c, the set M which contains the boundary a, and the additional affected diagram components.
(6) Represents the negated form of (5). Therefore, the resulting set of (5) has to be the empty set.

The **boundary-relation-filter** boundRelFil examines the relations of an boundary component with other elements and boundaries. This filter can only be applied to an boundary-pattern (boundPat). Therefore, the context c can only be a boundary identifier a.

- $[\![\text{Contains elPat}]\!]_{D,c} = \{(c, M) \mid \exists n, (n, M) \in [\![\text{elPat}]\!]_D \wedge (c, n) \in \kappa\}$ (1)
- $[\![\text{Contains no elPat}]\!]_{D,c} = \{(c, \{c\}) \mid [\![\text{Contains elPat}]\!]_{D,c} = \emptyset\}$ (2)
- $[\![\text{Contains boundPat}]\!]_{D,c} = \{(c, M) \mid \exists a, (a, M) \in [\![\text{boundPat}]\!]_D \wedge (c, a) \in \kappa\}$ (3)
- $[\![\text{Contains no boundPat}]\!]_{D,c} = \{(c, \{c\}) \mid [\![\text{Contains boundPat}]\!]_{D,c} = \emptyset\}$ (4)
- $[\![\text{Contained by boundPat}]\!]_{D,c} = \{(c, M) \mid \exists a, (a, M) \in [\![\text{boundPat}]\!]_D \wedge (a, c) \in \kappa\}$ (5)
- $[\![\text{Not Contained by boundPat}]\!]_{D,c} = \{(c, \{c\}) \mid [\![\text{Contained by boundPat}]\!]_{D,c} = \emptyset\}$ (6)

(1-6) The evaluation of this filter is similar to the previously presented element-relation-filter (elRelFil), except that the relationships of a boundary are examined.

The **connector-filter** (conFil) examines the incoming and outgoing connections of an element. This filter can only be applied to an element-pattern (elPat). Therefore, the context c can only be an element identifier n.

- $[\![\text{Has Connector (typeFil)? \{srcFil \& conPatFil\}}]\!]_{D,c} =$ (1)

  $\{(c, \{r\} \cup M_1 \cup M_2) \mid r \in \mathcal{R} \wedge \exists n_1, (n_1, M_1) \in [\![\text{srcFil}]\!]_D$ (2)

  $\wedge\ (\text{typeFil} = \text{nil} \vee (r, \_) \in \text{typeFil}_{\mathcal{T}})$ (3)

  $\wedge\ source(r) = n_1 \wedge target(r) = c\ \wedge (r, M_2) \in [\![\text{conPatFil}]\!]_{D,r}\}$ (4)
- $[\![\text{Has Connector (typeFil)? \{tgtFil \& conPatFil\}}]\!]_{D,c} =$ (5)

  $\{(c, \{r\} \cup M_1 \cup M_2) \mid r \in \mathcal{R} \wedge \exists n_1, (n_1, M_1) \in [\![\text{tgtFil}]\!]_D$ (6)

  $\wedge\ (\text{typeFil} = \text{nil} \vee (r, \_) \in \text{typeFil}_{\mathcal{T}})$ (7)

  $\wedge\ source(r) = c \wedge target(r) = n_1 \wedge (r, M_2) \in [\![\text{conPatFil}]\!]_{D,r}\}$ (8)
- $[\![\text{Has No Connector (typeFil)? \{srcFil \& conPatFil\}}]\!]_{D,c} =$ (9)

  $\{(c, \{c\}) \mid [\![\text{Has Connector (typeFil)? \{srcFil \& conPatFil\}}]\!]_{D,c} = \emptyset\}$ (10)
- $[\![\text{Has No Connector (typeFil)? \{tgtFil \& conPatFil\}}]\!]_{D,c} =$ (11)

  $\{(c, \{c\}) \mid [\![\text{Has Connector (typeFil)? \{tgtFil \& conPatFil\}}]\!]_{D,c} = \emptyset\}$ (12)

(1-4) Examines an incoming connector r since the connector-filter contains a source-filter (srcFil). Each result tuple consists of the context c and the union of r, M_1 and M_2 where M_1 is a set that contains the source element n_1 and the affected diagram components. The additional affected components of the connector-filter (conPatFil) are contained in the set M_2.
Lines (5-8) are similar to (1-4), except that in this case, an outgoing connector is evaluated since a target-filter (tgtFil) is specified.
(9-10) Represents the negated form of (1-4). Therefore, the resulting set of (1-4) has to be the empty set. As a non-existing connector cannot be part of the result, only the element identifier in context c is returned.
(11-12) Represents the negated form of (5-8).

The **flow-filter** (flowFil) examines the incoming and outgoing flows of an element. This filter can only be applied to an element-pattern (elPat). Therefore, the context c can only be an element identifier n.

$$
\begin{aligned}
&\bullet\ [\![\text{Has Flow } \{\text{srcFil \& flowPatFil}\}]\!]_{D,c} = && (1)\\
&\quad \{(c, \{p\} \cup M_1 \cup M_2) \mid \wedge \exists n_1, (n_1, M_1) \in [\![\text{srcFil}]\!]_D && (2)\\
&\qquad \wedge \exists p, p \in flows(D, n_1, c) \wedge (p, M_2) \in [\![\text{flowPatFil}]\!]_{D,p}\} && (3)\\
&\bullet\ [\![\text{Has Flow } \{\text{tgtFil \& flowPatFil}\}]\!]_{D,c} = && (4)\\
&\quad \{(c, \{p\} \cup M_1 \cup M_2) \mid \wedge \exists n_1, (n_1, M_1) \in [\![\text{tgtFil}]\!]_D && (5)\\
&\qquad \wedge \exists p, p \in flows(D, c, n_2) \wedge (p, M_2) \in [\![\text{flowPatFil}]\!]_{D,p}\} && (6)\\
&\bullet\ [\![\text{Has No Flow } \{\text{srcFil \& flowPatFil}\}]\!]_{D,c} = && (7)\\
&\quad \{(c, \{c\}) \mid [\![\text{Has Flow } \{\text{srcFil \& flowPatFil}\}]\!]_{D,c} = \emptyset\} && (8)\\
&\bullet\ [\![\text{Has No Flow } \{\text{tgtFil \& flowPatFil}\}]\!]_{D,c} = && (9)\\
&\quad \{(c, \{c\}) \mid [\![\text{Has Flow } \{\text{tgtFil \& flowPatFil}\}]\!]_{D,c} = \emptyset\} && (10)
\end{aligned}
$$

