Date: February 2022

Software Fault Pattern Metamodel (SFPM)

Version 1.0 – Beta 2

OMG Document Number: ptc/22-02-11

Normative reference: https://www.omg.org/spec/SFPM/1.0/PDF

Normative Machine Consumable files:

https://www.omg.org/spec/SFPM/20220201/SFPM.xml

OMG's Issue Reporting Procedure

Table of Contents

Table of Figures

Preface

About the Object Management Group

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Meta-model); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *https://www.omg.org/*.

OMG Specifications

As noted, OMG specifications address middleware, modeling and vertical domain frameworks. All OMG Formal Specifications are available from this URL: *https://www.omg.org/spec*

All of OMG‟s formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
109 Highland Avenue
Suite 300
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO/IEC standards. Please consult: http://www.iso.org

Issues

The reader is encouraged to report and technical or editing issues/problems with this specification to:
https://www.omg.org

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

`Courier - 10 pt. Bold:` Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

NOTE: Terms that appear in italics are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

1 Scope

One of the key steps in preventing cyber attacks is to collect, analyze and efficiently manage knowledge about *exploitable weaknesses.* This knowledge should be made available to the community as a resource to build more comprehensive prevention, detection and mitigation solutions. To this end, several classifications of weaknesses have been developed; the Common Weakness Enumeration (CWE) catalog describes a large collection of weaknesses building upon proposals by various researchers; however, all existing classifications remain informal and resist automation.

This document describes the Software Fault Pattern (SFP) approach to building machine-consumable knowledge of software weaknesses. The goal of the SFP approach is not to study weaknesses as some abstract objects, but instead to examine computations that exhibit certain "faults"; to reveal the invariants of such computations, and to provide a framework for describing and cataloguing "faults" in terms of these invariants. Invariants of computations determine certain characteristic elements of computations and common "patterns" in the flow of participating computations. Invariants also describe certain logical relations between the characteristic elements of computations. The key benefit of the SFP approach is that invariants of computations can be directly correlated with semantic descriptions of software. To describe invariants in terms of software, the SFP approach uses ISO/OMG Knowledge Discovery Metamodel (KDM) as a language-neutral, vendor-independent vocabulary for describing software facts. With KDM as the foundation, the SFP framework developed an apparatus for formally specifying invariants of computations, and describing and cataloguing faults as invariants of computations. The SFP apparatus involves the specification of the SFP Metamodel (SFPM) and the SFPM XMI schema. As the foundation of the SFP Catalog of Software Fault Patterns – collection of reusable, machine-consumable units of knowledge, the SFP Metamodel defines an infrastructure for new capabilities in software assurance. SFPM XMI is a common interoperable format for representing machine-consumable content related to software faults, their formal semantics and their mappings to the elements of the Common Weakness Enumeration (CWE) catalog.

1.1 SFP and CWE

CWE catalog has been selected as the “reference” of the SFP Catalog since it is a de facto body of community’s knowledge of software weaknesses. Objectives of the SFP program are complimentary to those of CWE. SFP emphasizes machine-consumable/formal definitions of semantics of weaknesses, focusing on the invariants, while CWE emphasizes the breadth of knowledge about weaknesses and human-consumable content. Developing SFP content is an important to build a better ecosystem of tools and services. SFP content can be available through cross-links from CWEs and vice versa.

The objective of SFP is to provide a semantic “viewpoint” on the content that is already in CWE, to provide a set of *formal compliance points* for software weaknesses as well as to resolve any inconsistencies and ambiguities in existing CWE content and fill any gaps in CWE.

A *formal compliance point* for a software weakness provides a rigorous, automatable way to address such questions as a) whether a certain code fragment is an example of a given software weakness, and b) whether a certain tool can detect a given software weakness. Existence of formal compliance points for individual "named" weaknesses – items of the CWE Catalog - has significant benefits in removing ambiguities in weakness findings reporting and development of new evidence collection capabilities for digital certification of systems.

A formal compliance points are particularly important for the industry of code analysis tools. In this context, formal compliance points can be used analytically (by comparing an implementation to a formal definition of a weakness); synthetically (by generating compliance test cases from the formal definition of a weakness) or constructively (by developing a content-driven code analysis tool, and importing formal definitions of weaknesses).

SFP addresses a certain important subset of software weaknesses in CWE – weaknesses that are fully discernible/described as properties of code. This class can be called discernible white-box code weaknesses.

The terms "weakness", "flaw", "bug" and "vulnerability" are often used inconsistently because the objects implied by these terms lack constructive formal definitions. This specification uses the term "software fault" as it refers to an identified – adjudged or hypothesized – cause of a failure of the service performed by a piece of software under investigation. Correct service is delivered when the service implements the system function. A service failure, often abbreviated to failure, is an event that occurs when the delivered service deviates from correct service. A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function. Further, the SFP apparatus is developed to provide formal, constructive definitions to the class of software faults that can be identified in the software alone. SFP metamodel as defined by this specification, is the machine-consumable representation of these formal definition of software faults. Further, the context in which SFP has been developed is system assurance, risk management and digital certification of systems. Consequentially, the class of software faults of interest for the SFP catalog is related to cybersecurity failures.

From this perspective, the CWE catalog has a broader scope as the CWE "weaknesses" can be attributed to artifacts other than software. CWE "weaknesses" are not necessarily "discernible either.

1.2 SFP Applications

Formal machine-consumable descriptions of software weaknesses are instrumental to establish an ecosystem of new capabilities that will consume the SFP content and use this content for various purposes including (but not limited to)

- producing analytics related to software faults (visualizations, reports, identifying gaps, etc.),
- collecting evidence for digital certification of systems (identifying instances of weaknesses in code and binary, proving absence of certain classes of weaknesses, etc.),

- synthesizing test cases for code analysis tools (measuring performance of code analysis tools, calibrating reporting capabilities in tools, etc.),
- digital certification of systems (unambiguous and precise mapping of classes of weaknesses to risks, communicating requirements for evidence collection between risk management tools and code analysis tools, communicating evidence findings from code analysis tools to risk management tools, etc.).

1.3 SFP Apparatus

The primary objective of the SFP Catalog is to bring clarity and precision to the study of weaknesses by building a systematic machine-consumable catalog of software faults and to enable analytics and automation of various workflows involving the knowledge of software faults. To this end, the SFP approach brings together several successful model-based techniques:

- ISO/OMG Knowledge Discovery Metamodel (KDM) as a language-neutral, vendor-independent vocabulary for describing software facts;
- community best practices for machine-consumable descriptions of software faults as data flows;
- ISO/OMG Semantics of Business Vocabularies and Rules (SBVR) as a logical foundation for formal definitions of logical propositions on top of vocabularies such as KDM, and
- Meta-Object Facility (MOF) as the foundation for building technology-neutral representations.

The resulting apparatus allows structured definitions of semantics of software faults, development of vendor-neutral content, accumulation of reusable content, analytics, and development of new capabilities.

SFPM defines a set of elements to describe denotational semantics of highly specialized objects – software weaknesses, as dataflows. These definitions are independent of the implementation language and thus are not related to syntax, structure of pragmatics of the programming language. In contrast, SFP describes the semantics of the dataflows as invariants of computations. These definitions are made modular, so that the resulting catalog can easily organize common clauses, and reference them to the CWE items, which are considered as signifiers of the software weaknesses.

By providing a semantic definition of otherwise very informally defined CWE items, SFP program achieves its objective of providing formal compliance points to CWEs.

The elements of meaning in SFPM have the following 4 layers:

- KDM elements define the meaning of the code elements. This is defined in the KDM specification. KDM definitions do not involve full denotational semantics, but instead are defined in reference to known programming languages.
- Common Logic Statements that use KDM vocabulary (as well as the common SBVR vocabulary). To increase the readability of the definitions, SFP allows referencing any

formally defined vocabulary based on KDM. So, common clauses can be arranged into vocabularies and referenced from other clauses.

- Semantic of a dataflow. This is defined informally by referencing to the well-known program analysis literature. SFPM includes several structural elements of a dataflow that receive their separate definitions as common logic statements. Thus, every dataflow is defined as its source, sink and data element, as wells as a global condition. Structural apparatus of SFPM allows managing individual modular semantic clauses and cross referencing them.
- Cross-referencing SFPs to enumerations of common causes and impacts.

The SFP Catalog provides a structured semantic approach to the enumeration of *software faults*. A *software fault* is a situation that manifests itself as a *faulty computation* exhibited by a *system*. The rest of this section defines the scope of the SFP Catalog in more detail by reviewing the background of the SFP approach.

The SFP approach borrows methods and apparatuses of program analysis to describe related families of computations *independent* of the existing code. The discipline of program analysis deals with various representations of the computations implemented by a given *code* in the larger context of an entire *system* under analysis. The purpose of SFP content is to be consumed by some program analysis capabilities that would use it, for example, as executable rules to prove that the code under analysis has faults described by the SFP. The SFP content exists *independently* of the corresponding capability and independent of any code under analysis. The SFP Metamodel provides guidelines to the coordinated development of the SFP content, and the capabilities that will use this content. It is important that the SFP content be defined as both technology-neutral and vendor-neutral, i.e. not assuming a specific capability that can utilize it.

1.3.1 Semantics of Dataflows

SFP Metamodel involves an apparatus for defining *invariants* of *data flows*. The main purpose of this apparatus is to define *computations* that exhibit certain *faults* (vulnerabilities, weaknesses). *Vulnerability* is defined as “a bug, flaw, weakness, or exposure of an application, system, device, or service that could lead to a failure of confidentiality, integrity, or availability”. In other words, “vulnerability” is a computation that can be exploited to produce (negative) impact. Certain computations in the system are designed to *mitigate* vulnerabilities. These computations and the corresponding mechanisms and “places” in the code are called “*safeguards*”. A “*faulty computatio*n” is defined as either a *computation* that has direct negative *impact* on the operations of the *system*, or a computation that corresponds to an incorrectly implemented security *safeguard*. The catalog of *faulty computations* focuses at *computations* that are common to large families of *systems*.

Certain computations are specific to a single system. However, there are certain computations that are common to large families of systems. For example, such common computations are related to input processing, authentication, access control, cryptography, information output, resource management, memory buffer management, exception management. To focus on the invariants of such common computations, further *abstractions* of the basic concept of a computation may need to be considered.

In general, a *computation* is a sequence of *events* performed by a system. The idea of formally describing invariants of computations using an *alphabet* of event names was outlined in C.A.R. Hoar "Communicating Sequential Processes". The choice of an alphabet usually involves a deliberate simplification, a decision to ignore many other properties and actions which are considered to be of lesser interest. This specification uses the word *computation* to stand for the behavior invariant of a system, insofar as it can be described in terms of the limited set of events selected as its *alphabet*. This first alphabet is referred to as the *observable alphabet* of a computation.

Events must be implemented by some *activities* which introduce another *alphabet* related to the computation. This second alphabet is referred to as the *activity alphabet* of the computation. *Activities* are implemented by the *code* and supported by other components of the *system* such as hardware, firmware, networks, operators, etc. For a *system implemented mostly in software*, the *code* provides the constraints to computations and therefore determines what computations can occur. An *activity* corresponds to a certain identifiable *place* in the code (represented by some artifacts) - also referred to as a *program point*. For example, a source code in C language is represented by one or more text files containing function definitions and statements – these files are referred to as "artifacts". An activity in this case corresponds to one or more statements, and its place can be described in terms of a region of line numbers in the source file(s), as well as in terms of function(s) owning the statement(s). Activities are usually defined at the semantic level (referrerd to as micro-KDM operations in [kdm]), so a "line of code" taken at syntax level usually corresponds to multiple activities and thus – multiple program points. Program points introduce the third and final alphabet of a computation – its *program point alphabet*. While activity defines a specific semantic micro operation, for example assigning a value to a pointer, a program point refers to a specific position in the control- and data-flow (same activity can happen at many different program points).
The fundamental decision of the SFP Catalog approach is to use the vocabulary defined by the KDM standard as *that activity alphabet* and *the program point alphabet* for defining *computations* as vendor-neutral content. This establishes a language-neutral foundation (SFP content can be mapped to the syntax of different programming languages) as well as a vendor-neutral foundation (SFP content is not expressed against some proprietary internal representation of a tool by some vendor).

When describing *vulnerabilities*, a larger context of *system* is important. An entire *system* is a collection of *activities* that *exchange data* to achieve some desired purpose. *Activities* occur at system *nodes* that are connected by *channels*. A *system node* is implemented by some *code.* A *channel* is an abstraction to represent *data exchanges* between *activities* owned by two *nodes*. Following the NIST Common Vulnerability Scoring System (CVSS) approach, we distinguish *local channels* between system *nodes* deployed at the same *machine* (host); *adjacent network channels* between system *nodes* deployed at the same local area network; and *remote channels*. This distinction is important because it determines the class of *access* required to exploit *vulnerabilities*. Each system *node* performs *activities* to provide *services* to other system *nodes* or the *environment* of the entire *system*. Thus, the *events* described by *activities* can be mapped to system *nodes*.
Data *exchanges* use *channels*. We distinguish between *data at rest* (for example, data in a database), *data in motion* (data in a channel) and *data in use* (data used by an *activity*).

In program analysis, any serious attempt at characterizing computation exhibited by a system must provide for some sort of account of the computation's structure in terms of one of its alphabets (*observable alphabet*, *activity alphabet* or *program point alphabet)*. Assertions regarding order of activities, location and disposition of transfers, identification of subroutines, internal consistency, as

well as state of the computation in terms of the values of the data elements at any program point, all involve a knowledge of the structure of the *code* under study. The structure of *code* is usually determined by code artifacts describing the program, and may usually be given a convenient geometric representation by means of control- and data- flow graphs.
Thus, a *computation* may traverse multiple *system nodes* and *channels* in the sense that the *activity* events are mapped to a sequence of *nodes* and *channels* involved in *data exchanges* between *activities* at connected *nodes*.
A *trace* of the behavior of a process is a finite sequence of symbols recording the events in which the *computation* has engaged up to some moment in time.
A *trace* records a sequence of *observable events*, *activities or program points*. For a trivial computation, a *trace* provides an adequate description of the computation. Obviously, any non-trivial *system* exhibits an infinite number of *traces*. If one wanted to enumerate representative traces of a certain system as a means of description, shorter traces may be preferred. For example, a system can be described by a finite number of traces corresponding to a single statement (or a *basic block* of statements). The number of larger traces of larger computations would be infinite as there is usually no upper bound imposed on the maximum length of a trace. A more adequate description of the code may be achieved by aggregating the initial single statement traces into longer sequences that are "recurring" throughout various end-to-end computations.

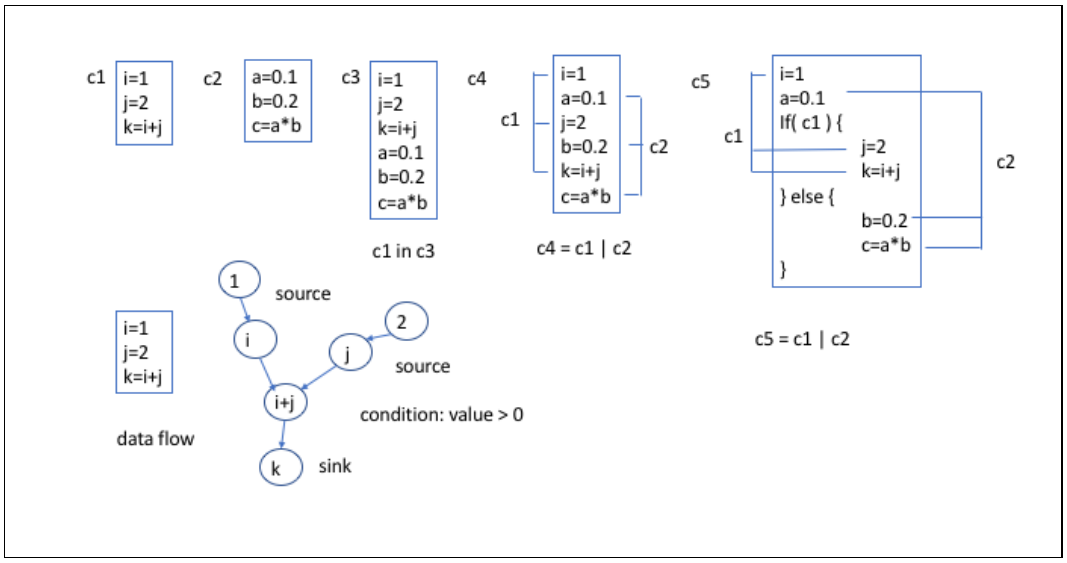

Figure 1. Computations and data flows

Selection of short "recurring" computations as the means of describing complex behavior patterns is important, as computations can be combined and/or interleaved. A (shorter) trace can be part of one or more (larger) traces. Computations can be *interleaved* as follows. Consider two computations, c1 with activities {a1,a2,a3} and c2 with activities {a4,a5,a6}. Computation c1 is *atomic*, if a1 is always followed by a2, and a2 is always followed by a3. The quantification "always" is taken over the set of all end-to-end traces for the code. Computation c1 may be interleaved with computation c2, if c1 is not atomic, and a1 is followed by a4, a4 is followed by a2, etc. This is illustrated at Figure 1.
Further, a useful way of enumerating "recurring" trace segments of a computation is to consider "*data flows*". A computation can be also viewed as a series of transformations of the data *state*, which consists of the *values* of all *data elements* (variables) across all system *nodes*, including data in motion, data in use and data at rest. A *data flow* is a computation that only includes activities that

are related to the state of a single "*data element*". The concept of a data element is essential for imperative programming languages, however, even in the context of non-imperative language, e.g. functional programming, or logic programming, there are data elements, such as formal parameters of functions and logical variables, and therefore, there are data flows to consider. Obviously, data flows are often interleaved between themselves.
A data flow focuses at assigning (or binding) *values* to *data elements*. A data flow can be viewed as a flattened inverted tree of computations that compute the value of the data element at its root (the last element in the computation). This is illustrated at Figure 1.