Lines (1-3) describe a flow-filter, which examines an incoming flow sequence p, as the flow-filter contains a source-filter (srcFil). The function $\mathsf{flows}(D, n_1, c)$ returns a set of flows between the source element n_1 and the target element in the context c. Each result tuple consists of the context c and the union of p, M_1 and M_2 where M_1 is a set that contains the source element n_1 and the affected diagram components. The additional affected components of the flow-filter (flowPatFil) are contained in the set M_2.
Lines (4-6) are similar to (1-4), except that in this case, an outgoing flow is evaluated since a target-filter (tgtFil) is specified. The function $\mathsf{flows}(D, c, n_2)$ returns a set of flows between the source element in the context c and the target element n_2.
Lines (7-8) represent the negated form of (1-4). Therefore, the resulting set of (1-4) has to be the empty set. As a non-existing flow sequence cannot be part of the result, only the element identifier in context c is returned.
Lines (9-10) represent the negated form of (5-8).

The **connector-crosses-filter** (conCrossesFil) examines whether a connector crosses a boundary or an element. This filter can only be applied to a connector-pattern (conPat) or a connector-filter (conFil). Therefore, the context c can only be a connector identifier r.

$$
\begin{aligned}
&\bullet\ [\![\text{Crosses elPat}]\!]_{D,c} = && (1)\\
&\quad \{(c, M) \mid \exists n, (n, M) \in [\![\text{elPat}]\!]_D && (2)\\
&\qquad \wedge (n, source(c)) \in \delta \neq (n, target(c)) \in \delta\} && (3)\\
&\bullet\ [\![\text{Crosses boundPat}]\!]_{D,c} = && (4)\\
&\quad \{(c, M) \mid \exists a, (a, M) \in [\![\text{boundPat}]\!]_D && (5)\\
&\qquad \wedge (a, source(c)) \in \kappa \neq (a, target(c) \in \kappa)) && (6)
\end{aligned}
$$

Lines (1-3) represent the first variant of this filter, which examines whether the connector in the context c crosses an element n. The element n has to conform to the specified element-pattern (elPat). The relation δ contains all element relations inside the diagram D. A connector r crosses the element if either the source element or the target element of the connector is contained by n, and the other is not contained by n. The resulting tuples contain the context c, a set M which contains the crossed element, the additional affected elements.
Lines (4-6) depict the second variant of this filter, which examines whether the connector in the context c crosses a boundary a. The boundary a has to conform to the specified boundary-pattern (boundPat). The relation κ contains all boundary-element relations inside the diagram D. A connector r crosses the element if either the source element or the target element of the connector is contained by a, and the other is not contained by a.

The **flow-crosses-filter** (flowCrossesFil) examines whether any connector in the flow sequence crosses a boundary or an element. This filter can only be applied to a flow-pattern (flowPat) or a flow-filter (flowFil). Therefore, the context c can only be a flow sequence p.

$$
\begin{aligned}
&\bullet\ [\![\text{Crosses elPat}]\!]_{D,c} = && (1)\\
&\quad \{(c, M) \mid \exists n, (n, M) \in [\![\text{elPat}]\!]_D \wedge \exists r \in connectors(c) && (2)\\
&\qquad \wedge\ (n, source(r)) \in \delta \neq (n, target(r)) \in \delta && (3)\\
&\bullet\ [\![\text{Crosses boundPat}]\!]_{D,c} = && (4)\\
&\quad \{(c, M) \mid \exists a, (a, M) \in [\![\text{boundPat}]\!]_D \wedge \exists r \in connectors(c) && (5)\\
&\qquad \wedge (a, source(r)) \in \kappa \neq (a, target(r)) \in \kappa\} && (6)
\end{aligned}
$$

A flow can consist of several connectors. The flow-crosses-filter is used to check whether any of the connectors crosses a boundary or element. The function $connectors(p)$ returns a set of connector identifiers, which contains all connectors in the flow sequence p.

Lines (1-4) represent the first variant of this filter, which examines whether any connector r in the flow sequence p crosses an element n. The element n has to conform to the specified element-pattern (elPat). The evaluation of whether a connector crosses an element is similar to the connector-crosses-filter.

(5-8) The second variant of this filter examines whether any connector r in the flow sequence p crosses a boundary a. The boundary a has to conform to the specified boundary-pattern (boundPat). The evaluation of whether a connector crosses a boundary is similar to the connector-crosses-filter.

The **includes-filter** (includesFil) examines the elements or connectors in a flow sequence. This filter can only be applied to a flow-pattern (flowPat) or a flow-filter (flowFil). Therefore, the context c can only be a flow sequence p.

$$\bullet\ [\![\text{Includes elPat}]\!]_{D,c} = \tag{1}$$

$$\{(c, M) \mid \exists n, (n, M) \in [\![\text{elPat}]\!]_D \wedge n \in elements(c)\} \tag{2}$$

$$\bullet\ [\![\text{Includes no elPat}]\!]_{D,c} = \{(c, \{c\}) \mid [\![\text{Includes elPat}]\!]_{D,c} = \emptyset\} \tag{3}$$

$$\bullet\ [\![\text{Includes only elPat}]\!]_{D,c} = \tag{4}$$

$$\{(c, M_1 \cup \dots M_i) \mid \{n_1, \dots n_i\} = elements(c), \tag{5}$$

$$(n_j, M_j) \in [\![\text{elPat}]\!]_D,\ \forall 1 \leq j \leq i\ \} \tag{6}$$

$$\bullet\ [\![\text{Includes conPat}]\!]_{D,c} = \tag{7}$$

$$\{(c, M) \mid \exists r, (r, M) \in [\![\text{conPat}]\!]_D \wedge r \in connectors(c)\} \tag{8}$$

$$\bullet\ [\![\text{Includes no conPat}]\!]_{D,c} = \{(c, \{c\}) \mid [\![\text{Includes conPat}]\!]_{D,c} = \emptyset\} \tag{9}$$

$$\bullet\ [\![\text{Includes only conPat}]\!]_{D,c} = \tag{10}$$

$$\{(c, \{M_1 \cup \dots M_i\}) \mid \{r_1, \dots r_i\} = connectors(c), \tag{11}$$

$$(r_j, M_j) \in [\![\text{conPat}]\!]_D,\ \forall 1 \leq j \leq i\ \} \tag{12}$$

Lines (1-2) represent the first variant of the includes-filter, which examines whether a flow sequence p contains an element n. The element n has to conform to the specified element-pattern (elPat). The returned tuples contain the context c, a set M that contains the additional affected components of the element-pattern.
(3) This filter variant represents the negated form of the first variant (1-2). Therefore, the resulting set of (1-2) has to be the empty set. As a non-existing element cannot be part of the result, only context c is returned.
(4-6) The third variant of the filter examines whether all elements $(n_1 \dots n_i)$ in the a flow sequence p conform to a specified element-pattern (elPat).
Lines (7-8) depict an includes-filter which examines whether a flow sequence p contains a connector r. The connector r has to conform to the specified connector-pattern (conPat).
Lines (9) represent the negated form of the filter in lines (8-9).
(10-12) The last filter variant examines whether all connectors $(r_1 \dots r_i)$ in the a flow sequence p conform to a specified connector-pattern (conPat).

2.3.2 The Flow-finding Algorithm

Section 2.2.10 introduced the concept of the flows. The function flows(D, src, tgt) is used to find all possible flows between a source element src and a target element tgt within a diagram D. The function is used within the flow-pattern (flowPat) and the flow-filter (flowFil). By definition, a flow consists of an alternating sequence of elements and connectors. The following algorithm describes the implementation of the function flows(D, src, tgt).

Algorithm 1 The Flow-finding Algorithm

```
1:  –Initialization –
2:  Input: Diagram (D)
3:  Input: Source Element (src)
4:  Input: Target Element (tgt)
5:  Output: Set of Flows (flowSet)
6:  procedure FLOWS(D, src, tgt)
7:      flowSet ← ∅
8:      usedConnectors ← ∅
9:      RECURSIVESEARCH(src,tgt,usedConnectors,flowSet,D)
10:     return flowSet
11: end procedure

12: procedure RECURSIVESEARCH(src, tgt, usedConnectors, flowSet)
13:     if |usedConnectors| > 0 then
14:         if src ≡ tgt then
15:             p ← createSeq(usedConnectors)
16:             flowset add p
17:             return
18:         end if
19:     end if
20:     srcConnectors ← connectors(D, src)
21:     for each connector ∈ srcConnectors do
22:         if connector ∉ usedConnectors then
23:             newSrc ← target(connector)
24:             usedConnectors add connector
25:             RECURSIVESEARCH(newSrc,tgt,usedConnectors,flowSet,D)
26:             usedConnectors remove connector
27:         end if
28:     end for
29: end procedure
```

The algorithm implements the depth-first search (DFS) approach. The analysis of an element is paused as soon as the next element has been identified (Mukhopadhyay et al., 2016, p.48).

The lines (1-5) describe the initialization of the function. As already mentioned, the flows function takes three arguments: a diagram instance (D), a source element (src), and a target element (tgt). The function returns a set of flows as output.
Lines (6-11) describe the initial call of the function flows(D, src, tgt). Line (7) initializes a new $flowSet$ is as an empty set. (8) A flow is an alternating sequence of elements and connectors. However, it is defined that each connector in a flow sequence must be unique. Therefore, the algorithm has to remember which connectors have already been visited in order to avoid endless loops. $usedConnectors$ is an initially empty set that stores the visited connectors. Line (9) shows the call of the function recursiveSearch. This function takes five parameters as input: source element, target element, the set of already visited connectors ($usedConnectors$), the $flowSet$, and the diagram instance D. The arguments are passed as references to the function. (10) The $flowSet$ is the return/output value of the algorithm as it contains the end result of the algorithm.
The recursive function $recursiveSearch$ is depicted in lines (12-29). (13) If the function is called the first time, the set of the $usedConnectors$ is empty. A flow must contain at least one connector, and therefore, the number of visited connectors ($usedConnectors$) must be greater than zero.
The lines (14-18) depict the end of a recursive call if the source element is equivalent to the target element. (15) If the source element is equivalent to the target element, a flow has been found. The set of $usedConnectors$ contains all connectors which are part of the flow p. The function createSeq($usedConnectors$) converts the set of connectors to a valid flow sequence. (16) The flow sequence is added to the $flowSet$. Line (17) depicts the end of a recursive call.
(20) If the $usedConnectors$ set does not contain any connectors or the current source element does not match the target element, the algorithm must continue. The function srcConnectors(D, src) returns a set of connectors whose source element is the current src element. These are stored inside the variable $srcConnectors$.
Lines (21-28) describe the search for the next source element for the next recursive iteration. (21-22) For each $connector$ within the $srcConnectors$ set, it is checked whether it has already been visited. If not, then the target element of this connector is the new source element ($newSrc$) for the next iterative call. (24) The $connector$ is added to the set of $usedConnectors$. Line (25) depicts the call of the next recursive iteration. (26) The $connector$ is removed from the $usedConnectors$ as soon as the recursive call returns.

3 Related Research/Tools

In this section, we look at the concept behind security analyses of IT-systems. To this end, we present the most common strategies, methods, existing tools and related research in this area.

3.1 Threat Modeling and Risk Management

Before describing the analysis approach of ThreatGet, we introduce the concept of threat modeling. An adequate definition of the threat modeling concept at the high level is the following: *"Threat modeling is the first step in any security solution. It's a way to start making sense of the vulnerability landscape. What are the real threats against the system? If you do not know that, how do you know what kind of countermeasures to employ?"* (Schneier, 2000, p.214). According to this quote by Schneier, identifying vulnerabilities and threats using an adequate threat modeling methodology is essential to a successful risk management process (Schneier, 2000). Consequently, threat modeling is a key subpoint in the larger process of risk management, as only threats and vulnerabilities that have already been identified can be addressed.