To focus on the invariants of computations that are common to broad classes of systems, further *abstractions* of the basic concept of a computation may need to be considered. For example, computations c1 and c2 in Figure 1 share common structure, with different names of the variables, types of the variables, data values and expression in the last statement. Both are data flows, where a variable is assigned a value that is the result of an arithmetic expression involving two other variables. Each of the two variables is assigned a constant value. This pattern can be considered an *invariant* of the corresponding data flow. When a formal description of this invariant is available as machine-consumable content, one can develop a generic data-driven capability that will collect evidence related to the presence of such data flows in the code (by enumerating the possible locations in the code), or to synthesize samples of this data flow as tests.
The key part of a data flow is its *sink*. By definition, a data flow has a single *sink*. Further, a data flow may have one or more *sources*. Sink and source(s) are defined as *propositions* that only use the *program point alphabet*. In other words, sink and source(s) are defined in terms of the code constructs (in terms of the KDM standard, in a language-neutral and technology-neutral way). They are not defined in terms of the *values* of the data elements, or in terms of the *state* of the computation. *Source* specification may only describe a statement. As a *source* specification of a data flow, this is an indirect way of specifying the possible range of values of a data element. For example, an assignment statement with constant "NULL" as the right-hand side expression as a single source to a data flow specifies that the value of the data element can only be 'NULL'.
A data flow may involve a characteristic *condition* that involves the *value* at the *sink* – a direct way of specifying ranges of *values*. *Condition* is a powerful way of specifying data flows. *Condition* correlates with the *values* specified by the *source*(s). For example, values {1,2} for the sources satisfy the condition in Figure 1, and so do values {10,20}, but not values {1,-2}.
The SFP approach describes a *sink* of a data flow in terms of the code constructs, in such a way that its *location* in the code can be established. This mechanism can be called a *program point pattern* that is effectively matched to the *code* and identifies certain *program points* as instances of the pattern. The rest of the data flow is described as one or more *logical propositions* the truth of which must be established to make a claim that an occurrence of a data flow is found in the code. The SFP approach assumes a capability that will match the sink program point pattern, and another capability that will keep finding longer and longer data flows leading to the sink, and yet another capability that will check the propositions that describe the invariant. Such capability must eventually make a verdict whether there is enough evidence to claim the presence of the pattern or not. The two latter capabilities must interact to keep extending the data flows, when possible, if no verdict has been made, and to stop, when the evidence becomes inconclusive (when neither verdict can be made).
Condition as a means of specifying invariants of a data flow is a significantly more computationally expensive, compared to more pattern-like propositions involving the *source* values.

1.3.2 Formalization of dataflows in SFP

The formalization approach of the SFP Catalog is based on the following considerations. An invariant of a data flow can be described as a set of facts such that any “compliant” data flow will exhibit these facts, and only compliant data flows will exhibit such facts.

Sink and sources of a data flow are defined using logical expressions built on top of program point patterns. The program point patterns use KDM facts as the base vocabulary. The rest of the logical expression for *sinks* and *sources* uses the first order logical formulations from the Semantics of Business Vocabularies and Rules (SBVR) standard.

The content of the SFP Catalog describes an argument justifying the claim that the code under assessment exhibits a certain fault. The starting point of this argument is the presence of the Indicator. Additional evidence is provided by matching of the elements of the SFP in relation to the Indicator. Final evidence is collected when the data flow satisfies the condition of the SFP.
An invariant of a data flow can be described as a set of propositions such that any “compliant” data flow will exhibit these propositions, and only compliant data flows will exhibit such propositions. Thus, the SFP Catalog accumulates content related to describing “interesting” data flows.

1.3.3 SFP-enabled capabilities

The content of the SFP Catalog can be used for a multitude of purposes, including the three fundamental ones:

1) [certification] How to collect evidence that a certain system under assessment exhibits a given SFP;

2) [synthesis] How to generate representative samples of a given SFP;

3) [analytics] better understanding software weaknesses and their impact on systems, including machine learning and artificial intelligence

From the **certification perspective**, the SFP approach assumes four supporting capabilities:

1) capability to locate certain “places” in the code under assessment;

2) capability to systematically identify data flows that involve a given “place” in the code;

3) capability to check certain conditions on a given data flow;

4) capability to eventually make a verdict whether there is enough evidence to claim the occurrence of a data flow at the given place in the code.

Thus, the evidence collected by this process involves the evidence of an (initial) location of a (possible) SFP, evidence of the identification of the data flows, and evidence to the condition checking.
A computation “*indicator*” is a known construct (such as an entry point, or an API call) manifested in the system’s *artifacts*, such that it is a *necessary condition* for the *computation*. Certain places in

the *code* can directly cause (negative) *impact*. Such places are indicators for the impacting computations. *Safeguards* also have indicators, related to the safeguard itself as well as to the protected region. Thus, a significant part of the SFP Catalog is the enumeration of the unique *places* in the code associated with *faulty computations* that directly have *impact* or to the failed *safeguards*. Indicators are described as program point patterns using KDM vocabulary.

From the **synthesis perspective**, the SFP Catalog accumulates content related to the full context in which an *invariant* of a certain *fault* may occur, as well as the canonical samples of both "compliant" and "non-compliant" data flows. Further, the SFP approach assumes the following capabilities:

1) capability to generate a sample code in selected programming language from a formal description adopted by the SFP Catalog;

2) capability to select a coherent variant of the "compliant" (or "non-compliant") data flow from the formal description provided by the SFP Catalog;

3) capability to extend the code invariant provided by the SFP Catalog with local and global variations (or "code and data complexities") in a systematic way.

CWE already provides many illustrative examples of weaknesses in selected languages. While illustrative examples are important for human consumption, such examples cannot be considered as a useful part of machine-consumable knowledge. Code examples need to be parsed, they do not identify the core parts of the "fault" (not often precise enough to do using the language syntax); they do not provide guidance on true positive/false positive; they are very limited in the code and data complexity and in their language coverage. On the other hand, the industry of code analysis tools requires millions of systematic test cases with appropriate metadata.
The SFP approach separates the knowledge of a "software fault" in the form of dataflow invariants from and "code and data complexities" and the language-specific details. By focusing on the semantics of the dataflow, SFP provides the necessary "scaffolding" that can be used to generate detailed metadata for a test case.
A software fault can be "implemented" (embodied) in an infinite number of ways: different variable names, embedding a faulty computation into various contexts, introducing intermediate fragments to the invariant without changing its semantics and many other ways. Concise specification of the dataflow invariant allows to use the SFP content in a synthesizer/test case generator tool that can introduce systematic code and data variations to the selected dataflow "slice" in a language-independent form.
Language-specific details and variations are addressed by the KDM standard in the form of language-specific mapping to and from KDM.

From the **analytics perspective**, the SFP Catalog accumulates a multitude of reusable, machine-consumable *units of knowledge* that provide semantic denotations to families of faulty computations (as dataflows), and reference them to the signifiers in the CWE catalog. The SFP Catalog has modular organization of the semantic element to facilitate analytics, cross-reference between elements, and reuse. This content facilitates machine learning and cross-referencing various characteristics of software weaknesses, and other artificial intelligence applications.

1.3.4 The role of the SFP Metamodel

The Software Fault Pattern approach involves a certain apparatus for developing semantical definitions of software weaknesses as dataflows, the SFP Metamodel that uses MOF to define the "language" in which the items of the SFP catalog are defined, and the SFP catalog itself.

The SFP Catalog provides a catalog of the *faulty computations*, focuses at the "places" in the code, that are the *indicators* of the corresponding *computations*. Therefore, the Software Fault Pattern approach is driven by the invariants in the code as they determine classes of faulty computations. The items of the SFP Catalog are grouped together into SFP items and further into primary and secondary clusters based on their common indicators, and common impact. This viewpoint is constructive and systematic and therefore enables automation. This uniform viewpoint makes the Software Fault Pattern approach systematic and repeatable.

The SFP Metamodel (SFPM) – the normative part of this specification. SFPM determines the interchange format via the XML Metadata Interchange (XMI) by applying the standard MOF to XMI mapping to the SFPM MOF model. The interchange format defined by SFPM is called the SFPM XMI schema.

SFPM XML (XMI) is a common interoperable format for representing machine-consumable content related to software faults, their formal semantics and their mappings to the elements of the Common Weakness Enumeration (CWE) catalog. SFPM XMI is the foundation for the OMG Catalog of Software Fault Patterns that will over time accumulate formal machine-consumable definitions of individual software faults and other structured content related to software faults. SFPM XMI supports a larger ecosystem of capabilities that need to exchange formal definitions of weaknesses, including but not limited to test generation tools, static code analysis tools, data repositories, machine learning tools, visualization tools, training tools. The SFPM XMI is the canonical format in which this content is available.

This specification describes the SFPM XMI schema and illustrates the usage of the SFPM XMI schema by describing example SFPM XMI data representations compliant with the SFPM XMI schema. To further facilitate development and review of the SFP content, Appendix A of this specification describes a readable textual representation of the SFPM XMI. The specification illustrates SFP Metamodel elements with numerous examples of real SFP content. All examples are provided in SFPM XMI as well as in the readable SFP language. The readable SFP language is not a normative part of the SFPM specification. This notation is a highly-specialized format optimized for the SFP content. By utilizing the OMG MOF ecosystem, the SFP Metamodel allows multitude of other technology-specific representations of the SFP content.

2 Conformance

The principle goal of SFPM is to define a common normalized format for representing machine-consumable content related to software faults, their formal semantics and their mappings to the elements of the Common Weakness Enumeration (CWE) catalog. SFPM is defined via the Meta-

Object Facility (MOF). SFPM determines the interchange format via the XML Metadata Interchange (XMI) by applying the standard MOF to XMI mapping to the SFPM MOF model. The interchange format defined by SFPM is called the SFPM XMI schema.

To be SFP compliant, a document or an implementation (such as a capability, a tool, a repository, a service) shall fully support SFPM as one compliance point. A compliant document shall comply to the SFPM XMI schema. A compliant implementation shall provide either or both of the following:

- The capability to generate XMI documents based on the SFPM XMI schema capturing content in the scope of the SFP Catalog.
- The capability to import and use content via representations based on the SFPM XMI schema.

The "use" of imported SFP content in compliant tools is not limited to one of the use cases described in this specification.

3 References

3.1 Normative References

The following normative documents contain provisions which, through reference in this text, provide normative context for material in this specification.

[kdm] Knowledge Discovery Metamodel (KDM), v1.4, http://www.omg.org/spec/KDM/1.4
[sbvr] Semantics for Business Vocabulary and Rules (SBVR), v1.5, http://www.omg.org/spec/SBVR/1.5/
[uml] Unified Modeling Language (UML), v2.5, http://www.omg.org/spec/UML/2.5
[mof] Meta-Object Facility (MOF), v.2.4.2, http://www.omg.org/spec/MOF/2.4.2
[xmi] XML Metadata Interchange (XMI), v2.5.1, http://www.omg.org/spec/XMI/2.5.1
[xml] Extensible Markup Language, v1.1, http:// http://www.w3.org/TR/xml11
[xsd-1] XML Schema Definition Language (XSD) v1.1 Part 1: Structures, http://www.w3.org/TR/xmlschema11-1
[xsd-2] XML Schema Definition Language (XSD) v1.1 Part 2: Datatypes, http://www.w3.org/TR/xmlschema11-2

[cwe] Common Weakness Enumeration (CWE) – a repository maintained by MITRE Corporation of known weaknesses in software that can be exploited to modify data, read data, create a denial-of-service that results in unreliable execution, create a denial-of-service that results in resource consumption, execute unauthorized code or commands, gain privileges / assume identity, bypass protection mechanism, and/or hide their activities[1]. <https://cwe.mitre.org>.

Also, ITU standard: ITU X.1524 Common Weakness Enumeration < https://www.itu.int/rec/T-REC-X.1524-201203-I/en >

3.2 Informative References

The following non-normative documents contain provisions which, through reference in this text, provide informative context for material in this specification.

[1] CWE technical impact enumeration <https://cwe.mitre.org/cwraf/enum_of_ti.html>

- Software Fault Patterns (SFP) Catalog –
    - AFRL-RY-WP-TR-2012-0111, V2 - DoD document approved for public release, distribution unlimited;
    - Software Fault Pattern Clusters - a repository maintained by MITRE Corporation of links connecting SFPs and CWEs <https://cwe.mitre.org/data/definitions/888.html>

- [NIST CVSS] NISTR Interagency Report 7435 "The Common Vulnerability Scoring System (CVSS) and its applicability to Federal Agencies".

4 Terms and Definitions

This section provides a glossary of terms used by this specification.

Computation	Behavior pattern of a system, insofar as it can be described in terms of the limited set of events selected as its *alphabet*.
Alphabet	A set of basic parts of elements, esp. the set of characters or symbols with which a language is written. An alphabet of a computation can be a set of all events that the computation can exhibit (or a set of all activities, or a set of all program points). An alphabet of a computation is an abstraction to define behavior patterns.
Trace	A trace of the behavior of a computation is a finite sequence of symbols recording the events in which the computation has engaged up to some moment in time.
Control flow	A representation of the order of activities of the computation
Data flow	A *data flow* is a computation that only includes activities that are related to the state of a set of *data elements*.
Data flow invariant	A formal description characterizing multiple possible instances of a data flow implemented as code in a variety of programming languages, runtime support systems, hardware, etc. in a variety of system contexts.
Data element	[KDM] DataElement represents computational objects of a software system that are associated with a value of a particular datatype.
Data flow sink	A proposition describing a final program point of a data flow.
Data flow source	A proposition describing one or more starting program points of a data flow
Data flow condition	A proposition describing some invariant property involving the values of the data elements of a data flow

Indicator	A proposition in the form of a possibly recursive statement in KDM vocabulary that can be effectively matched to the KDM representation of the code under analysis so that the instances of the indicator can be enumerated. In SFP Metamodel, data flow sinks are specified as disjunctions of indicators.
Invariant	A property of all objects in a collection or a family. A "logical invariant" is a certain condition that is true for all objects in a family. A "structural invariant", is a certain fragment that all objects in the family have.
Proposition	A logic statement that uses semantic formulation and terms of KDM vocabulary
Program point	A location in the code described in selected program point alphabet. As the basis for defining some content that is independent of the code under analysis, the foundation for the program point alphabets is KDM. Program points can be defined as complex sets of KDM facts (statements in KDM vocabulary). Such program point alphabet provides a further abstraction on top of KDM vocabulary.
Program point pattern	A proposition describing a program point as some content that is independent of the existence and the nature of the code under analysis.
Weakness	Software weaknesses are situations in software implementation, code, design or architecture that if left unaddressed could result in systems and networks being vulnerable to attack. Weaknesses can be referred to as flaws, bugs, vulnerabilities.
Vulnerability	Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by an attacker.
Fault	The adjudged or hypothesized cause of a failure is called a fault. Correct service is delivered when the service implements the system function. A service failure, often abbreviated to failure, is an event that occurs when the delivered service deviates from correct service. A service fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function.
Software Fault	An identified – adjudged or hypothesized – cause of a failure of the service performed by a piece of software under investigation (a discernible white-box code weakness), often related to cybersecurity failures
Root cause	A root cause is an initiating cause of either a condition or a causal chain that leads to an outcome or effect of interest. The term denotes the earliest, most basic, 'deepest', cause for a given behavior (usually of a failure). It is customary to refer to the root cause in singular form, but one or several factors may in fact constitute the root cause(s) of the problem under study. A factor is considered the root cause of a problem if removing it prevents the problem from recurring. A causal factor, conversely, is one that affects an

	event's outcome, but is not the root cause. Although removing a causal factor can benefit an outcome, it does not prevent its recurrence with certainty. Effective problem statements and event descriptions (as failures, for example) are helpful and usually required to ensure the execution of appropriate root cause analyses.
Impact	The magnitude of harm that can be expected to result from the consequences of successful attack resulting in unauthorized disclosure of information, unauthorized modification of information, unauthorized destruction of information, or loss of information or information system availability. Impact can be further categorized as harm to operations, harm to assets, harm to individuals, harm to other organizations, and harm to the nation.
SFP Catalog	The goal of the SFP program is to establish the SFP Catalog. SFP Catalog is a collection of formal machine-consumable content related to software weaknesses. SFPM (the SFP metamodel) is the specification of the content in the SFP Catalog. In addition to the content, the SFP Catalog involves custodians, technical support, business support, and technical infrastructure to access and search the catalog. This specification defines the SFP metamodel and the SFPM XML/XMI format.
Program point	Reference to a specific place in control- and data-flow of the computation. Program point corresponds to a certain activity. Activities are semantic micro operations that can be performed by a computation. In SFP activities are micro-KDM operations.

5 Symbols

List of symbols/abbreviations:

SFP	Software Fault Pattern
SFPM	Software Fault Pattern Metamodel
CWE	Common Weakness Enumeration
KDM	Knowledge Discovery Metamodel
MOF	Meta-Object Facility
XMI	XML Metadata Interchange
SBVR	Semantics of Business Vocabularies and Rules

6 Additional Information

6.1 How to Read this Specification

SFPM XMI is a common normalized format for representing machine-consumable content related to software faults, their formal semantics and their mappings to the elements of the Common Weakness Enumeration (CWE) catalog. SFPM XMI is the canonical representation of the SFP content as defined by the MOF specification and MOF to XMI mapping. This document describes the SFP Metamodel and provides illustrations of SFPM XMI content. In addition, this specification defines and informative "readable SFP language" that provides a very concise representation of the SFP content, suitable for reviews by humans. The SFP content is also illustrated in the "readable SFP language". The specification of the readable SFP language is provided in Appendix A.

This specification has the following structure.

Section 7.1 "SFP Exchange Format" summarizes the key design objectives for the SFP Metamodel and the SFPM XMI format as the canonical representation of the SFP content.

Section 8 "Software Fault Pattern Metamodel" describes the classes of the SFPM and provides examples of the SFPM XMI as well as examples of SFP content in the readable SFP language.