Risk management is not a straightforward process that ends after a single execution but should be performed iteratively to respond to new threats, information, and changes (Schmittner et al., 2019). To make the entire process as effective as possible, it is also important to involve as many stakeholders as possible and to take a holistic view of it.
Over time, three different threat modeling strategies have emerged. Most existing methods can be assigned to at least one of these strategies. However, there are also methods and frameworks that combine all strategic approaches, such as the "Hybrid Threat Modeling Method" (hTMM) by Mead, Shull, Vemuru, and Villadsen (2018). In the following, we present the strategies and their representatives and show which of them has been implemented in ThreatGet.

3.2 The Attacker-Centric Strategy

The focus of this strategy, as the name implies, is on the person behind the attack. The goal of this strategy is to identify the attacker archetype (Shevchenko et al., 2016). Therefore, the analyst tries to put himself in the position of the attacker and thus simulate the activities. For instance, the motivation, the background and the necessary knowledge for an attack are analyzed. A well-known method of this strategy is Persona-non-Grata (PnG). The main process of this method is to collect the information and document it on index cards. For this strategy, the approach and information extraction are difficult to automate because they are based on rather subjective and non-technical procedures. Nevertheless, this approach is used to get a first impression or to initiate the threat modeling process in general (Mead et al., 2018).

3.3 The Asset-Centric Strategy

The focus is on the assets of an organization or system. In broad terms, an asset denotes an object of interest or value. The initial process of an asset-centric strategy is to identify all assets of a given system Eng. It can be helpful to analyze what might be of interest to an attacker and, in particular, what is essential to the company or organization. Based on this, it is analyzed how an attack could damage the assets or how an aggressor could access the assets (Shostack, 2014). A well-known method is the CORAS framework, which defines a graphical modeling language for this purpose (den Braber et al., 2007). A major drawback of this approach is that the set

of assets can contain a large number of entries, depending on the number of stakeholders and the size of the system. In order to avoid scalability issues, the entries must be reviewed and ranked by importance. Subsequently, possible attacks and damage scenarios are worked out for all assets, starting with the most important ones. Similar to the attacker-centric strategy, most processes in this strategy are based on subjective rather than objective approaches. In particular, the identification and valuation of assets depends to a large extent on the stakeholders involved. In addition, the attack scenarios and damage scenarios are based on the experience of the analysts involved (Eng, 2017; Poller et al., 2014; Shevchenko et al., 2016). However, it makes sense to think about what data assets are in a system and how they could be compromised, as they can provide information about the impact and likelihood of an attack. For this reason, ThreatGet allows the user to highlight assets in its diagram and consider them in the analysis.

3.4 The Software-Centric Strategy

The primary focus of this approach is on models that represent the underlying architecture of a system and its operating software. The straightforward focus on system components facilitates involving stakeholders in the security analysis process. Designers and developers do not typically think in terms of business goals or assets. Therefore, it is easier to involve them in a process that relates directly to what they create (Shostack, 2014; Gumbley, 2020). Another positive aspect is that this approach can already be used during the planning phase of a project, as initial conceptual designs of the system can already be made there, which can be used for security analysis. Furthermore, existing systems can also be analyzed with the help of this process.

Different diagram types can be used to represent the systems, e.g., those from the standard "Unified Modeling Language" (UML) or "Data Flow Diagrams" (DFD). DFDs are often used because they have a simpler structure and are therefore easier to design. Section 3.4.1 takes a closer look at the data-flow diagram since most threat modeling methods are based on this diagram, and ThreatGet also uses an extended form of this diagram.. However, the decision for a diagram type should depend on the preferences of the developers.

The best-known threat modeling method that can be assigned to this strategy is STRIDE. This method was developed back in 1999 by Loren Kohnfelder, and Praerit Garg (Kohnfelder & Garg, 1999). The name STRIDE is an acronym composed of the first letters of the STRIDE categories: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege. The STRIDE method is intended as a mnemonic and threat categorization to assist the user in the security analysis of a system under consideration (Shostack, 2014).

The method can be performed in two ways. The first approach is called "STRIDE-per-element" and focuses on each element within the system. The analyst iterates over each element and examines it for all known threats within the STRIDE categories. This approach is very simple and can be performed by a relatively inexperienced person.
The second approach is called "STRIDE-per-Interaction" and focuses on the analysis of the data exchange between two components. The procedure here is similar to the first way, but in this case, more variables are involved in the process, as both the elements and the transmission path

of data are analyzed. *"STRIDE-per-element is a simplified approach to identifying threats, designed to be easily understood by the beginner. However, in reality, threats don't show up in a vacuum. They show up in the interactions of the system."* (Shostack, 2014, p.80).

3.4.1 The Data-Flow Diagram

Most of the tools that aim to automate security analysis, which we present in Section 3.5 and Section 3.6, are based on a STRIDE guided approach and a DFD. A data flow diagram is composed of a total of five different components, which are listed and specified below:

- **External Entity**: Describes a component that is not directly part of the system, but can interact with it.
- **Data Store**: Represents some form of data storage. For example a database or file.
- **Process**: Describes an action that is executed within a system.
- **Data-Flow**: Represents a potential data exchange between to components.
- **Boundary**: Describes a distinction between logically or physically separated system components. In addition, boundaries can be defined to represent multiple privilege levels within the system. They are not an actual part of the system, but can be modeled to represent an attack surface.(Shostack, 2014, p.49-50).

Except for boundaries, properties can be assigned to all diagram components. Properties are defined in the form of key-value pairs and can provide additional information about the specific component.
By focusing on what is being developed and taking a structured approach to analyzing individual system components and their interaction possibilities, the software-centric strategy creates the best conditions for automating security analysis.
Due to this fact, several tools have already been developed that focus on assisting with analysis or implement an automated approach. Four of these tools have been studied prior to the development of ThreatGet, and we present them in the following two sections.
We divide these tools into two categories because three of each are very similar in their approach to analysis.