Section 8.1. describes the core concepts of the SFP Catalog.
Section 8.2 describes the sections of the SFP Catalog as the main structuring mechanism for managing content in the catalog.
Section 8.3 describes the framework for the formal definitions of the faulty computations captured by the core elements of the SFP Catalog. These elements specify invariants of data flows as logical propositions for sink, source, the data element of the data flow.
Section 8.4 describes the formalization apparatus developed to provide formal definitions to the elements of data flows. This apparatus is aligned with existing ISO and OMG standards.
Section 8.5 describes the representation of the referenced vocabularies of the SFP Catalog. The formalization apparatus of the SFP Catalog does not define the *meaning* of constructs involved in the definitions of the data flows and their invariants. Instead, this apparatus defines the *structure* of the meaning. The elements of meaning, identified as "atomic formulations" in section 8.4, are supplied by one or mode referenced vocabularies. The SFP Catalog assumes the use of the ISO/OMG Knowledge Discovery Metamodel (KDM) vocabulary as the foundation for the formalizations, and some generic parts of the vocabulary described in the Semantics of Business Vocabularies and Rules (SBVR) specification.

Appendix A provides the specification of the "Readable SFP language" as a context-free grammar. The mapping of the constructs of this language to the elements of the SFPM and thus to SFPM XMI is straightforward. This appendix is informative.

6.2 Acknowledgements

The following companies submitted this specification:

- KDM Analytics
- Lockheed Martin
- MITRE Corporation
- 88solutions
- NoMagic

7 SFP Exchange Format

7.1 Objectives

- Define a common normalized format for representing reusable machine-consumable content related to software faults, their formal semantics and their relationships
- Define a common normalized format for structuring knowledge of software faults
- Define a common format for representing mappings to the formally defined and structured units of software faults to the items in the Common Weakness Enumeration (CWE) catalog
- Contribute to the evolution of the CWE catalog by defining formal compliance points to CWEs
- Define the infrastructure to identify ambiguities, inconsistencies and gaps in the CWE catalog based on the formal descriptions of software faults and the mapping apparatus to the CWE catalog, and the means for sharing these findings throughout the community.
- Align with the standard Knowledge Discovery Metamodel (KDM) for describing basic facts about the software system under assessment
- Align formal definitions of software faults with their impact and define a common format for enumerating impacts of software faults and their variants
- Align with the risk analysis interchange protocol and the TOIF protocol as well as other protocols of the OMG System Assurance Ecosystem to link findings as evidence to risks
- Define a common format for enumerating root causes of software faults
- Align with the OMG TOIF protocol by defining a consistent enumeration of software faults.
- Establish a uniform, vendor-neutral, normalized environment for analyzing knowledge related to software faults
- Define the foundation for the SFP Catalog that will accumulate structured, machine-consumable content related to software faults
- Establish an ecosystem for development of new capabilities that will consume the SFP content and use this content for various purposes including (but not limited to) analytics related to software faults, collecting evidence for digital certification of systems, synthesizing test cases for code analysis tools.

8 Software Fault Pattern Metamodel

This section describes the MOF model for SFPM using UML class diagrams. The SFPM model is the normative part of the SFPM specification. This model determines the SFPM XMI schema by applying the standard MOF to XMI mapping to the SFPM MOF model. The canonical interchange format defined by SFPM is called the SFPM XMI schema. As the means of illustrating the SFPM, examples of the SFP content are provided as fragments of XML/XMI documents compliant to the SFPM XMI schema, as well as in "readable SFP language". This readable SFP language is described in Appendix A to this specification. This language constitutes an informative part of the specification.
The SFPM MOF model consists of a single UML package and includes 16 class diagrams to represent the following:

- Core elements of the SFP Catalog
- Sections of the SFP Catalog
- SFP Defined Elements
- Semantic Formalization Apparatus
- Referenced Vocabularies

The rest of this section has the following organization. Section 8.1 presents UML class diagrams that describe the Core elements of the SFP Catalog. Section 8.2 presents UML class diagrams that describe the structuring mechanism of the SFP Catalog, called "section" and the corresponding classes. Section 8.3 presents UML class diagrams that describe the SFP Defined elements. These elements specify invariants of data flows as logical propositions for sink, source, and the data element of the data flow. Section 8.4 presents UML class diagrams for the SFP's apparatus to define the formal semantics of the SFP elements. Section 8.5 concludes the definition of the SFPM by describing the UML diagrams for the referenced vocabularies.

8.1 Core Elements of the SFP Catalog

This section describes several UML class diagrams that represent the core elements of the SFP catalog: SFP Catalog, SFP and SFP Cluster. Several other classes are also considered as part of the "core": these are the elements representing the parameters and variation of SFPs, elements capturing the common root causes and injuries of software faults, as well as the elements involved in representing mappings of SFP variants to the elements of the Common Weakness Enumeration (CWE) catalog.

8.1.1 SFP Catalog Diagram

This section provides an overview of the core elements of the Software Fault Patterns Catalog. The SFP Catalog class diagram defines the root element – class SFPCatalog – with owned elements Cluster and SFP. The diagram also shows the related CWE elements, organized into one or more CWESection containers. A "section" is a general structuring mechanism of the SFP Catalog. Sections are described in more detail in section 8.2.

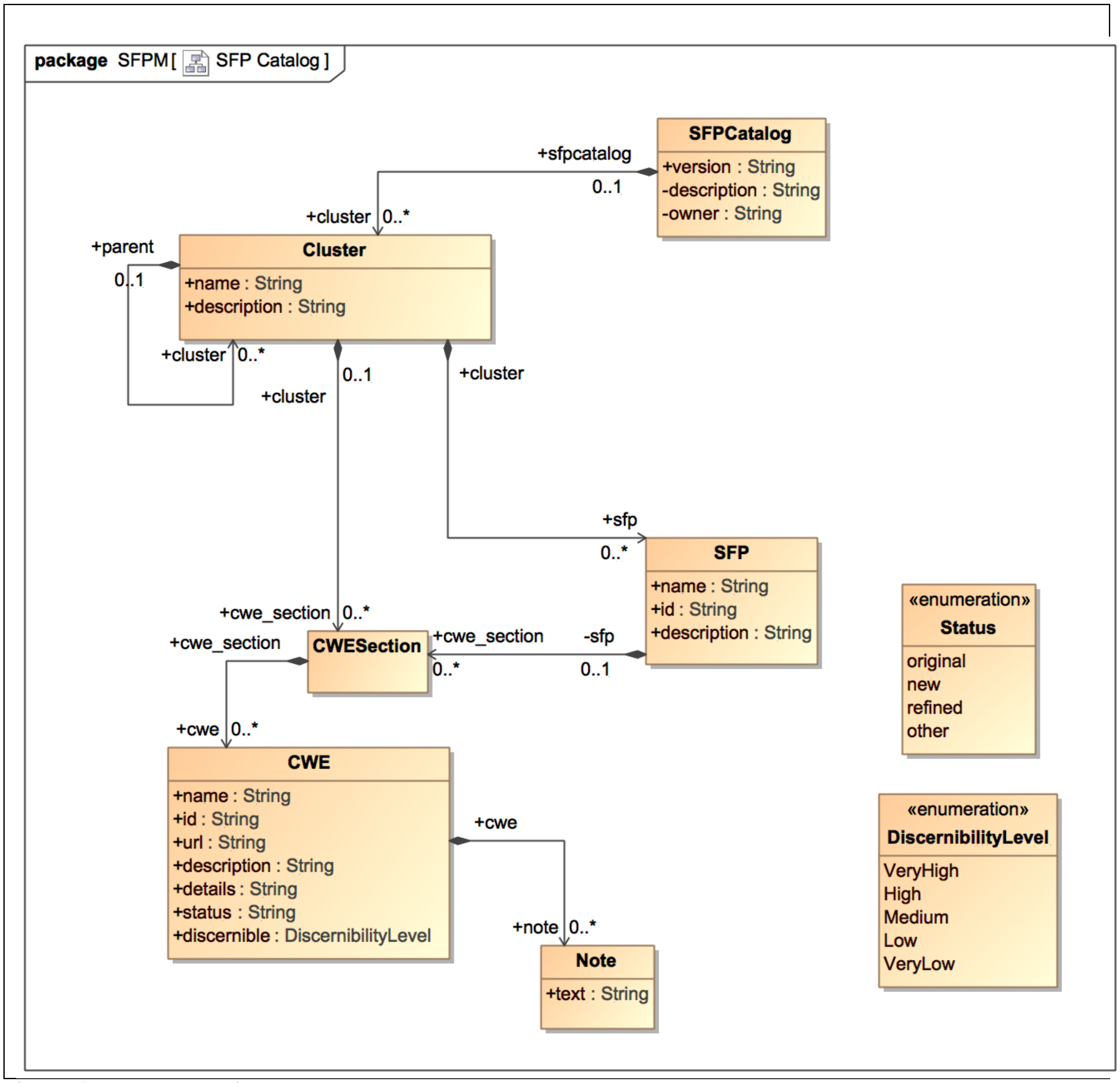

Figure 2. UML class diagram SFP Catalog

8.1.1.1 SFPCatalog Class

The SFPCatalog class is the root class of SFPM. This class represents an instance of an SFP Catalog. One of the objectives of the SFPM is to support the SFP Catalog as the reference collection of the formal machine-consumable content related to software faults. At the same time, multiple SFP Catalog instances can be established. SFPCatalog is simply a container for some SFP content created under some authority. SFPM does not impose any claims regarding completeness or usefulness of the content of any SFPCatalog instance. For example, an instance of SFPCatalog can be used to pack the content related to a single SFP and deliver it to the SFP Catalog custodians to be validated and added to the SFP Catalog. The benefits of the SFP approach come from the content that is shared among multiple SFPs using the mechanism of common sections (CWEsection is an example of a section that can be linked to a single SFP, other sections can be linked to a cluster, or to the entire catalog). Some instances of SFPCatalog may be focused at delivering such common content.

Superclass

Attributes

version:String[1]	Owned attribute that specifies the version of the SFP catalog. The version of the SFP Metamodel is given in the namespace in the XMI
description:String[1]	Informal description of the purpose and content delivered as the owned elements of this element
owner:String[1]	Organization that is the owner of the catalog

Associations

cluster:Cluster[0..*]	Owned collection of Cluster elements

Example 1. SFPM XMI

```
<?xml version="1.0" encoding="UTF-8"?>
<sfpm:SFPCatalog xmlns:xmi="http://www.omg.org/spec/XMI/20131001"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:sfpm="https://www.omg.org/spec/SFPM/20200202"
version="03-08-2015_fp" owner=”sample organization”>
  <cluster name="Memory Access">
    <cluster name="Faulty Pointer Use">
      <sfp name="Faulty Pointer Use" id="7">
        <parameter_section name=""> <!–- body omitted --> </parameter_section>
        <variation_section name=""> <!–- body omitted --> </variation_section>
        <cwe_section name=""> <!–- body omitted --> </cwe_section>
        <element_section name=""> <!–- body omitted --> </element_section>
        <characteristic_section name="">
                  <!–- body omitted --></characteristic_section>
        <canonical_section name=""> <!–- body omitted --> </canonical_section>
        <cwe_mapping_section > <!–- body omitted --> </cwe_mapping_section >
      </sfp>
    </cluster>
  </cluster>
  <context_section name=""> <!–- body omitted --> </context_section>
  <vocabulary_section name=""> <!–- body omitted --> </vocabulary_section>
  <property_section name=""> <!–- body omitted --> </property_section>
  <indicator_section name=""> <!–- body omitted --> </indicator_section>
  <rootcause_section name=""> <!–- body omitted --> </rootcause_section>
  <injury_section name=""> <!–- body omitted --> </injury_section>

  <vocabulary_section name="referenced">
    <vocabulary name="KDM"> <!–- body omitted --> </vocabulary>
    <vocabulary name="Hooks"> <!–- body omitted --> </vocabulary>
    <vocabulary name="Analysis API"> <!–- body omitted --> </vocabulary>
    <vocabulary name="Strings"> <!–- body omitted --> </vocabulary>
    <vocabulary name="SBVR"> <!–- body omitted --> </vocabulary>
    <vocabulary name="Platform Meta"> <!-- body omitted --> </vocabulary>
    <vocabulary name="Platform APIs"> <!-- body omitted --> </vocabulary>
  </vocabulary_section >
</sfpm:SFPCatalog>
```

Example 2 Readable SFP language

```
#####################################
##########  SFP 7   #################
#####################################

Catalog 03-08-2015_fp

Cluster Memory Access
Secondary Faulty Pointer Use
   SFP 7 Faulty Pointer Use

   Parameters
   End Parameters

   Variations
   End Variations

   CWEs
   End CWEs

##############################################################
############### SFP Elements ##################################
##############################################################

   Elements
   End Elements

##############################################

   Characteristics
   End Characteristics

  Canonicals
  End Canonicals

End SFP
End Secondary
End Cluster

#####################################################
##############  Context Elements ####################
#####################################################

SharedContextElements
End SharedContextElements

#####################################################
###########         definitions              ######
#####################################################

Vocabularies
```

```
Definitions KDM Patterns

 End Definitions
End Vocabularies

#####################################
########### Properties ##############
#####################################
Properties
End Properties

###################################################
###############   Indicators   ####################
###################################################

Indicators
End Indicators

End Catalog
```

8.1.1.2 Cluster Class

The Cluster class represents a logically coherent collection of SFP items. The SFP catalog supports at least two levels of clusters: primary clusters and secondary clusters. Primary clusters are represented by instances of the Cluster class owned directly by the SFPCatalog. Secondary clusters are represented by the instances of Cluster class owned the primary clusters.

A primary SFP cluster is a collection of one or more secondary SFP clusters. A primary SFP cluster shall not directly own SFP elements. A secondary SFP cluster is a collection of one or more SFP elements.

A Cluster may have one or more CWE sections which are references to the related elements of the Common Weakness Enumeration (CWE) catalog.

Superclass

Attributes

name: String[1]	Name of the cluster
description:String[1]	Description of the cluster

Associations

cluster:Cluster[0..*]	Owned collection of (secondary) clusters.
sfp:SFP[0..*]	Owned collection of SFP elements
cwe_section:CWESection[0..*]	Owned collection of CWE sections.

Constraints

1. Each Cluster instance of the SFPCatalog shall have a unique name in the scope of the catalog.

2. Each Cluster instance that owns another Cluster shall not own SFP instances

3. Each Cluster instance that owns SFP shall not own another Cluster instances

Example 1. SFPM XMI

```
<cluster name="Memory Access">
    <cluster name="Faulty Pointer Use">
      <sfp name="Faulty Pointer Use" id="7"> <!–- body omitted --> </sfp>
    </cluster>
</cluster>
```

Example 2. Readable SFP language

```
Cluster Memory Access
 Secondary Faulty Pointer Use

   SFP 7 Faulty Pointer Use

   End SFP
 End Secondary
End Cluster
```

8.1.1.3 SFP Class

The SFP class represents a single Software Fault Pattern – a core item of the SFP Catalog. This specification will refer to an instance of SFP class as “SFP” or an “SFP item”, and to the semantically significant parts of its definition as “SFP elements”.

Superclass

Attributes

name:String[1]	Name of the SFP item
id: String[1]	Unique identifier of the SFP item
description:String[1]	Description of the SFP item

Associations

cwe_section:CWESection[0..*]	Owned collection of CWE sections
rootcause:RootCause[0..*]	References to the related Root Cause elements (see SFP Causal Context diagram)
injury:Injury[0..*]	References to the related Injury elements (see SFP Causal Context diagram)

Example 1. SFPM XMI

```
<sfp name="Faulty Pointer Use" id="7">
   <parameter_section name=""> <!—- body omitted --> </parameter_section>
   <variation_section name=""> <!—- body omitted --> </variation_section>
   <cwe_section name=""> <!—- body omitted --> </cwe_section>
   <element_section name=""> <!—- body omitted --> </element_section>
   <characteristic_section name="">
            <!—- body omitted --> </characteristic_section>
   <canonical_section name=""> <!—- body omitted --> </canonical_section>
   <cwe_mapping_section > <!—- body omitted --> </cwe_mapping_section >
   <injury_mapping_section > <!—- body omitted --> </injury_mapping_section >
</sfp>
```

Example 2. Readable SFP language

```
SFP 7 Faulty Pointer Use

   Parameters
   End Parameters

   Variations
   End Variations

   CWEs
   End CWEs

##############################################################
############### SFP Elements #################################
##############################################################

   Elements
   End Elements

##############################################

   Characteristics
   End Characteristics

  Canonicals
  End Canonicals

End SFP
```

8.1.1.4 CWE Class

The CWE class represents an element of the Common Weakness Enumeration (CWE) catalog. CWE catalog has been selected as the reference body of knowledge of software weaknesses. The objective of SFP is to provide structured "viewpoint" on the content that is already in CWE, to provide a set of formal compliance points for the software weaknesses, as well as to resolve the inconsistencies and ambiguities in existing CWE content and fill any gaps in CWE. From the versioning perspective, versions of the SFP catalog are aligned with the versions of CWE catalog, such that a CWE element is identical in all implementations that are based on the same CWE version. However, SFP catalog may suggest new elements to the CWE catalog that resolve inconsistencies or address some gaps. Such new elements are represented by instances of CWE class with "derived" names which includes the name of some existing CWE element and a suffix. The intention of the CWE sections is to facilitate the knowledge transfer to the CWE community.

Superclass

Attributes

name:String[1]	Full name of the element as it appears in the Common Weakness Enumeration (CWE) catalog (or derived from such name by ways of a suffix to indicate refined elements).
id: String[1]	Unique identifier of the element in the CWE catalog (or derived from such identifier by ways of a suffix to indicate refined elements)
url:String[0..1]	Unique URL of the element
description:String[0..1]	Description of the element
details:String[0..1]	Detailed description of the element
status:Status[1]	Status of the element to indicate if this is an original element, or a new element that fills a gap, or a refinement of another element
discernible:DiscernibilityLevel[1]	Level of discernibility of the content available for this CWE in the CWE catalog (established in the course of SFP formalization of CWE, not part of the CWE catalog)

Associations

note:Note[0..*]	Owned collection of informal notes for this element. SFP often includes notes related to the applicable languages, even if this is completely redundant given that the SFP content is formalized using language-neutral KDM representation. Informal notes are often useful to

explain the relationship between CWE and SFP variants

Example 1. SFPM XMI

```
<cwe_section name="">
   <cwe xmi:id="cwe416"
name="Use After Free"
id="416"
description=""
details=""
status="original"
discernible=”Very High”
url="http://cwe.mitre.org/data/definitions/416.html" >
            <note text="Rename to Use After Release" />
            <note text="this pattern involves an explicit release" />
            <note text="the kind of entity must be releasable. This involves read
             or write access via pointer that still exists while the target
            entity was released" />
            <note text="not applicable to java, since there is no explicit
             delete" />
            <note text="c,c++" />
   </cwe>
   <cwe xmi:id="cwe416a" name="Use After Expiration" id="416a"
status="refinement"
discernible=”Very High” >
            <note text="This pattern involves use of an entity that ceased to
            exist for reasons other than an explicit release. The use is via a
            pointer. This involves non-releasable named entities which cease to
            exist while the pointer still exists. This pattern involves read or
            write access." />
            <note text="uses involve passing to known api" />
            <note text="this is not applicable to java, as objects are garbage-
            collected" />
            <note text="c,c++" />
    </cwe>
</cwe_section>
```

Example 2. Readable SFP language

```
#####################################
###########  CWE ####################
#####################################

CWEs

      CWE 416 Use After Free
         description=
         details=
         status=”other”
         discernible=Very High
         url="http://cwe.mitre.org/data/definitions/416.html"
            Mapping: 1.10 2.5 3.3
            Note: Rename to Use After Release
            Note: this pattern involves an explicit release
            Note: the kind of entity must be releasable. This involves read or
```

```
             write access via pointer that still exists while the target entity
             was released
            Note: not applicable to java, since there is no explicit delete
            Note: c,c++
      End CWE

      CWE 416a Use After Expiration
            Mapping: 1.10 2.6 3.3
            Discernible=Very High
            Note: This pattern involves use of an entity that ceased to exist for
            reasons other than an explicit release. The use is via a pointer.
            This involves non-releasable named entities which cease to exist
            while the pointer still exists. This pattern involves read or write
            access.
            Note: uses involve passing to known api
            Note: this is not applicable to java, as objects are
                  garbage-collected
            Note: c,c++
 End CWE
End CWEs
```

8.1.1.5 Note Class

The Note class represents a text note for the CWE element.

Superclass

Attributes

text:String[1]	The body of the note

Example

See 8.1.1.4

8.1.1.6 CWESection Class

The CWESection class represents a container for one or more CWE elements. CWESection is part of the structuring mechanism of the SFP catalog called “sections” that are described in full detail in section 8.2

Superclass

ClusterSection

Associations

cwe:CWE[0..*]	Owned collection of the CWE elements

Example

See 8.1.1.4

8.1.1.7 DiscernibilityLevel Enumeration

The DiscernibilityLevel class introduces levels of discernibility of content available for a CWE element in the Common Weakness Enumeration (CWE) Catalog. CWE catalog introduces signifiers of software weaknesses. Each signifier in CWE is linked to an informal description, and to one or more sections with code samples and cross-references to other content. Discernibility level is an informal measure of how easy it is to recognize the underlying situation (described by CWE signifier) in the code artifacts. A more discernible description can be formalized. A discernible characteristic is a property (used in a semantic definition representing some computation) that can be expressed as a formal statement in the vocabulary of the system's artifacts. The foundation for such vocabulary is KDM, however the definition does not preclude certain extensions. A common way of referring to situations that can be recognized in code is "white-box property" (as opposed for example to a "black box property" that is described purely as a function of the values of inputs and outputs). Thus, a discernible description of a computation is a logical statement that is based entirely on discernible characteristics. A non-discernible description is either ambiguous (the meaning is ill-defined, the description is not a logical statement), uses ill-defined characteristics, uses one or more non-discernible characteristics or is not "white-box". A discernible characteristic emphasizes the artifacts rather than values or state – consistent with the SFP approach.

A non-discernible description can be turned into a discernible one by:

- Providing more clarity and precision
- Using structured language based on controlled vocabulary of well-defined meanings
- Performing additional research to better define the corresponding family of computations, and better defining the characteristics involved in the definition
- Defining additional facts and extending the currently available vocabulary of facts related to the system's artifacts.

Literals

Very High	The content of this CWE weakness description is based directly on the well-understood discernible white-box properties
High	The content of this CWE weakness description is based on discernible white-box properties
Medium	The content of this CWE weakness description is based on discernible white-box properties or

	properties that are believed to be derivable from them
Low	The content of this CWE weakness description involves properties that are not derivable from discernible white-box properties
Very Low	The content of this CWE description is not discernible

Example

See 8.1.1.4 where SFPM XMI representation is illustrated. The actual values of discernibility levels for CWEs are provided in the SFP catalog.

8.1.1.8 Status Enumeration

The Status class introduces Status of a referenced CWE element to indicate if this is an original element, or a new element that fills a gap, or a refinement of another element

Literals

original	The CWE element represents an existing item from the CWE catalog
new	The CWE element represents a new item, not present in the CWE catalog
refinement	The CWE element represents a modification of an existing item in the CWE catalog
other	The CWE element represents a situation not covered by other literals

Example

See 8.1.1.4 where SFPM XMI representation is illustrated.

8.1.2 SFP Variations Class Diagram

This section describes the analytical mechanism of the SFP Catalog that allows managing the content and establishing new properties of the software faults. The elements of this mechanism are SFP Parameters, Variations and Variants. A Software Fault Pattern (SFP) – an SFP item - represents a family of similar faulty computations by identifying a common indicator, common data flow elements and possibly some associated conditions. When generalized, an SFP definition refers to the entire secondary cluster and is arranged into an invariant core and variation points. By focusing at the dataflow elements of faulty computations, the SFP approach allows a generalized statement to cover many situations that share an invariant of the data flow- and thus concisely describe the entire family of computations. A generalized statement includes several "variation points" that are disjunctions of more detailed situations. To ensure full coverage, variation points are identified through top-down analysis of entire cluster space. Once all variation points are identified, they are defined as specific "parameters". In other words, variations introduce additional details for the generalized definition, focusing at the named variation points – the parameters. Each SFP Parameter defines a set of distinct situations, referred to as its Variants. SFP also includes a mechanism to achieve "horizontal" consistency between multiple "slices" of the tree of variants.

Parameters and Variants are part of the mechanism for establishing a mapping between SFP and related CWEs which also contributes to the analytical capabilities of SFP. SFP items map to multiple CWEs in such a way that each CWE in the family can be defined as a specialization of an SFP through a specific set of variants for certain parameters – this can be called a "profile" of the CWE. This specialization is formally defined as a unique set of variants of one or more SFP parameters. Based on this mapping, CWEs can serve as a reporting mechanism for SFP.

Identified Software Fault Pattern definitions provide the foundation for developing more accurate testing tools and improving developer education since it is easier to manage the knowledge of fewer SFPs than hundreds of CWEs. They also provide for a more cost-effective formalization.

Each SFP element owns one or more Parameters, Variants and Variations. This ownership is implemented by the structuring mechanism of SFPM called "sections". Section as fully explained in section 8.2. Parameters, and Variations are owned by separate sections. Variants are owned by each Parameter as illustrated below. Properties are owned by yet another section owned by the entire catalog.

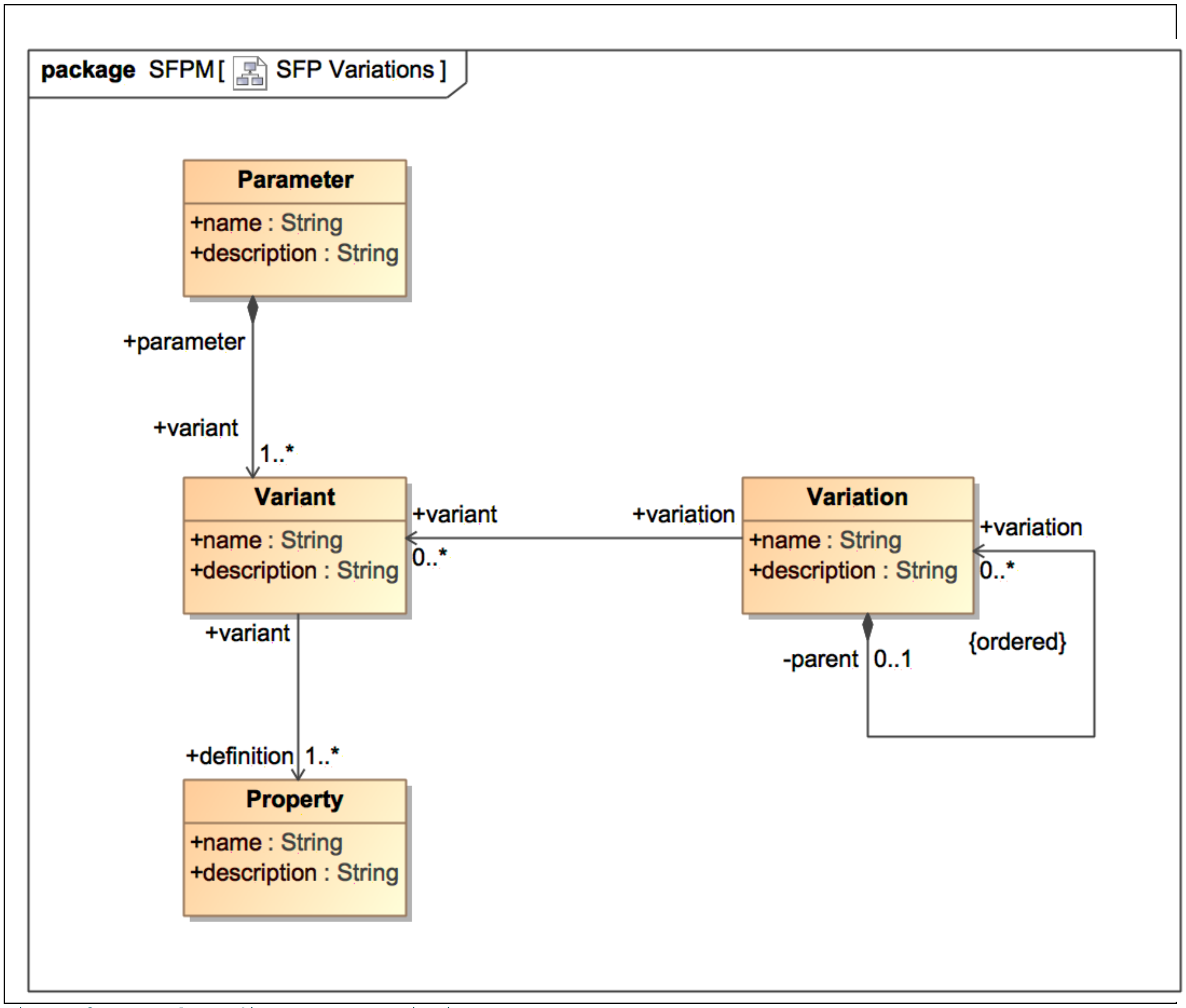

Figure 3. UML class diagram SFP Variations

8.1.2.1 Parameter

Parameter is one of the key concepts of the structured approach to formally defining software faults. According to this approach, an SFP defines a family of computations that exhibits a certain fault. First these computations are defined by their characteristic Sink, Source and Data (referred to as the SFP Dataflow Elements). Then a set of Parameters is identified and enumerated where a Parameter is one of the "concepts" involved in the definition of the faulty computation (part of the Sink, Source or Data). Each Parameter defines a set of distinct situations, referred to as its Variants. Parameters are owned by an SFP element through a ParameterSection (as described in section 8.2 in more detail).

Superclass

Attributes

name:Name[1]	Name of the parameter

Associations

variant:Variant[1..*]	Owned set of Variants for the Parameter

Example 1. SFPM XMI

```
<parameter_section name="">
  <parameter name="Pointer Use Kind">
    <variant xmi:id="variant1" name="1.1 Dereference" definition="prop1" />
    <variant xmi:id="variant2" name="1.2 Call via pointer" definition="prop1" />
    <variant xmi:id="variant3" name="1.3 Access to Member via pointer"
            definition="prop2" />
    <variant xmi:id="variant4" name="1.4 Method call via pointer"
            definition="prop3" />
    <variant xmi:id="variant5" name="1.5 Access with index" definition="prop4" />
    <variant xmi:id="variant6" name="1.6 Cast" definition="prop5" />
    <variant xmi:id="variant7" name="1.7 Hidden access via api"
            definition="prop6" />
    <variant xmi:id="variant8" name="1.10 Any use" definition="prop7" />
    <variant xmi:id="variant9" name="1.11 Access to Member via overlay struct"
            definition="prop8" />
    <variant xmi:id="variant10" name="1.12 Access to Method via overlay class"
            definition="prop9" />
  </parameter>
  <parameter name="Incorrect Value Kind">
    <variant xmi:id="variant11" name="2.1 Pointer is NULL" definition="prop10" />
    <variant xmi:id="variant12" name="2.2 Pointer is invalid" definition="prop11"
            />
    <variant xmi:id="variant13" name="2.4 Faulty Type" definition="prop12" />
    <variant xmi:id="variant14" name="2.5 Entity is released" definition="prop13"
            />
    <variant xmi:id="variant15" name="2.6 Entity ceased to exist"
definition="prop14" />
    <variant xmi:id="variant16" name="2.7 Any value" definition="prop15" />
    <variant xmi:id="variant17" name="2.8 Not valid for call" definition="prop16"
            />
  </parameter>
  <parameter name="Access Kind">
    <variant xmi:id="variant18" name="3.1 Read access" definition="prop17" />
    <variant xmi:id="variant19" name="3.2 Write access" definition="prop18" />
    <variant xmi:id="variant20" name="3.3 Read or Write" definition="prop19" />
    <variant xmi:id="variant21" name="3.4 Call" definition="prop20" />
    <variant xmi:id="variant22" name="3.5 Not applicable" definition="prop21" />
    <variant xmi:id="variant23" name="3.7 Object oriented access"
            definition="prop22" />
  </parameter>
</parameter_section>
```

Example 2. Readable SFP language

```
Parameters

  Parameter Pointer Use Kind
    Variant 1.1 Dereference -> Property "access mechanism pointer"
    Variant 1.2 Call via pointer -> Property "access mechanism pointer"
    Variant 1.3 Access to Member via pointer -> Property "access mechanism
            member"
    Variant 1.4 Method call via pointer -> Property "access mechanism method"
    Variant 1.5 Access with index -> Property "access mechanism index"
    Variant 1.6 Cast -> Property "access mechanism cast"
    Variant 1.7 Hidden access via api -> Property "access mechanism hidden"
    Variant 1.10 Any use -> Property "access mechanism any"
    Variant 1.11 Access to Member via overlay struct -> Property "access
             mechanism overlay"
    Variant 1.12 Access to Method via overlay class -> Property "access
            mechanism overlay call"
  End Parameter

  Parameter Incorrect Value Kind
    Variant 2.1 Pointer is NULL -> Property "value null"
    Variant 2.2 Pointer is invalid  -> Property "value invalid"
    Variant 2.4 Faulty Type -> Property "value faulty type"
    Variant 2.5 Entity is released -> Property "value released"
    Variant 2.6 Entity ceased to exist -> Property "value expired"
    Variant 2.7 Any value -> Property "any value"
    Variant 2.8 Not valid for call -> Property "value not callable"
  End Parameter

  Parameter Access Kind
    Variant 3.1 Read access -> Property "access read"
    Variant 3.2 Write access  -> Property "access write"
    Variant 3.3 Read or Write -> Property "access read or write"
    Variant 3.4 Call -> Property "access call"
    Variant 3.5 Not applicable -> Property "access any"
    Variant 3.7 Object oriented access -> Property "access oo"
  End Parameter

End Parameters
```

8.1.2.2 Variant Class

An SFP Variant is a fundamental concept of the structured approach to formally defining software faults. According to this approach, an SFP defines a family of computations that exhibits some fault. First these computations are defined by describing its characteristic Sink, Source and Data (referred to as the SFP Dataflow Elements). Then a set of Parameters is identified and enumerated where a Parameter is one of the concepts involved in the definition of the faulty computation (part of the Sink, Source or Data). Each Parameter defines a set of distinct situations, referred to as its Variants. The computation is defined by a covering set of cases each uniquely identified by a combination of distinct Variants. Another element called Variation helps manage the permutations of the Variants.

A combination of variants for the SFP's parameters provides a slice of the faulty computation. Each such slice may have own root causes and impacts. SFP Catalog provides a mapping between computation slices and elements of the Common Weakness Enumeration (CWE) catalog. The structured approach of the SFP allow to formally define individual CWEs as SFP slices, defined as a set of variants for SFP's parameters. This approach allows to detect ambiguities, overlaps and gaps in the CWE catalog. These observations are captured as notes in the CWE mappings in the SFP Catalog.

Variants are owned by the corresponding Parameter of the SFP. SFP element owns Parameters through a ParameterSection (as described further in section 8.2 in more detail).

Superclass

Attributes

name:String[1]	The name of the variant
description:String[1..*]	Description of the variant

Associations

definition:Property[1..*]	Definition of the variant in terms of one or more properties

Example 1. SFPM XMI

```
<variant xmi:id="variant2" name="1.2 Call via pointer" definition="prop1" />
```

Example 2. Readable SFP language

```
Variant 1.2 Call via pointer -> Property "access mechanism pointer"
```

See also 8.1.2.1

8.1.2.3 Variation Class

Variation class is involved in constructing "variation trees" – auxiliary structures that help manage variants of an SFP. Variation trees are represented as follows. A Variation element may own several (nested) variation elements. This parent variation usually corresponds to a certain parameter element, although this link is not explicit in SFPM. The leaf variations refer to certain variants. Variation section owns a set of top variations. Nesting of variations imposes dependencies between parameters and their variants. The variation tree restricts acceptable permutations of the variants. Variations in the variation tree are ordered. The ordering of the variants in the tree may be utilized to achieve predictable enumeration of all possible permutations of the variants.