3.5 Template-based Analysis Tools

The first group of software-centric analysis tools includes two of the total five, and we have classified these as template-based analysis or supporting tools. Both tools use the data flow diagram presented above to represent the system under investigation. They provide an interface in which the system can be modeled and modified, using predefined diagram component templates. We have placed these tools in this group because they also show the user which threats could occur during the analysis. This requires the user to define all potential threats in a template-based format and enter them into the database. As soon as the analysis is started, the user must work

manually through the list of threats according to the "STRIDE-per-Element" and "STRIDE-per-Flow" principles and decide which of the entries may or may not apply.
ThreatGet uses an approach comparable to the tools from the second group. However, these are also presented here as they can make a non-negligible contribution to improving the security posture of systems.

The first tool we want to highlight is **Threat Dragon**. It is an open-source threat modeling tool developed by the Open Web Application Security Project (OWASP)[2]. Its interface provides a non-modifiable list of diagram component templates that can be used to create a data flow diagram. Every template, except for the boundary, comes with its own but not extendable set of properties. Using the tool is relatively straightforward, as creating the diagram and defining threats is quite simple. In addition to the text-based definition of the threats, the user can add severity and mitigation information to each threat entry. During the analysis, the tool supports the user in selecting threats by proposing threat entries based on the diagram component and its assigned properties.

The second tool is called **IriusRisk**. It is a commercial tool designed and developed by Continuum Security[3]. This tool also uses a template-based approach but offers a much more comprehensive functions compared to Threat Dragon. In contrast to Threat Dragon, the user can freely define individual diagram components and use them for system-modeling. Furthermore, it is also possible to extend the set of assigned properties of the template with user-defined properties. The IriusRisk threat template consists of a textual description of the threat and additional information, such as an assessment of exploitability and potential impact. The defined threat templates can be linked to the component templates. The underlying concept of this approach is that systems are typically built with similar components. Especially when an organization reuses the same components for different products, therefore, once defined, the components and their threats can be reused in multiple diagrams. IriusRisk presents the user with the associated threats for a specific component or data flow during the analysis. This reduces the time needed. However, the user still has to decide which of the threats apply manually.

3.6 Logic Analysis Tools

The second group of software-centric analysis tools includes the remaining two tools. We classified them as logic-based analysis tools, as they implement additional functionality to determine whether a defined threat is present in a given system.
For this purpose, all three use a specific analysis language with their own syntax and semantic, defined in a context-free grammar (CFG). Such an analytical language allows threats not only to be described in plain text, but also to be interpreted by a machine. The formulation of the threat in such a language is denoted in the following as an anti-pattern. An anti-pattern describes a state or condition in a system that can lead to a particular threat.

[2] `https://docs.threatdragon.org/`
[3] `https://iriusrisk.com/`

The analysis languages consist of a set of specified sentences. These sentences are each composed of individually defined symbols. The grammar of the language gives the order of the sentences and symbols. *"A context-free grammar (CFG) consists of a set of production rules. Each rule describes how a non-terminal symbol can be "replaced" or "expanded" by a string that consists of non-terminal symbols and terminal symbols."* (Casanova, 2016, p.2). Whereby a terminal symbol represents a fixed part of the language and a non-terminal symbol must be replaced by the given production rule until a terminal symbol is reached.
During the analysis, the tools iterate over the defined threat catalog, interpret the anti-patterns and automatically check whether the declared threat is present in the system under investigation. Due to this approach, only those threats are displayed that actually apply to the system. This significantly reduces the time required for threat identification. The time saved can be used more effectively for the evaluation and mitigation of the identified threats. In addition, non-sophisticated security analysts can perform this procedure, learn from it, and react directly to identified threats. ThreatGet also uses this approach and extends the functionality of the tools we are highlighting in this section.

3.6.1 pyTM

The first tool we want to discuss in this group is **pyTM**. It is open-source tool which is also created an maintained by OWASP[4]. Compared to the other tools, pyTM differs significantly in its approach, as it does not provide a graphical diagram editor. The system-model, i.e. the diagram, is defined in the python programming language. Components, their internal connections, and properties are defined in individual lines of code and added to the diagram object. Based on this code, pyTM provides the functionality to generate a visual diagram. In addition to the data-flow diagram presented in a previous section, it is also possible to create diagrams from the UML specification.
The tool provides a predefined library of diagram components and properties, which the user can extend. Furthermore, the tool contains a catalog of threats that can also be extended. To manage the threats, the tool uses the JavaScript Object Notation (JSON) data format. The user can not only describe individual threats textually but also define an assessment of the potential impact and likelihood. Furthermore, descriptions regarding severity, necessary preconditions and potential mitigation options can be stored. However, this data is not included in the automatic analysis, but the user is shown this information when a threat is detected. The most important entry, which contains the anti-pattern in pyTM, is called "condition".
The condition of a threat states the logical semantics for the evaluated component. The user can provide additional description of required circumstances for this threat, such as whether a component has a particular property, whether it is in a boundary, or whether it is connected to another component. These individual anti-patterns can also be logically linked to each other. Thus it can be stated whether both conditions must take place or only one of them.
Unfortunately, we could not find a complete definition of the full analysis language, as well as its evaluation. The GitHub account[5] includes some predefined examples. Furthermore, pyTM

[4] `https://owasp.org/www-project-pytm/`
[5] `https://github.com/izar/pytm/`

is a purely console-based tool and does not provide a graphical user interface (GUI). Only the generated diagrams and the analysis result are presented to the user. Therefore, it is difficult to use for people without programming skills, especially during the diagram and threat definition.

3.6.2 The Microsoft Threat Modeling Tool

The Microsoft Threat Modelling Tool (MTMT) is probably the best-known threat modeling tool. It is developed as part of Microsoft's Security Development Lifecycle[6]. The tool is primarily used to implement the STRIDE-per-interaction approach. It includes a catalog of diagram components that can be used to develop a data-flow diagram in the tool's own modeling interface. The components, as well as the defined threat entries, are organized in templates that can be grouped for use in specific domains.