The "variation tree" defines the initial structure of the family of computations identified as an SFP. Further, the SFP elements define the invariant of the data flows involved, by defining the sink (a collection of the Indicators), the primary data element of the data flow, the source and the invariant condition. These elements are defined as a disjunction of "clauses", enumerating various distinct situations involved in the data flow. Consistency of the clauses of the SFP element, as well as their correlation with the “variation tree” is achieved using “properties”. Each property is defined as a set of “tags”. Two clauses are compatible if they include tags with matching values.

When the SFP content is used to synthesize representative samples of “compliant” or “non-compliant” (but similar-looking) computations, the tags guide the selection of the computation slices, and can be used to identify a given computation slice.

Properties are further defined in section 8.3.

Variation tree is closely aligned with the CanonicalForm of the SFP that describes the structure of the multitude of canonical representations of the computations described by the SFP with full context. Canonical Elements are further described in section 8.3.

SFP element owns Variations though a VariationSection (as further described in section 8.2 in more detail).

Superclass

Associations

name:String[1]	Name of the variation
description:String[1]	Description of the variation

Associations

variation:Variation[0..*] {ordered}	Owned (nested) variations (ordered)
variant:Variant[0..*]	Specific variant that defines the variation

Example 1. SFPM XMI

This example illustrates variation tree for SFP-7. Parameters for SFP-7 are illustrated in section 8.1.2.1. The top level of the variation tree has two variations: DataType and Parameter Value Kind. For each variant of the Parameter Value Kind, the tree has all variations of the Parameter Access Kind. Then for each variation of the Access Kind, the tree has appropriate variations of the Parameter Use Kind. DataType is “built-in” Parameter, describing variants of a data type (e.g. character, integer, Boolean, string, pointer, etc.).

```
<variation_section name="">
  <variation name="DataType" />
  <variation name="Parameter Value Kind" >
    <variation name="Pointer is NULL" variant="variant11" >
      <variation name="Parameter Access Kind" />
      <variation name="Read" variant="variant18" >
        <variation name="Ordinary Pointer Dereference" variant="variant1" />
        <variation name="Access with index" variant="variant5" />
        <variation name="Access to member via pointer" variant="variant3" />
                  <variation name="Access to member via overlay struct"
                  variant="variant9" >
          <variation name="Hidden call via API" variant="variant7" />
        </variation>
      </variation>
      <variation name="Write" variant="variant19" >
        <variation name="Ordinary Pointer Dereference" variant="variant1" />
        <variation name="Access with index" variant="variant5" />
        <variation name="Access to member via pointer" variant="variant3" />
        <variation name="Access to member via overlay struct"
             variant="variant9" >
            <variation name="Hidden call via API" variant="variant7" />
        </variation>
      </variation>
      <variation name="Call" variant="variant21" > … </variation>
    <variation name="Pointer is invalid" variant="variant12" > … </variation>
    <variation name="Entity has been released" variant="variant14" > …
            </variation>
    <variation name="Entity ceased to exist" variant="variant15" > … </variation>
    <variation name="Pointer is valid but faulty type" variant="variant13" > …
            </variation>
  </variation>
</variation_section>
```

Example 2. Readable SFP language

```
Variations

  DataType
  Parameter Value Kind
      Pointer is NULL -> 2.1
        Parameter Access Kind
            Read ->  3.1
                  Ordinary Pointer Dereference ->  1.1
                  Access with index -> 1.5
                  Access to member via pointer -> 1.3
                  Access to member via overlay struct-> 1.11
                        Hidden call via API -> 1.7
            Write -> 3.2
                  Ordinary Pointer Dereference ->  1.1
                  Access with index -> 1.5
                  Access to member via pointer -> 1.3
                  Access to member via overlay struct-> 1.11
                     Hidden call via API -> 1.7
            Call -> 3.4
                  …
      Pointer is invalid -> 2.2
            …
      Entity has been released -> 2.5
            …
      Entity ceased to exist -> 2.6
…
      Pointer is valid but faulty type -> 2.4
…
End Variations
```

8.1.2.4 Property Class

Property class is a semantic element that provides definitions for variants in terms of special tags (markers). This class is described in more detail in section 8.3.
The purpose of the tags is to correlate variations with the clauses of the formalized descriptions.

When the SFP content is used to synthesize representative samples of "compliant" or "non-compliant" (but similar-looking) computations, the tags guide the selection of the computation slices, and can be used to identify a given computation slice.

8.1.3 SFP Causal Context Class Diagram

This section describes the UML representation of the elements that capture the cause and effect of a software fault.

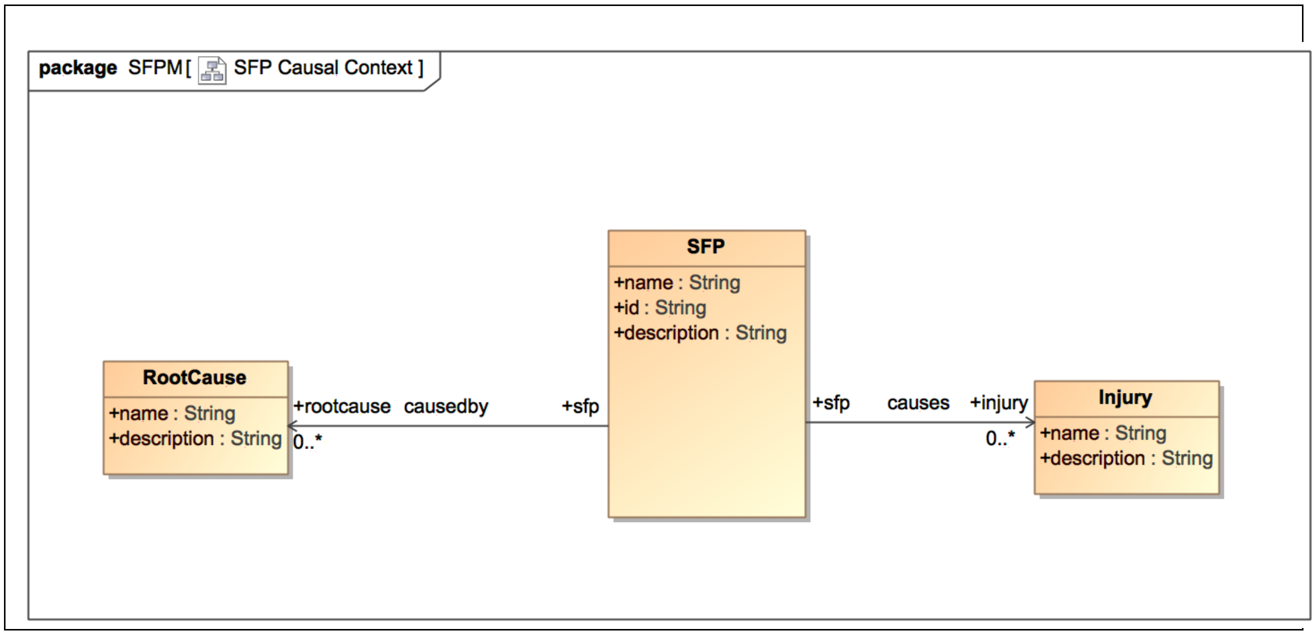

Figure 4. UML class diagram SFP Causal Context

8.1.3.1 RootCause Class

RootCause class defines a typical root cause for an SFP item. Root causes – also known as "vulnerability fundamentals" – are typical factors that may be facilitating vulnerabilities, especially as they are introduced during the design and development of systems. Root causes may be attributed to programming languages, the runtime systems, the hardware or any other parts of the environment. A typical root cause may not be the same as the actual root cause for a bug in a specific system under assessment. A RootCause is a useful abstraction. Enumerating possible root causes for the Software Fault Patterns as part of the SFP Catalog is aimed at steering research into hardening systems. SFPM facilitates analytics that may reveal common root causes.

RootCause elements are owned by the SFPCatalog through one or more RootCauseSection containers. Collectively, RootCause elements define the set of possible root causes of the faults covered by the SFP Catalog.

Superclass

Attributes

name:String[1]	Name of the root cause
description:String[1..*]	Description of the root cause

Example 1. SFPM XMI

```
<rootcause_section name="">

  <rootcause xmi:id="rc1" name="Lack of automatic management of buffers"
            description="language runtime"/>
  <rootcause xmi:id="rc2" name="Failure to provide integrity of internal
             references to memory buffer contents"/>
  <rootcause xmi:id="rc3" name="Disconnect between dumb pointers and resources
       that they represent"/>
  <rootcause xmi:id="rc4" name="Failure to compute size of memory buffer content
             parts"/>
  <rootcause xmi:id="rc5" name="Lack of exception on incorrect pointer use"/>
  <rootcause xmi:id="rc6" name="Failure to process fault state"/>

</rootcause_section>

<sfp name="Faulty Pointer Use" id="7" rootcause="rc1 rc2 rc3 rc4 rc5 rc6">
<sfp name="Faulty Buffer Access" id="8" rootcause="rc1 rc7 rc8 rc9 rc10 rc11">
```

Example 2. Readable SFP language

```
SFP 7 Faulty Pointer Use
      RootCauses
            Lack of automatic management of buffers
            Failure to provide integrity of internal references to memory buffer
            contents
            Disconnect between dumb pointers and resources that they represent
            Failure to compute size of memory buffer content parts
            Lack of exception on incorrect pointer use
            Failure to process fault state
      End RootCauses

End SFP
```

8.1.3.2 Injury Class

Injury class defines a specific impact caused by a vulnerability to the operations of the system. Impact consists of Confidentiality Impact, Integrity Impact and Availability Impact. NIST Common Vulnerability

Scoring System (CVSS) provides measurement schema for impact, and the NIST National Vulnerability Database (NVD) provides measures of impact for known vulnerabilities in open source and commercial software systems.

Confidentiality Impact measures the impact on confidentiality of a successfully exploited vulnerability. Confidentiality refers to limiting information disclosure to only authorized users, as well as preventing access by, or disclosure to, unauthorized users.

Integrity Impact measures the impact on integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and guaranteed veracity of information. Integrity impact involves modification of some system files or information.

Availability Impact measures the impact of availability of a successfully exploited vulnerability. Availability refers to the accessibility of information resources. Attacks that consume bandwidth, processor cycles, or disk space all impact availability of the system.

The Injury element of the SFP Catalog represents an enumeration of the situations with impact. Enumerations of the impact situations and mapping specific variants of Software Fault Patterns to impact aims at establishing a mapping between weakness findings and risks.

SFPM views injuries as a flat enumeration however they can be described hierarchically, with the following 3 tiers. The first tier is the base type of impact: Confidentiality, Integrity and Availability. The second tier considers the object of impact: Data, Service and Resource. The third tier considers several specific situations. Data impacts can involve Data at rest, Data in motion or Data in use. Service impacts involve Disclosure, Distortion, Subversion, Shutdown and Lock. Further Subversion of a service may involve Code at rest or code in motion. Availability of the Data at rest may involve Damage or Lock. Also, a fault may not cause impact directly, but may contribute to other faults.

Some weaknesses may not have an impact from the cybersecurity perspective, but may contribute to other weaknesses. This type of indirect impact is also represented by the Injury elements (illustrated below).

SFP Catalog provides two places where the impact of a fault is described. First the injuries of an entire SFP are enumerated. Second, SFP variants are mapped to specific injuries.

The SFP Catalog owns Injury elements through an InjurySection container. Links between individual Variants of an SFP to Injuries are established through InjuryMapping class (described in a subsequent section). Each SFP owns InjuryMapping through InjuryMappingSection container. Sections of the SFP Catalog are described in more detail in section 8.2

Superclass

Attributes

name:String[1]	Name of the injury
description: String[1]	Description of the injury

Example 1. SFPM XMI

```
<injury_section name="">

  <injury xmi:id="inj1" name="Availability of service"/>
  <injury xmi:id="inj2" name="Contributes to SFP-4"/>
  <injury xmi:id="inj3" name="Subversion of service (especially bulk write
access)"/>
  <injury xmi:id="inj4" name="Distortion of service (write access)"/>
  <injury xmi:id="inj5" name="Confidentiality (read access)"/>

</injury_section>

<sfp name="Faulty Pointer Use" id="7" injury=”inj1 inj2”>
<sfp name="Faulty Buffer Access" id="8" injury=”inj1 inj3 inj4 inj5”>
```

Example 1. Readable SFP language

```
SFP 7 Faulty Pointer Use
      Injuries
            Shutdown of service
            Contributes to SFP11
      End Injuries

End SFP

SFP 8 Faulty Buffer Access
      Injuries
            Shutdown of service
            Subversion of service (especially bulk write access)
            Distortion of service (write access)
            Confidentiality (read access)
      End Injuries

End SFP
```

8.1.4 SFP Variant Mappings Class Diagram

This section describes the UML representation of the variant mappings.

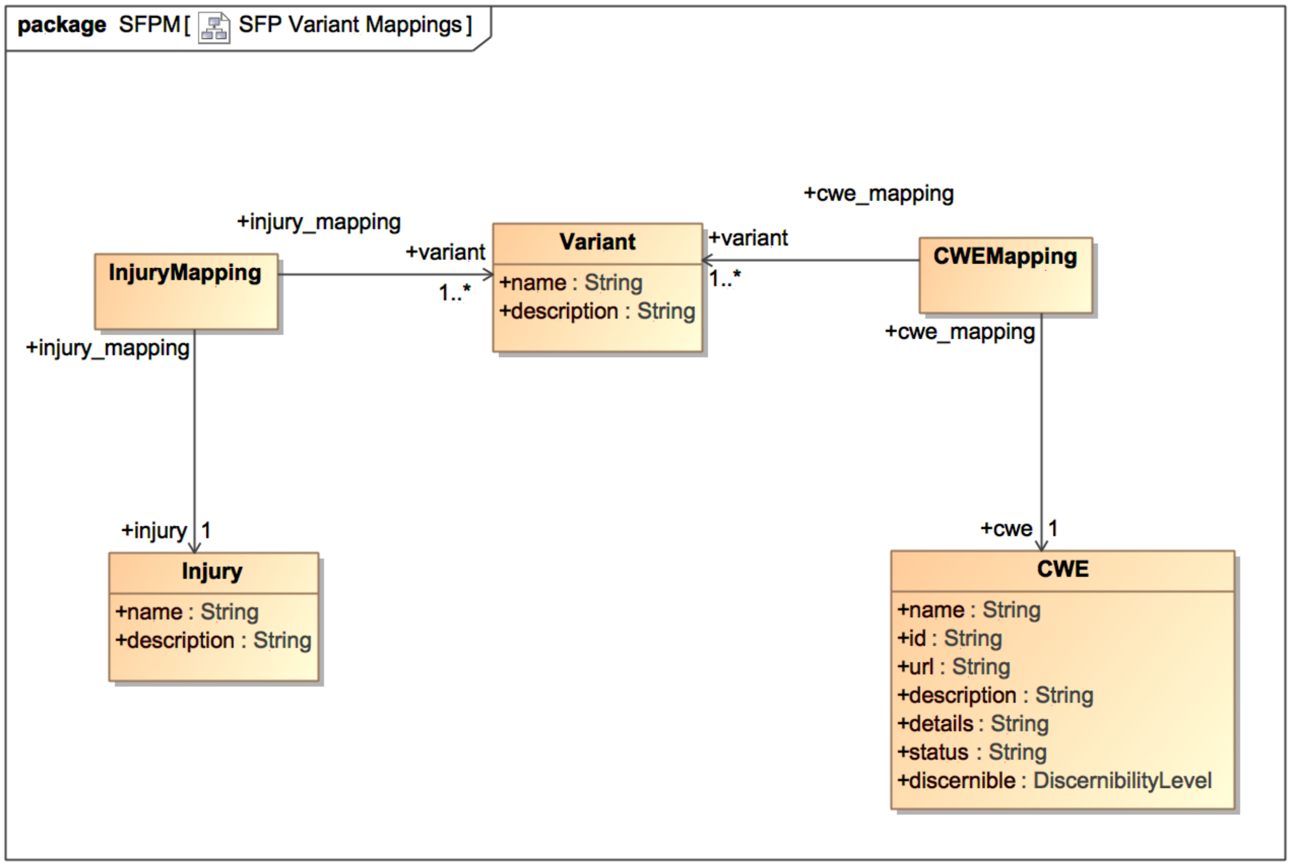

Figure 5. UML class diagram SFP Variant Mappings

8.1.4.1 InjuryMapping Class

InjuryMapping class defines a mapping between Variants of an SFP and Injury elements. An SFP element owns InjuryMappings through InjuryMappingSection container (further described in section 8.2 in more detail).

Superclass

Associations

injury:Injury[1]	Reference to an Injury
variant:Variant[1..*]	Reference to one or more Variant

Constraints

1. Each Injury referenced by the SFP shall be mapped to one or more Variants of the SFP

Example 1. SFPM XMI

```
<injury_mapping_section >
```

```
  <injury_mapping injury="inj1" variant="variant18 variant19 variant20 variant21
variant22 variant23" />
  <injury_mapping injury="inj2" variant="variant18 variant19 variant20 variant21
variant22 variant23" />
</injury_mapping_section >

<injury_section name="">

  <injury xmi:id="inj1" name="Availability of service"/>
  <injury xmi:id="inj2" name="Contributes to SFP-4"/>
  <injury xmi:id="inj3" name="Subversion of service (especially bulk write
access)"/>
  <injury xmi:id="inj4" name="Distortion of service (write access)"/>
  <injury xmi:id="inj5" name="Confidentiality (read access)"/>

</injury_section>

<sfp name="Faulty Pointer Use" id="7" injury=”inj1 inj2”>
```

Example 2. Readable SFP language

```
    Variant 3.1 Read access -> Property "access read"
      Injury: “Availability of service”, “Contributes to SFP-4”
```

8.1.4.2 CWEMapping Class

CWEMapping class defines a mapping between a CWE element and one or more Variant elements. The intent of the CWE mapping is to provide a formal definition of a CWE element as a profile of SFP variants.