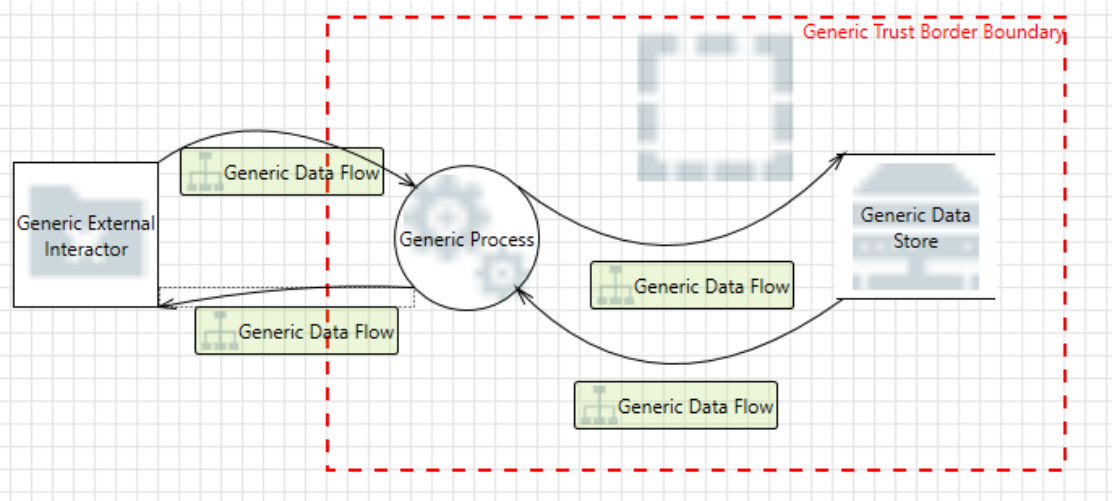

Figure 5: Microsoft Microsoft Threat Modeling Too Diagram

Figure 5 shows a rudimentary example diagram created using the MTMT. It shows three elements, each connected by data flows. This diagram is intended to represent the retrieval of data from a database by a cell phone. The left element, called the "Generic External Interactor," lies outside the "Generic Trust Boundary" and represents the cell phone. Inside the trust boundary lies a "Generic Process" that handles the query and the "Generic Data Store" containing the data. The flow could be as follows, the interaction sends a request to the process, which validates the request, retrieves the data, and returns it. The boundary represents the actual system, which can be accessed from outside. With this example, we want to show how easy it is to map such facts in a DFD.

To define threats with the MTMT, the user must define them in the analysis language provided by the tool. The grammar of this syntax is displayed in the following. The terminal symbols of the grammar are highlighted in red and non-terminals in blue.

[6] https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling

rule	::= expression (op expression)$*$
expression	::= object (.[literal])? IS value
	| Flow crosses literal | NOT expression
object	::= Source | Target | Flow
value	::= YES | NO | literal
op	::= AND | OR
literal	::= "$text$"

Figure 6: Syntax of MTMT Language

Further information to this grammar and its usage can be found in the Getting Started Guide (Microsoft, 2016a) and the User Guide (Microsoft, 2016b). However, these documents provide no detailed description about the semantic evaluation during the analysis. Using this grammar, the following anti-patterns can be formulated, for example:

- Source.["$SecureBoot$"] IS "NO" (1)
- Target.["$Authentication$"] IS "NO" (2)
- Source.["$SecureBoot$"] IS "NO" AND
  Target.["$Authentication$"] IS "NO" AND Flow crosses "$Boundary$" (3) (4)

(1) The first anti-pattern checks whether some element that has an outgoing data-flow has the property "Secure Boot" set to "NO".
(2) The second anti-pattern checks whether some element that has an incoming data-flow has the property "Authentication" set to "NO".
(3-4) The last anti-pattern combines the previous ones and also checks, in addition, whether the data-flow between these two elements "crosses" a "boundary". In this example, the anti-pattern can be used to check if the mobile (Generic External Interactor) inside the boundary can communicate with the process without authenticating itself. This circumstance could mean that an attacker could send malicious data to our process and thus damage the system. A special feature of this analysis language is that it is possible to check whether a boundary is crossed, as these can also represent an attack surface.

The previous examples are intended to show how easy it is to create a system-model with the MTMT and analyze it using the anti-patterns. However, we have found a few limitations of the tool during testing. The MTMT only detects strict anti-patterns during analysis. That is, if the system model is not modeled exactly as described in the pattern, then the anti-pattern does not kick in. This fact is especially apparent when different people model the same system and then different results are displayed during analysis, even though the models are nearly indistinguishable. This drawback indicates that the modeling of the system and the definition of the threats are too closely related (Karahasanovic, Kleberger, & Almgren, 2017).

4 Conclusion

IT-systems have changed significantly over the last two decades, and it is hard to imagine most economic sectors and most daily processes without them (Jang-Jaccard & Nepal, 2014; Chen, 2019). The development towards increasingly interconnected devices is far from complete. It becomes even more apparent when we look at the current trends in the automotive sector and the Internet of Things.

In addition to the positive contributions and improvements that this development has brought, these connected systems are potentially more vulnerable to malicious attacks due to their openly accessible communication interfaces (Wolf, 2018). This circumstance illustrates the importance of IT-Security. Nevertheless, many manufacturers neglect the security factor to save costs and compete on the free market (Gilchrist, 2017).
Given the future mandatory security standards, the associated accountability, and the increasing awareness of customers regarding data security and integrity, these developers are under pressure to produce not only smart but also more secure devices. Since there will probably not be enough security experts to cover all domains and areas in the near future, there is a high demand for an automated solution to detect security vulnerabilities and threats.