An SFP element owns CWEMappings through CWEMappingSection container (further described in section 8.2 in more detail).

Superclass

Associations

cwe:CWE[1]	CWE element being defined in terms of SFP variants
variant: Variant[1..*]	Set of SFP variants that defines a CWE element

Example 1. SFPM XMI

```
<cwe_mapping_section >
  <cwe_mapping cwe="cwe476a" variant="variant12 variant20 variant8" />
  <cwe_mapping cwe="cwe476b" variant="variant11 variant21 variant8" />
  <cwe_mapping cwe="cwe476c" variant="variant17 variant21 variant8" />
</cwe_mapping_section >
```

Example 2. Readable SFP language

```
CWE 476a Invalid Pointer Dereference
      Mapping: 1.10 2.2  3.3
      Note: c,c++
End CWE

CWE 476b NULL Pointer Call
      Mapping: 1.10 2.1 3.4
      Note: c,c++, java
 End CWE

CWE 476c Invalid Pointer Call
      Mapping: 1.10 2.8 3.4
      Note: c,c++
 End CWE
```

8.2 Sections of the SFP Catalog

A section is the structuring mechanism of the SFP catalog. Sections group common content and provide scoping: common sections contain content for the entire SFP catalog that can be shared between all SPF items owned by the SFP Catalog; cluster sections contain content that can be shared by the SFP items within this cluster; SFP sections contain content referenced by a single SFP item. SFP Catalog allows multiple sections of the same type at the same scope. This provides additional grouping capability for the readers, however semantically there is no difference between such sections.

8.2.1 All Sections Class Diagram

This section provides an overview of all sections of the SFP Catalog.

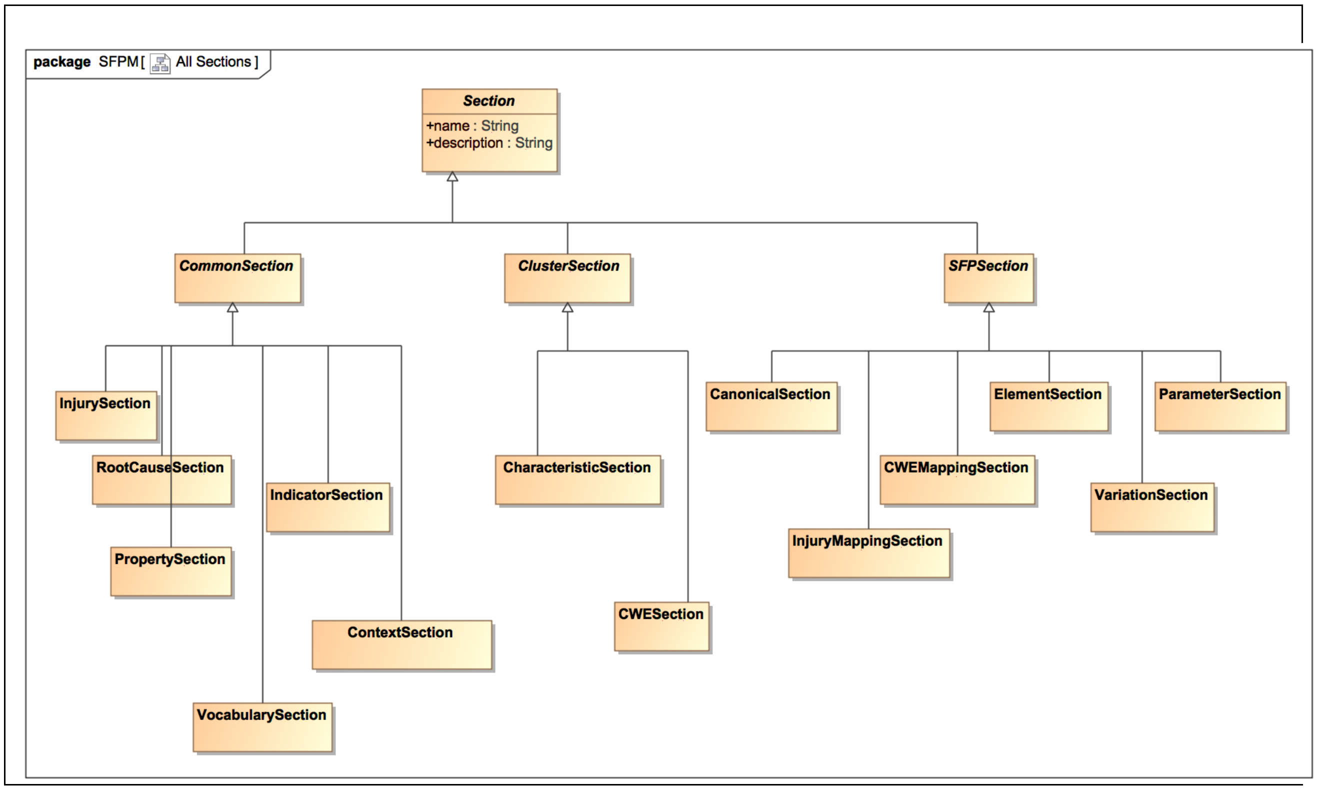

Figure 6. UML class diagram All Sections

8.2.1.1 Section Class (abstract)

The Section class is the common parent of all sections in the SFPM.

Superclass

Attributes

name:String[1]	Name of the section
description:String[1]	Description of the section

8.2.1.2 CommonSection Class (abstract)

CommonSection class is a parent class for all sections that represent the common reusable content of the SFP catalog, i.e. the content that is applicable to the entire collection of SFP items, and is referenced by these items (in the current catalog or in other catalogs) or may be referenced by the items in the future releases of the current catalog. Other kinds of sections represent the content that is specific to either an individual SFP item or specific to a certain cluster of SFP items.

Superclass

Section

Constraints

1. Owned elements of a common section shall not reference elements owned by any cluster section or by any SFP section. An element references another element either directly or indirectly in its owned semantic definition.

8.2.1.3 ClusterSection Class (abstract)

ClusterSection class is a parent class for all sections that represent the content specific to a certain SFP cluster or to an individual SFP item.

Superclass

Section

Constraints

1. Owned elements of a cluster section shall not reference elements owned by any section from a different cluster or owned by any SFP section

8.2.1.4 SFPSection Class (abstract)

SFPSection class is a parent class for all sections that represent the content specific to an individual SFP item.

Superclass

Section

Constraints

1. Owned elements of an SFP section shall not reference elements owned by any section from a different cluster or owned by any SFP section owned by a different SFP

8.2.2 SFP Sections Class Diagram

This section describes the sections of the SFP that are specific to an individual SFP item. Another section called CharacteristicsSection may be owned by SFP as well as by a Cluster, and is described separately in section 8.2.4.

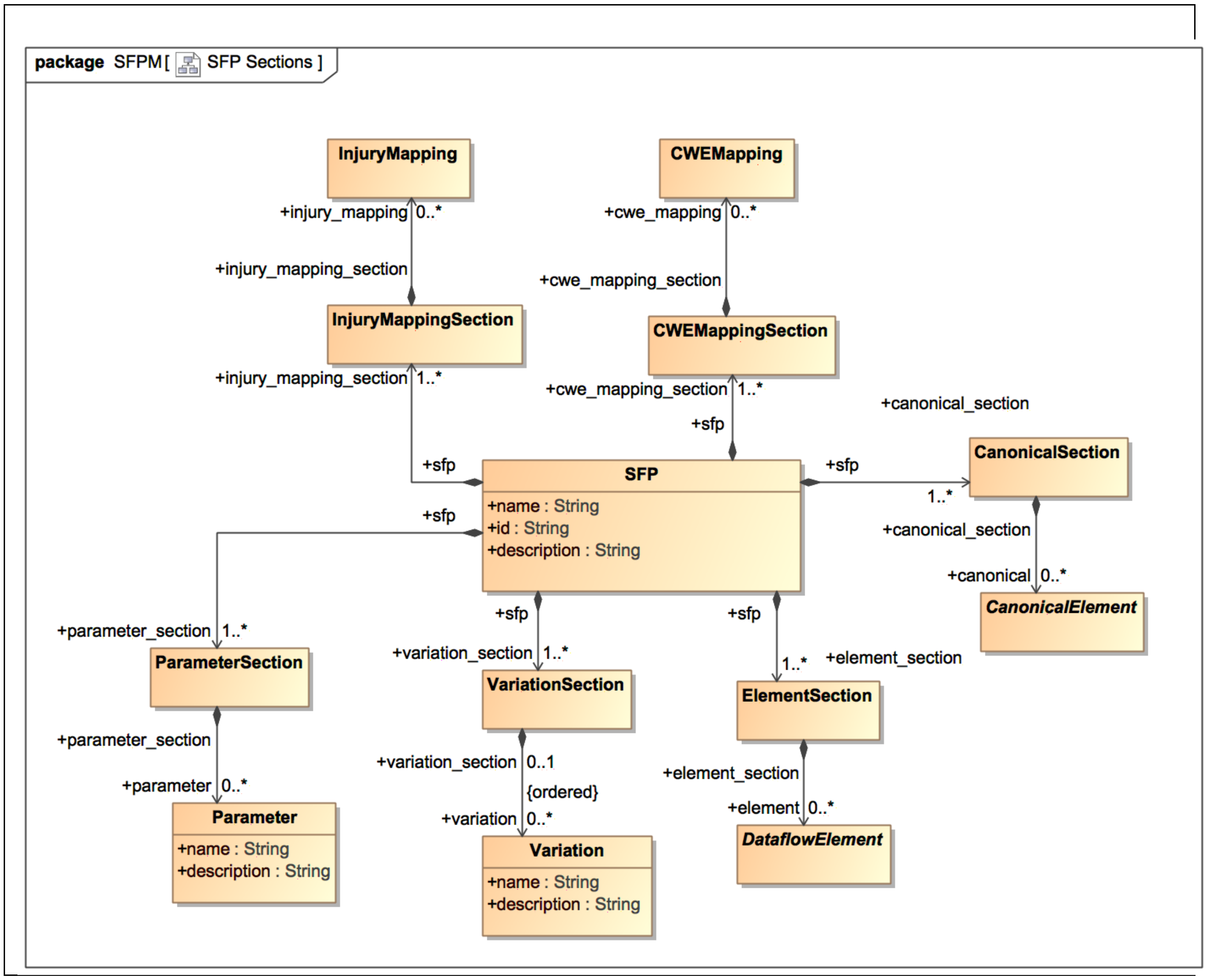

Figure 7. UML class diagram SFP Sections

8.2.2.1 InjuryMappingSection Class

The InjuryMappingSection class is a container for the InjuryMapping elements. Each SFP item owns the InjuryMapping elements for its variants.

Superclass

SFPSection

Associations

injury_mapping:InjuryMapping[0..*]	Owned set of the injury mapping elements for the SFP item

Constraints

1. Each SFP item shall own at least one InjuryMappingSection

Example

See 8.1.4.1

8.2.2.2 CWEMappingSection Class

CWEMappingSection class is a container for the CWEMapping elements. Each SFP item owns the CWEMapping elements for its variants.

Superclass

SFPSection

Associations

cwe_mapping:CWEMapping[0..*]	Owned set of CWEMapping elements for the SFP

Constraints

1. Each SFP shall own at least one CWEMappingSection

Example

See 8.1.4.2

8.2.2.3 ParameterSection Class

ParameterSection class is a container for the Parameter elements.

Superclass

SFPSection

Associations

parameter:Parameter[0..*]	Owned set of Parameter

Example

See 8.1.2.1

8.2.2.4 VariationSection Class

VariationSection class is a container for the Variation elements.

Superclass

SFPSection

Associations

variation:Variation[0..*] {ordered}   Owned set of Variation (ordered)

Example

See 8.1.2.3

8.2.2.5 ElementSection Class

ElementSection class is a container for the SFP DataflowElement. These elements specify dataflows that constitute the *extension* of the SFP as a concept. "Extension" is the totality of objects to which a concept corresponds. According to the SFP approach, the "objects" of software weaknesses are dataflows implemented in code. SFP items are denotations (semantic definitions) for families of dataflows that correspond to the classes of software weaknesses introduced by the CWE catalog. CWE catalog provides signifiers to the software weaknesses. SFP provides formal semantic definitions to a subset of software weaknesses in CWE catalog, and links these definitions to the corresponding CWE items. SFP DataflowElement is an SFP Defined Element, so semantics of DataflowElements is defined according to the formalization apparatus defined in section 8.4. DataflowElements correspond to the key parts of a dataflow. DataflowElement class and its subclasses are further described in section 8.3.

Superclass

SFPSection

Associations

element:DataflowElement[0..*]   Owned set of Dataflow Element of the SFP

Example 1. SFPM XMI

```
<element_section name="">
    <element xmi:type="sfpm:PrimaryDataStatement" xmi:id="cla1">
            <!—- body omitted --> </element>
    <element xmi:type="sfpm:SourceStatement" xmi:id="cla2">
            <!—- body omitted --> </element>
    <element xmi:type="sfpm:SinkStatement" xmi:id="cla3">
            <!—- body omitted --> </element>
</element_section>
```

Example 2. Readable SFP language

```
Elements
      PrimaryDataStatement … End PrimaryDataStatement
      SourceStatement … End SourceStatement
      SinkStatement … End SinkStatement
End Elements
```

8.2.2.6 CanonicalSection Class

CanonicalSection class is a container for CanonicalElement. CanonicalElement provide canonical definition of the dataflow with full context. CanonicalElement class and its subclasses are further described in section 8.3.

Superclass

SFPSection

Associations

canonical:CanonicalElement[0..*]	Owned set of Canonical Element

Example 1. SFPM XMI

```
<canonical_section name="">
   <canonical xmi:type="sfpm:CanonicalForm" xmi:id="cla40" name="CF1" >
            <!—- body omitted --> </canonical>
    <canonical xmi:type="sfpm:PrimaryDataSegment" xmi:id="cla41" >
            <!—- body omitted --> </canonical>
    <canonical xmi:type="sfpm:SourceSegment" xmi:id="cla42" >
            <!—- body omitted --> </canonical>
    <canonical xmi:type="sfpm:SinkSegment" xmi:id="cla43" >
            <!—- body omitted --> </canonical>
    <canonical xmi:type="sfpm:MitigatedSourceSegment" xmi:id="cla44" >
            <!—- body omitted --> </canonical>
    <canonical xmi:type="sfpm:MitigatedSinkSegment" xmi:id="cla45" >
            <!—- body omitted --> </canonical>
</canonical_section>
```

Example 2. Readable SFP language

```
Canonicals
      Canonical CF1  … End Canonical
      Segment PrimaryDataSegment … End Segment
      Segment SourceSegment … End Segment
      Segment SinkSegment … End Segment
      Segment MitigatedSourceSegment … End Segment
      Segment MitigatedSinkSegment … End Segment
End Canonicals
```

8.2.2.7 SFP Class (additional properties)

Class diagram SFP Sections introduces several additional properties to the SFP class.

Superclass

Associations

injury_mapping_section:InjuryMappingSection[1..*]	Injury mapping section of the SFP

cwe_mapping_section:CWEMappingSection[1..*]	CWE mapping section of the SFP
parameter_section:ParameterSection[1..*]	Parameters and variants of the SFP
variation_section:VariationSection[1..*]	Variations of the SFP
element_section:ElementSection[1..*]	Elements of the SFP
canonical_section:CanonicalSection[1..*]	Canonical elements of the SFP

Example

See 8.1.1.3, 8.1.2.1, 8.1.2.3, 8.1.4.1, 8.1.4.2 and also 8.3.2 and 8.3.4

8.2.3 Common Sections Class Diagram

This section describes the sections of the SFP Catalog. These sections are containers for the formal content that is common across multiple SFPs. Accumulation of the common content for multiple software faults is one of the objectives of the SFP approach. The SFPM is structured to enable analytics related to the software faults. The common content includes Indicators, shared characteristics, common referenced vocabularies, enumeration of the root causes and injuries, as well as the enumeration of the common properties.