For this reason, the Dependable Systems Engineering Group (DSE) at the Austrian Institute of Technology (AIT) has developed the security analysis tool ThreatGet. The tool allows users to automatically scan even complex IT-systems for potential vulnerabilities and threats in a consistent and repeatable way.
In this paper, we present the ThreatGet approach in detail. It is based on three essential components. The first component is an advanced data-flow diagram to represent a system under investigation and its security-related properties. In Section 2.2.2, we present the modeling notation, as well as the complete formal definition of the diagram, using a basic example.
The second component is a threat and vulnerability knowledge database. In order to create and manage this threat knowledge base, a domain-specific analysis language has been developed that can be interpreted by both humans and machines. The language is used to define vulnerability and threat information in the form of anti-patterns. Each anti-pattern describes a potentially exploitable state or condition in the system that must be considered to create a secure product. The full syntax of the language is highlighted in Section 2.2.9.
The third component is an automated analysis engine that takes the other components as input and compares them. The result of the analysis contains all the anti-patterns that are in the system. Section 2.3.1 contains the full semantic evaluation of the anti-patterns, formulated in the analysis language. The semantic analysis allows each user to clearly understand how the analysis is performed and how the results are derived.
This approach is similar to other system-centric approaches such as the Microsoft Threat Modeling Tool but extends existing solutions in all aspects.

ThreatGet is designed to improve the security posture of systems without the need for a security expert. However, it is not feasible to assume that one system can be totally secured. New security exploits are discovered on an almost daily basis. Nevertheless, a system has to be de-

signed and developed to be as secure as possible. Therefore, the best approach to achieve this is to examine the system for known weaknesses and implement suitable security measures to defend it from potential threats.
The analysis approach of ThreatGet provides an automated approach towards vulnerability and threat identification. It further extends the approaches of existing solutions. Furthermore, it features a persistent approach concerning the management of threat knowledge.

5 Future Work / Outlook

In this section, we discuss how we will extend the ThreatGet tool itself and the analysis language for formulating anti-patterns. The "ThreatGet" tool is being further developed in collaboration with industry partners, private customers, and public projects. Since the tool is already commercially distributed in partnership with Lieber Lieber, the development is not entirely independent of external requirements. Nevertheless, the following two features have been identified and will be integrated into the development process.

The first future development focuses on the integration of threat dependencies during the analysis. Currently, individual threats can be mapped and analyzed with the analysis language. However, threat and vulnerability dependencies cannot yet be fully integrated into the analysis. As described initially, attacks rarely consist of single actions that lead the attacker to the target. In most cases, several attack steps that build upon each other must be conducted in order to achieve the actual goal (Navarro et al., 2018). These individual attack steps together form a more complex attack path that describes the entire attack. In order to map such an attack path with the help of the analysis language, it is extended by the concept of pre-and post-conditions (Lallie et al., 2020). This concept allows the analyst to depict the requirements as well as the resulting effects of a threat in an intuitive way (Aksu et al., 2018). For example, conditions such as the required proximity of the attacker, or required knowledge can be integrated into the analysis and refine the analysis result.
This concept has already been evaluated in the master thesis of Korbinian Christl, in cooperation with the University of Vienna.

The second extension of the ThreatGet tool concerns the system-model used, i.e., the diagram used to represent the system under investigation in an abstracted form. As mentioned in Section 3, different diagram models and approaches can be used to represent the security-related and security-critical aspects of a system. The currently used "Advanced Data-Flow Diagram" (ADFD) provides an easy entry point to modeling such a system due to its simplicity and clear structure (Shostack, 2014). However, we have found in several use-case studies that supporting additional diagram models would improve the efficacy of the tool.
Therefore, we evaluate the support of the so-called "Internal Block Diagram" according to the "SysML" standard. This approach has the advantage that this type of diagram is often used to plan and IT-system (Schaad & Borozdin, 2012).
Support for multiple diagram types reduces the additional effort required by the user to transfer their system to the ADFD schema. Adapting the system-model would not require any change to the analysis language. Nevertheless, the semantic analysis may need to be adapted to the data model of the diagram.

References

Aksu, M. U., Bicakci, K., Dilek, M. H., Ozbayoglu, A. M., & Tatli, E. (2018). Automated Generation of Attack Graphs Using NVD. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy - CODASPY '18* (pp. 135–142). Tempe, AZ, USA: ACM Press. Retrieved 2020-04-24, from http://dl.acm.org/citation.cfm?doid=3176258.3176339 doi: 10.1145/3176258.3176339

Black, P. E. (2007). *Total function* [From Dictionary of Algorithms and Data Structures]. Retrieved 2021-03-09, from https://xlinux.nist.gov/dads/HTML/totalfunc.html (Publisher: National Institute of Standards and Technology)

Black, P. E. (2019). *Partial function* [From Dictionary of Algorithms and Data Structures]. Retrieved 2021-03-09, from https://www.nist.gov/dads/HTML/partialfunct.html (Publisher: National Institute of Standards and Technology)

Casanova, H. (2016). *Syntactic Analysis.* Retrieved 2020-12-07, from http://courses.ics.hawaii.edu/ReviewICS312/morea/Compiling/ics312_parsing.pdf

Chen, C. (2019, September). With Great Abstraction Comes Great Responsibility: Sealing the Microservices Attack Surface. In *2019 IEEE Cybersecurity Development (SecDev)* (pp. 144–144).

den Braber, F., Hogganvik, I., Lund, M. S., Stølen, K., & Vraalsen, F. (2007, January). Model-based security analysis in seven steps — a guided tour to the CORAS method. *BT Technology Journal*, *25*(1), 101–117. Retrieved 2020-04-25, from http://link.springer.com/10.1007/s10550-007-0013-9 doi: 10.1007/s10550-007-0013-9

Eng, D. (2017). *Integrated Threat Modelling.* Reprosentralen, University of Oslo. Retrieved from http://www.duo.uio.no/

Gilchrist, A. (2017). *IoT Security Issues*. Walter de Gruyter GmbH & Co KG. (Google-Books-ID: xipDDgAAQBAJ)

Gumbley, J. (2020). *A Guide to Threat Modelling for Developers.* Retrieved 2021-02-20, from https://martinfowler.com/articles/agile-threat-modelling.html

Hussain, S., Kamal, A., Ahmad, S., Rasool, G., & Iqbal, S. (2014). THREAT MODELLING METHODOLOGIES: A SURVEY. In (p. 3).