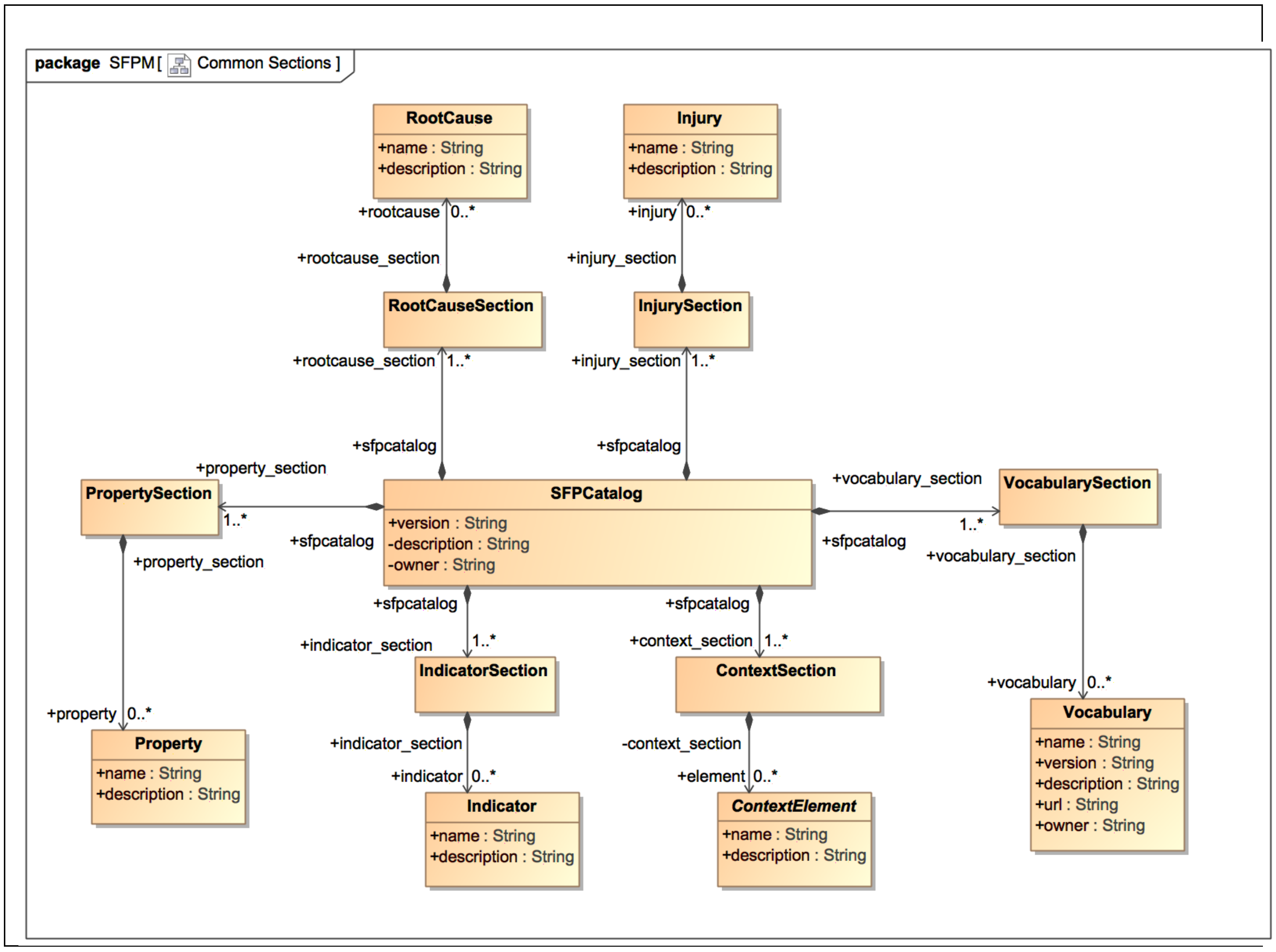

Figure 8. UML class diagram Common Sections

8.2.3.1 RootCauseSection Class

RootCauseSection class is a container for the RootCause elements. All RootCause elements are owned by the SFPCatalog through one or more of the RootCauseSection containers. Individual SFP items reference the RootCause elements as defined in the SFP Causal Context class diagram. The same RootCause element can be referenced by several SFP items. The RootCause class is defined in section 8.1.3.1.

Superclass

CommonSection

Associations

rootcause:RootCause[0..*]	Owned set of RootCause element

Example

See 8.1.3.1

8.2.3.2 InjurySection Class

InjurySection class is a container for the Injury elements. All Injury elements are owned by the SFPCatalog through one or mode InjurySection containers. Individual SFP items reference the Injury elements as defined in the SFP Casual Context class diagram. The same Injury element can be referenced by several SFP items. The Injury class is defined in section 8.1.3.2.

Superclass

CommonSection

Associations

injury:Injury[0..*]	Owned set of Injury element

Example

See 8.1.3.2

8.2.3.3 IndicatorSection Class

IndicatorSection class is a container for the Indicator elements. All Indicator elements are owned by the SFPCatalog through one or mode IndicatorSection containers. Individual SFP items reference the Indicator elements. The same Indicator element can be referenced by several SFP items. Indicator class is a semantic element of the SFP Catalog. This class is further described in section 8.3.

Superclass

CommonSection

Associations

indicator:Indicator[0..*]	Owned set of Indicator element

Example 1. SFPM XMI

```
<indicator_section name="">
    <indicator xmi:type="sfpm:Indicator" xmi:id="cla4"
      name="ordinary pointer dereference read">
            <!—- body omitted --> </indicator>
    <indicator xmi:type="sfpm:Indicator" xmi:id="cla5"
      name="array with index read">
            <!—- body omitted -->  </indicator>
  <!—- body omitted -->
</indicator_section>
```

Example 2. Readable SFP language

```
Indicators
      Indicator "ordinary pointer dereference read"  … End Indicator
      Indicator "array with index read" … End Indicator
```

```
…
End Indicators
```

8.2.3.4 PropertySection Class

PropertySection class is a container for the Property elements. All Property elements are owned by the SFPCatalog through one or more PropertySection containers. Individual SFP items reference the Property elements. The same Property element can be referenced by several SFP items. Property class is a semantic element of the SFP Catalog. This class is further described in section 8.3.

Superclass

CommonSection

Associations

property:Property[0..*]	Owned set of Property element

Example 1. SFPM XMI

```
<property_section name="">
    <property  xmi:type="sfpm:Property" xmi:id="prop1"
      name="access mechanism pointer"> <!—- body omitted --> </property>
    <property  xmi:type="sfpm:Property" xmi:id="prop4"
      name="access mechanism index"> <!—- body omitted --> </property>
      <!—- body omitted -->
</property_section>
```

Example 2. Readable SFP language

```
Properties
  Property "access mechanism pointer" … End Property
  Property "access mechanism index" … End Property
End Properties
```

8.2.3.5 ContextSection Class

ContextSection class is a container for the ContextElement. All ContextElement are owned by the SFPCatalog through one or mode ContextSection containers. Individual SFP items reference the ContextElement in two stages, by first referencing a local ReferencedContextElement which then in turn references a common ContextElement. Local ReferencedContextElement are owned by CharacteristicSection of SFP or one of the Cluster elements that owns the SFP directly or through another Cluster. The set of ReferencedContextElement for an SFP or a Cluster is its "profile". Eventually the same ContextElement can be referenced by several SFP items. This approach allows formal grouping of SFPs based on the characteristics that they share. The analytics can establish the exact nature of the relation between two or more SFPs.

ContextElement class is a semantic element of the SFP Catalog. This class is further described in section 8.3.

Superclass

CommonSection

Associations

element:ContextElement[0..*]   Owned set of Context element

Example 1. SFPM XMI

```
<context_section name="">
    <element xmi:type="sfpm:DataType" xmi:id="shared1"  name="ElementType">
      <definition> <!—- body omitted --> </definition>
    </element>
      <!—- body omitted -->
</context_section>
```

Example 2. Readable SFP language

```
SharedContextElements
  DataType ElementType … End DataType
…
End SharedContextElements
```

8.2.3.6 VocabularySection Class

VocabularySection class is a container for one or more Vocabulary representing a referenced vocabulary. A Vocabulary class owns one or more VocabularyElement. At a minimum, a VocabularyElement is a proxy to some externally defined concept, however it can also have a full formal definition using the formalization apparatus defined in section 9.4. All VocabularyElement are owned by the SFPCatalog through one or mode VocabularySection and Vocabulary containers. Individual SFP items reference the VocabularyElement in SemanticFormulations. Vocabulary class and VocabularyElement class and its subclasses is further described in section 8.5.

Superclass

CommonSection

Associations

vocabulary:Vocabulary[0..*]   Owned set of Vocabulary element

Example

See 8.5.1.1-3

8.2.3.7 SFPCatalog Class (additional properties)

Class diagram Common Sections introduces several additional properties to the SFPCatalog class.

Superclass

Associations

rootcause_section:RootCauseSection[1..*]	RootCause section of the SFP Catalog
injury_section:InjurySection[1..*]	Injury section of the SFP Catalog
property_section:PropertySection[1..*]	Properties of the SFP Catalog
indicator_section:IndicatorSection[1..*]	Indicators of the SFP Catalog
context_section:ContextSection[1..*]	Context elements of the SFP Catalog
vocabulary_section:VocabularySection[1..*]	Referenced vocabularies of the SFP Catalog

Example

See 8.1.1.1

8.2.4 Characteristic Sections Class Diagram

This section describes the sections of the Cluster and SFP containing referenceable definitions (clauses) that are used by the CanonicalSegments. ReferencedContextElements can be owned by SFP or Cluster. This allows introducing local names and scoping. Cluster has more generic referenced elements, SFP has more specific ones if needed. CanonicalForm references these element. CanonicalForm describes how various segments (specific to an SFP and referenced context elements) can be arranged into a coherent piece of source code which "implements" a fault in an appropriate context.

RefernceedContextElement determine the common characteristics of an SFP (and all SFPs in a cluster) and constitute an important part of the overall SFP content. Based on the shared ContextElement, SFP can be systematically grouped into clusters, and the nature of the relationships between different SFPs can be formally described. ContextElement class is described in section 8.3.

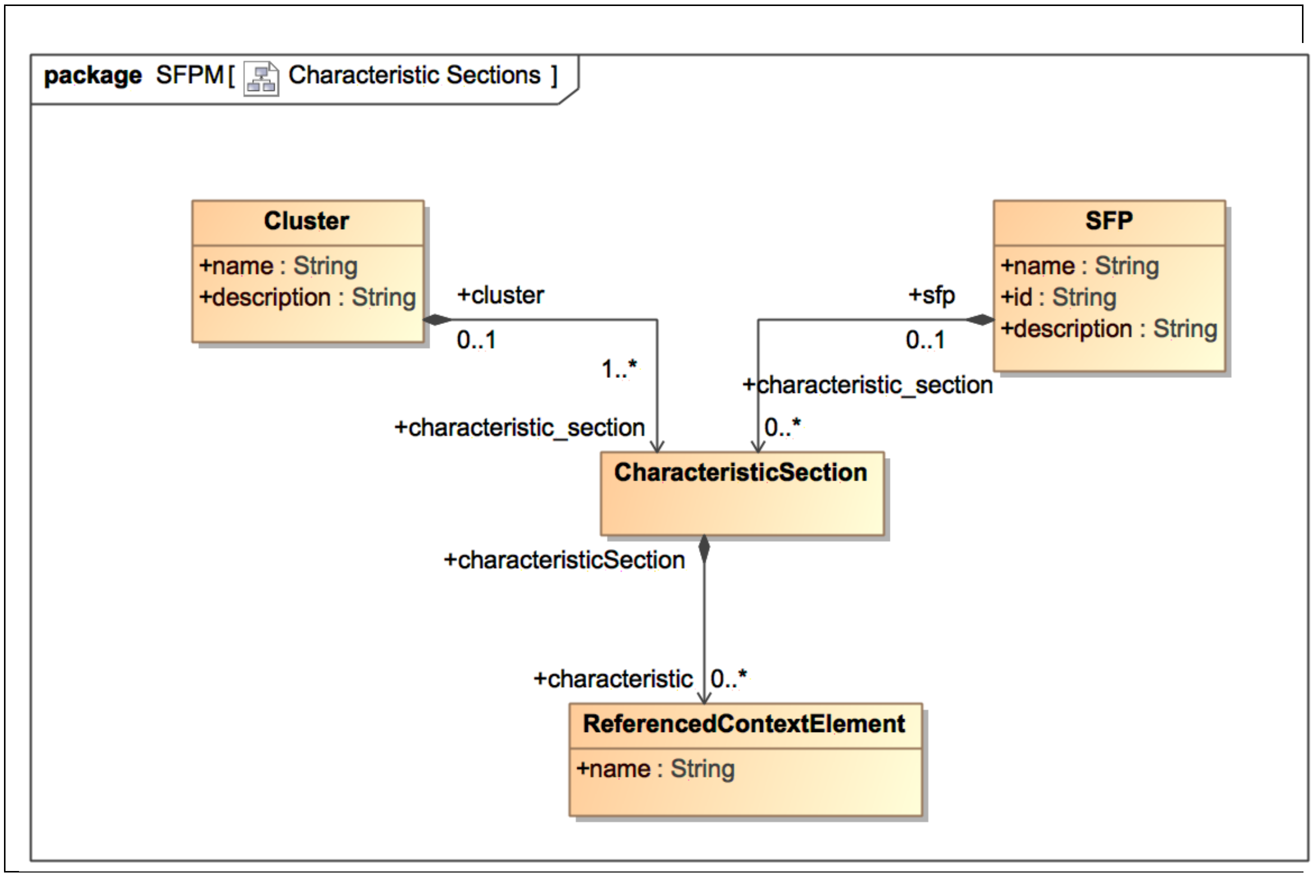

Figure 9. UML class diagram Characteristic Sections

8.2.4.1 CharacteristicSection Class

CharacteristicSection class is a container for zero or more ReferencedContextElement. Local ReferencedContextElement are owned by CharacteristicSection of SFP or one of the Cluster elements that owns the SFP directly or through another Cluster. The set of ReferencedContextElement for an SFP or a Cluster is its "profile". ReferencedContextElement references a common ContextElement. All ContextElement are owned by the SFPCatalog through one or mode ContextSection containers. Thus, individual SFP items reference the ContextElement in two stages, by first referencing a local ReferencedContextElement which then in turn references a common ContextElement (illustrated in more detail in section 8.3.1). Eventually the same ContextElement can be referenced by several SFP items. This approach allows formal grouping of SFPs based on the characteristics that they share. The analytics can establish the exact nature of the relation between two or more SFP. ContextElement class is a semantic element of the SFP Catalog, with a formal semantic definition in the form of a common logic statement on top of the KDM vocabulary. The ContextElement class is further described in section 8.3.

Superclass

ClusterSection

Associations

characteristic:ReferencedContextElement[0..*]   Owned set of characteristics

Constraints

1. Owned elements of a CharacteristicSection shall only reference a local ReferenceContextElement and shall not reference any ContextElement in common ContextSections of the SFPCatalog. An element references another element either directly or indirectly in its owned semantic definition. A ReferencedContextElement is local when it is owned by a CharacteristicSection of the SFP or the (secondary) Cluster that owns the SFP or the (primary) Cluster that owns the (secondary) Cluster that owns the SFP.

Example 1. SFPM XMI

```
<characteristic_section name="">
  <characteristic xmi:id="cla25" element="shared1" name="ElementType"/>
  <characteristic xmi:id="cla26" element="shared2" name="TargetBuffer"/>
  <characteristic xmi:id="cla27" element="shared3" name="TargetBufferType"/>
  <characteristic xmi:id="cla28" element="shared4" name="BufferPointerType"/>
  <characteristic xmi:id="cla29" element="shared5" name="BufferPointer"/>
  <characteristic xmi:id="cla30" element="shared6" name="BufferOffset"/>
  <characteristic xmi:id="cla31" element="shared7" name="BufferLength"/>
  <characteristic xmi:id="cla32" element="shared8" name="DataLengthGood"/>
  <characteristic xmi:id="cla33" element="shared9" name="DefineData"/>
  <characteristic xmi:id="cla34" element="shared10" name="DefineIndex"/>
  <characteristic xmi:id="cla35" element="shared11" name="DefineTargetBuffer"/>
  <characteristic xmi:id="cla36" element="shared12"
       name="BindPointerToTargetBuffer"/>
  <characteristic xmi:id="cla37" element="shared13" name="ReleaseTargetBuffer"/>
  <characteristic xmi:id="cla38" element="shared14" name="Cleanup"/>
  <characteristic xmi:id="cla39" element="shared15" name="DefineValidReference"/>
</characteristic_section>
```

Example 2. Readable SFP language

```
Characteristics

      Ref DataType  ElementType

      Ref Resource TargetBuffer
      Ref DataType  TargetBufferType

      Ref DataType BufferPointerType
      Ref DataElement BufferPointer
      Ref DataElement BufferOffset

      Ref DataElement BufferLength
      Ref DataElement DataLengthGood

      Ref DataElement DefineData
      Ref DataElement DefineIndex

      Ref Operation DefineTargetBuffer
```

```
     Ref Operation BindPointerToTargetBuffer
     Ref Operation ReleaseTargetBuffer
     Ref Operation Cleanup
     Ref Operation DefineValidReference

End Characteristics
```

8.2.4.2 Cluster Class (additional properties)

Class diagram Characteristic Sections introduces several additional properties to the Cluster class.

Superclass

Associations

characteristic_section:CharacteristicSection[1..*]	Owned set of characteristics for the cluster

Example

See 8.1.6.1

8.2.4.3 SFP Class (additional properties)

Class diagram Characteristic Sections introduces several additional properties to the SFP class.

Superclass

Associations

characteristic_section:CharacteristicSection[1..*]	Owned set of characteristics for the SFP

Example

See 8.1.6.1

8.3 SFP Defined Elements

This section describes the framework for the formal semantic definitions of the faulty computations represented by the core elements of the SFP Catalog. These elements constitute the formal semantic content of the SFP catalog.

These elements specify dataflows that constitute the *extension* of an SFP as a concept. "Extension" is the totality of objects to which a concept corresponds. According to the SFP approach, the "objects" of software weaknesses are dataflows implemented in code. SFP items are denotations (semantic definitions) for families of dataflows that correspond to the classes of software weaknesses introduced by the CWE catalog. CWE catalog provides signifiers to the software weaknesses. SFP provides formal semantic definitions to a subset of software weaknesses in CWE catalog, and links these definitions to the corresponding CWE items. Formal semantic definitions of DataflowElement are given using the formalization apparatus defined in section 8.4. An overview of the SFP approach to formal semantics of dataflows is given in the introduction to this specification.

8.3.1 SFP Defined Elements Class Diagram

This section describes the elements of the SFP Catalog that have a formal semantic definition and constitute the formalization framework of the SFP Catalog. These elements are Properties, Indicators, SFP DataflowElements, ContextElements and CanonicalElements. In addition, some VocabularyElements can be formally defined using the same formalization apparatus.

SFP DataflowElements describe each SFP as a data flow with a primary data element, source and sink. The faulty computation is assumed to involve the values of the data element and "flow" from the source to the sink. This approach is based on the best practices of the community.