ISO/SAE. (2020). *ISO/SAE DIS 21434 Road vehicles — Cybersecurity engineering*. ISO - International Standardization Organization. Retrieved 2020-05-01, from https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/07/09/70918.html

Jang-Jaccard, J., & Nepal, S. (2014, August). A survey of emerging threats in cybersecurity. *Journal of Computer and System Sciences*, *80*(5), 973–993. Retrieved 2020-04-25, from https://linkinghub.elsevier.com/retrieve/pii/S0022000014000178

Karahasanovic, A., Kleberger, P., & Almgren, M. (2017). *Adapting Threat Modeling Methods for the Automotive Industry.*

Kohnfelder, L., & Garg, P. (1999). The threats to our products. , 8.

Krisper, M., Dobaj, J., & Macher, G. (2020). Assessing Risk Estimations for Cyber-Security Using Expert Judgment. In M. Yilmaz, J. Niemann, P. Clarke, & R. Messnarz (Eds.), *Systems, Software and Services Process Improvement* (Vol. 1251, pp. 120–134). Cham: Springer International Publishing. Retrieved 2020-11-26, from http://link.springer.com/10.1007/978-3-030-56441-4_9 (Series Title: Communications in Computer and Information Science)

Lallie, H. S., Debattista, K., & Bal, J. (2020, February). A review of attack graph and attack tree visual syntax in cyber security. *Computer Science Review*, *35*, 100219. Retrieved 2020-04-25, from https://linkinghub.elsevier.com/retrieve/pii/S1574013719300772

Mead, N., Shull, F., Vemuru, K., & Villadsen, O. (2018). A Hybrid Threat Modeling Method. *Carnegie Mellon University*, 53.

Microsoft. (2016a). *Microsoft Threat Modeling Tool Getting Started Guide.* Microsoft Trustworthy Computing.

Microsoft. (2016b). *Microsoft Threat Modeling Tool User Guide.* Microsoft Trustworthy Computing.

Miller, D. C., & Valasek, C. (2015). *Remote Exploitation of an Unaltered Passenger Vehicle.*

MITRE, C. (2019, November). *CVE - Home.* Retrieved 2020-08-05, from https://cve.mitre.org/about/index.html

MITRE, C. (2020a). *CAPEC - Common Attack Pattern Enumeration and Classification (CAPEC).* Retrieved 2020-05-05, from https://capec.mitre.org/

MITRE, C. (2020b, February). *CWE - About - CWE Overview.* Retrieved 2020-08-05, from https://cwe.mitre.org/about/index.html

Mos, M. A., & Chowdhury, M. M. (2020, July). The Growing Influence of Ransomware. In *2020 IEEE International Conference on Electro Information Technology (EIT)* (pp. 643–647). (ISSN: 2154-0373)

Mukhopadhyay, S., Agarwal, R., Patel, D., Som, K., & Sarkar, A. (2016). Complexity Based on Traversal of Graphs. , *02*(12), 7.

Navarro, J., Deruyver, A., & Parrend, P. (2018, July). A systematic survey on multi-step attack detection. *Computers & Security*, *76*, 214–249. Retrieved 2020-04-24, from https://linkinghub.elsevier.com/retrieve/pii/S0167404818302141

NVD. (2020). *NVD - General Information.* Retrieved 2020-10-05, from https://nvd.nist.gov/general

Poller, A., Türpe, S., & Kinder-Kurlanda, K. (2014). An Asset to Security Modeling?: Analyzing Stakeholder Collaborations Instead of Threats to Assets. In *Proceedings of the 2014 workshop on New Security Paradigms Workshop - NSPW '14* (pp. 69–82). Victoria, British Columbia, Canada: ACM Press. Retrieved 2020-11-27, from http://dl.acm.org/citation.cfm?doid=2683467.2683474 doi: 10.1145/2683467.2683474

Schaad, A., & Borozdin, M. (2012). TAM 2: automated threat analysis. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12* (p. 1103). Trento, Italy: ACM Press. Retrieved 2020-04-20, from http://dl.acm.org/citation.cfm?doid=2245276.2231950 doi: 10.1145/2245276.2231950

Schmittner, C., Dobaj, J., Macher, G., & Brenner, E. (2020, March). A Preliminary View on Automotive Cyber Security Management Systems. In *2020 Design, Automation Test in*

Europe Conference Exhibition (DATE) (pp. 1634–1639). (ISSN: 1558-1101)

Schmittner, C., Tummeltshammer, P., Hofbauer, D., Shaaban, A., Meidlinger, M., Tauber, M., ... Brandstetter, M. (2019, January). Threat Modeling in the Railway Domain. In (pp. 261–271).

Schneier, B. (2000). Threat Modeling and Risk Assessment. In H. Bäumler (Ed.), *E-Privacy: Datenschutz im Internet* (pp. 214–229). Wiesbaden: Vieweg+Teubner Verlag. Retrieved 2020-10-16, from https://doi.org/10.1007/978-3-322-89183-9_20

Shevchenko, N., Chick, T. A., O'Riordan, P., Scanlon, T. P., & Woody, C. (2016). Threat Modeling: A Summary of Available Methods. In (p. 26).

Shostack, A. (2014). *Threat modeling: designing for security*. Indianapolis, IN: Wiley. (OCLC: 855043351)

Weisstein, E. W. (2021a). *Antisymmetric relation* [From MathWorld–A Wolfram Web Resource]. Retrieved 2021-03-09, from https://mathworld.wolfram.com/AntisymmetricRelation.html (Publisher: Wolfram Research, Inc.)

Weisstein, E. W. (2021b). *Irreflexive* [From MathWorld–A Wolfram Web Resource]. Retrieved 2021-03-09, from https://mathworld.wolfram.com/Irreflexive.html (Publisher: Wolfram Research, Inc.)

Weisstein, E. W. (2021c). *Power set* [From MathWorld–A Wolfram Web Resource]. Retrieved 2021-03-09, from https://mathworld.wolfram.com/PowerSet.html (Publisher: Wolfram Research, Inc.)

Weisstein, E. W. (2021d). *Transitive* [From MathWorld–A Wolfram Web Resource]. Retrieved 2021-03-09, from https://mathworld.wolfram.com/Transitive.html (Publisher: Wolfram Research, Inc.)

Wolf, M. (2018). Combining Safety and Security Threat Modeling to Improve Automotive Penetration Testing. , 143.