The SFP approach focuses at the discernible "places" in the code that are "indicators" of particular computations. The indicators may describe places in the code that implement operations directly linked to some injury (for example, access to a buffer is a necessary condition for a buffer overflow), or describe important regions of the code based on its purpose, such as common safeguards, authentication, access control, privilege management, cryptography, data validation, memory management, resource management, exception management, etc. Each discernible SFP includes some Indicators that provide a starting point for identifying the presence of the SFP in the code under assessment. The rest of the SFP definition includes a set of propositions that may be eventually traced to the code, always in relation to the Indicator. Usually SFP definitions involve a condition that must be satisfied to make the claim that the SFP has been detected in code.

The concept of a data flow with a data element, source, sink and the invariant condition is central to SFP formalization. SFP DataflowElements capture these elements. Indicator element represent discernible "places" in code. Typically, the sink of an SFP is defined a disjunction of references to Indicators.

The content of the SFP Catalog describes an argument justifying the claim that the code under assessment exhibits a certain fault. The starting point of this argument is the presence of the Indicator. Additional evidence is provided by matching of the elements of the SFP in relation to the Indicator. Final evidence is collected when the data flow satisfies the condition of the SFP.
An invariant of a data flow can be described as a set of propositions such that any "compliant" data flow will exhibit these propositions, and only compliant data flows will exhibit such propositions. Thus, the SFP Catalog accumulates content related to describing "interesting" data flows.

CanonicalElements define a broader context for faulty computations, sufficient to generate complete (compilable, executable) examples of the faulty computation in the form of a test case. These test cases can be used to validate CWE compliance mappings of existing and future Static Code Analysis tools. The ContextElements represent reusable elements that are used in CanonicalElements. CanonicalElements also represent "mitigated" computations that shared significant fragments with faulty computations but do not exhibit the fault. The latter content can be used to generate additional test cases for the "false positives" reporting in existing and future Static Code Analysis tools.

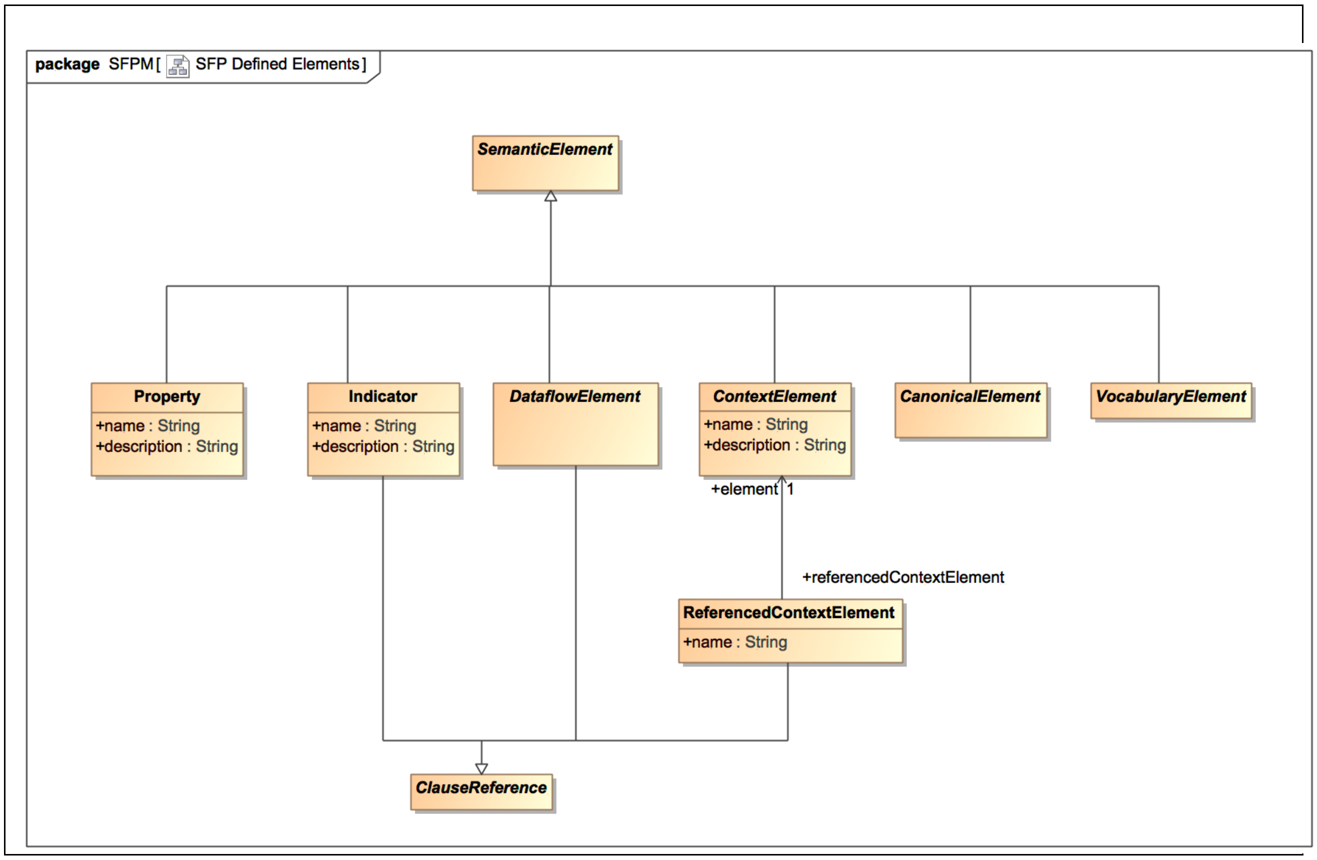

Figure 10. UML class diagram SFP Defined Elements

8.3.1.1 Property Class

Property is a special semantic element that contains statements only from a special vocabulary of tags and values (markers). This vocabulary is usually aligned with the implementation capabilities. Properties is a mechanism that is used to manage the variations of the SFP. Typically, a family of related faulty computations that exhibit a certain fault (identified as an SFP), involves many “variations” that share many common elements of the overall data flow. The overall structure of the variants of the family is defined as a set of SFP Parameters, their Variants, and is represented by a tree of Variations. SFP data element, source, and sink are usually defined as disjunctions of statements, each focusing at a certain specific case. The connections between these cases and the variants of the SFP is made through Properties which are certain tags and values (markers) added to the KDM Fragments.

SFP uses a completely generic formalization mechanism to extend the semantics of other elements with tags and markers. Thus, the formalization uses an SBVR statement “Thing1 is Thing2” where the role of “Thing1” is bound to a tag, and the role “Thing2” is bound to a value. Both tag and value are defined as Individual concepts in a special vocabulary in the VocabularySection.

The SFP implementation may use the tag and value definitions of properties in a multitude of ways, for example as annotations, markers, metadata for variants, or executable clauses that use single assignment to prevent synthesis of incompatible variants, or to cut matching incompatible subtrees of the semantic definition.

The vocabulary of property tags and values in not part of the SFP Metamodel, and should be explained in the SFP catalog.

Superclass

SemanticElement

Attributes

name:String[1]	Name of the property
description:String[1]	Description of the property

Example 1. SFPM XMI

```
<property  xmi:type="sfpm:Property" xmi:id="prop1" name="access mechanism
pointer">
  <definition>
    <meaning xmi:id="sem1208" identificator="property access mechanism pointer"
            kind="SetProjection" description="" >
      <variable xmi:id="var492" range="nc29" name="phantom"/>
      <operand xmi:id="sem1209" identificator="" kind="Conjunction"
              description="">
         <operand xmi:id="sem1210" verb="vc4" identificator=""
                  kind="AtomicFormulation" description="">
           <binding rolename="Thing1" target="ic93"/>
           <binding rolename="Thing2" target="ic103"/>
         </operand>
         <operand xmi:id="sem1211" verb="vc4" identificator=""
                   kind="AtomicFormulation" description="">
           <binding rolename="Thing1" target="ic19"/>
           <binding rolename="Thing2" target="ic5"/>
         </operand>
         <operand xmi:id="sem1212" verb="vc4" identificator=""
                  kind="AtomicFormulation" description="">
           <binding rolename="Thing1" target="ic18"/>
           <binding rolename="Thing2" target="ic5"/>
         </operand>
         <operand xmi:id="sem1213" verb="vc4" identificator=""
                   kind="AtomicFormulation" description="">
           <binding rolename="Thing1" target="ic8"/>
           <binding rolename="Thing2" target="ic3"/>
         </operand>
      </operand>
    </meaning>
  </definition>
</property>

<verb xmi:id="vc4" name="Thing1 is Thing2"/>

<vocabulary name="Platform Meta">
    <individual xmi:id="ic93" name="core.bufferaccessmechanism"/>
    <individual xmi:id="ic103" name="pointerdereference"/>
</vocabulary>
```

Example 2. Readable SFP language

```
Property "access mechanism pointer"
      [meta] core.bufferaccessmechanism,pointerdereference
      [meta] isindex,no
      [meta] isapi,no
      [meta] buffermode,regular
End Property
```

8.3.1.2 Indicator Class

The SFP approach focuses at the discernible "places" in the code that are "indicators" of computations. The indicators may describe places in the code that implement operations directly linked to some injury (for example, access to a buffer is a necessary condition for a buffer overflow), or describe important regions of the code based on its purpose, such as common safeguards, authentication, access control, privilege management, cryptography, data validation, memory management, resource management, exception management, etc. Each discernible SFP includes some Indicators that provide a starting point for identifying the presence of the SFP in the code under assessment. The rest of the SFP definition includes a set of propositions that may be eventually matched to the code, always in relation to the Indicator.

Superclass

SemanticElement, ClauseReference

Attributes

name:String[1]	Name of the indicator
description: String[1]	Description of the indicator

Example 1. SFPM XMI

```
<indicator xmi:type="sfpm:Indicator" xmi:id="cla4" name="ordinary pointer
dereference read">
  <definition>
    <meaning xmi:id="sem1388" kind="SetProjection"
            description="Definition of indicator ordinary pointer
                  dereference read" >
      <variable xmi:id="var514" range="nc4" name="S1"/>
      <variable xmi:id="var515" range="nc4" name="S2"/>
      <variable xmi:id="var516" range="nc2" name="BP"/>
      <variable xmi:id="var517" range="nc2" name="BPTI"/>
      <variable xmi:id="var518" range="nc2" name="Data"/>
      <operand xmi:id="sem1389"
            identificator="ordinary pointer dereference read"
            kind="Conjunction" description="">
        <operand xmi:id="sem1390" verb="vc4" identificator=""
                   kind="AtomicFormulation" description="">
          <binding rolename="Thing1" target="ic26"/>
          <binding rolename="Thing2" target="ic27"/>
        </operand>
        <operand xmi:id="sem1391" verb="vc4" identificator=""
             kind="AtomicFormulation" description="">
          <binding rolename="Thing1" target="ic86"/>
          <binding rolename="Thing2" target="ic87"/>
        </operand>
        <operand xmi:id="sem1392" verb="vc4" identificator=""
             kind="AtomicFormulation" description="">
          <binding rolename="Thing1" target="ic93"/>
          <binding rolename="Thing2" target="ic103"/>
        </operand>
        <operand xmi:id="sem1393" verb="vc4" identificator=""
                  kind="AtomicFormulation" description="">
          <binding rolename="Thing1" target="ic8"/>
          <binding rolename="Thing2" target="ic3"/>
        </operand>
        <operand xmi:id="sem1394" verb="vc4" identificator=""
                   kind="AtomicFormulation" description="">
          <binding rolename="Thing1" target="ic143"/>
          <binding rolename="Thing2" target="ic144"/>
        </operand>
        <operand xmi:id="sem1395" verb="vc109" identificator=""
                   kind="AtomicFormulation" description="">
          <binding rolename="ActionElement" target="var514"/>
        </operand>
        <operand xmi:id="sem1396" verb="vc9" identificator=""
                   kind="AtomicFormulation" description="">
          <binding rolename="ActionElement" target="var514"/>
          <binding rolename="DataElement" target="var516"/>
        </operand>
        <operand xmi:id="sem1397" verb="vc6" identificator=""
                  kind="AtomicFormulation" description="">
          <binding rolename="ActionElement" target="var514"/>
          <binding rolename="DataElement" target="var517"/>
        </operand>
        <operand xmi:id="sem1398" verb="vc7" identificator=""
                   kind="AtomicFormulation" description="">
          <binding rolename="ActionElement" target="var514"/>
          <binding rolename="DataElement" target="var518"/>
        </operand>
        <operand xmi:id="sem1399" verb="vc4" identificator=""
                   kind="AtomicFormulation" description="">
          <binding rolename="Thing1" target="var515"/>
          <binding rolename="Thing2" target="var514"/>
        </operand>
     </operand>
    </meaning>
  </definition>
</indicator>
```

Example 2. Readable SFP language

```
Indicator "ordinary pointer dereference read"

      Var S1 : ActionElement [KDM] ;;; segment Begin
      Var S2 : ActionElement [KDM] ;;; segment End
      Var BP: DataElement [KDM]
      Var BPTI: DataElement [KDM]
      Var Data: DataElement [KDM]

      Clause "ordinary pointer dereference read"
        # data=*p;
              [meta] platform,c or cpp
              [meta] core.bufferaccess,read
              [meta] core.bufferaccessmechanism,pointerdereference
              [meta] buffermode,regular
              [meta] core.indicator,deref_read

              [ActionElement is ptrselect :KDM] S1
              [ActionElement addresses DataElement :KDM] S1,BP
              [ActionElement reads DataElement :KDM] S1, BPTI
              [ActionElement writes DataElement :KDM] S1,Data
              [Thing1 is Thing2 :SBVR] S2, S1

End Indicator
```

8.3.1.3 ReferencedContextElement Class

ReferencedContextElement class represents an element of a conceptual "profile" of an SFP defined in terms of common ContextElement. Local ReferencedContextElement are owned by CharacteristicSection of SFP or one of the Cluster elements that owns the SFP directly or through another Cluster. The set of ReferencedContextElement for an SFP or a Cluster is its "profile". ReferencedContextElement references a common ContextElement. All ContextElement are owned by the SFPCatalog through one or mode ContextSection containers. Thus, individual SFP items reference the ContextElement in two stages, by first referencing a local ReferencedContextElement which then in turn references a common ContextElement. Eventually the same ContextElement can be referenced by several SFP items.

The purpose of a local ReferencedContextElement is to provide visibility to the common characteristics of multiple SFPs and to the conceptual "profile" of an SFP. This approach allows formal grouping of SFPs based on the shared characteristics. Analytics can establish the exact nature of the relation between two or more SFP based on the shared ContextElement as well as other content, such as Property or Indicator. ContextElement class is a semantic element of the SFP Catalog. This class is further described in section 8.3.

Superclass

ClauseReference

Attributes

name:String[1]	Name of the referenced context element

Associations

element: ContextElement[1]		Reference to a common context element

Example 1. SFPM XMI

```
<characteristic_section name="">
  <characteristic xmi:id="cla25" element="shared1" name="ElementType"/>
… </characteristics_section>

<context_section name="">
    <element xmi:type="sfpm:DataType" xmi:id="shared1"  name="ElementType">
      <definition>
        <meaning xmi:id="sem131" kind="SetProjection"
            description="Definition of DataType ElementType" >
          <variable xmi:id="var143" range="nc1" name="DT"/>
          <operand xmi:id="sem132" identificator=""
                  kind="ExistentialQuantification" description="">
            <variable xmi:id="var144" range="nc8" name="T">
              <restriction xmi:id="sem133" verb="vc2" identificator=""
                         kind="AtomicFormulation" description="">
                <binding rolename="KDMEntity" target="var144"/>
                <binding rolename="Name" target="ic11"/>
              </restriction>
            </variable>
            <operand xmi:id="sem134" identificator="" kind="Conjunction"
                         description="">
              <operand xmi:id="sem135" verb="vc4" identificator=""
                        kind="AtomicFormulation" description="">
                <binding rolename="Thing1" target="ic12"/>
                <binding rolename="Thing2" target="ic13"/>
              </operand>
              <operand xmi:id="sem136" verb="vc4" identificator=""
                         kind="AtomicFormulation" description="">
                <binding rolename="Thing1" target="var143"/>
                <binding rolename="Thing2" target="var144"/>
              </operand>
            </operand>
          </operand>
        </meaning>
      </definition>
    </element>
<!-- body omitted -->
</context_section>
```

Example 2. Readable SFP language

```
Characteristics

      Ref DataType  ElementType
…

 End Characteristics

SharedContextElements

  DataType ElementType
```

```
        Var DT : DataType [KDM]
        Clause
              Var T : CharType [KDM] such that
                    [KDMEntity has Name :KDM] T, {"char": Strings}
              where
                   [meta] complexity.datatype,char
                    [Thing1 is Thing2 :SBVR] DT, T

  End DataType
…
End SharedContextElements
```

8.3.2 SFP Dataflow Elements Class Diagram

This section describes the main structural parts of a faulty computation as a data flow. The challenge in describing software faults is to manage complexity of possible computations that may exhibit the given fault. The main dimensions of the set of computations that exhibit a given fault include

- Programming language
- Code libraries and components
- Runtime environment, operating system
- Lexical variations (e.g. names of variables)
- Semantic variations of the indicator
- Semantic variations of the overall data flow
- The context into which the faulty computation is embedded

The SFP approach follows community's best practices in providing machine-consumable descriptions of software faults based on common data flows (for example Juliette test cases). According to this approach, the invariant of a faulty computation is a certain data flow, which has several structural parts:

- The data statement
- The sink statement
- The source statement

As an invariant of a faulty computation, the data flow has a certain condition that involves the data element, and the source and sink statements. Some common terminology is reviewed and illustrated in the introduction to this specification.

Each of the elements above is an SFP DefinedElement, such that it has a semantic definition in the form of the SFP SemanticFormulation, further defined in section 8.4. Thus, a "statement" can be a conjunction of several logical propositions.

The data element determines that set of values that flow from the source statement to the sink statement. The scope of the data flow is restricted to the activities that occur at the source statement, followed by the activities at the sink statement. In other words, the data flow is assumed to flow from the source statement to the sink statement. Further, it is assumed that any activities between the source and the sink do not affect the values of the data element. The compliant data flows in the "extension" of the semantic definition can be interleaved with any other activities as long as they do not violate the above assumptions.

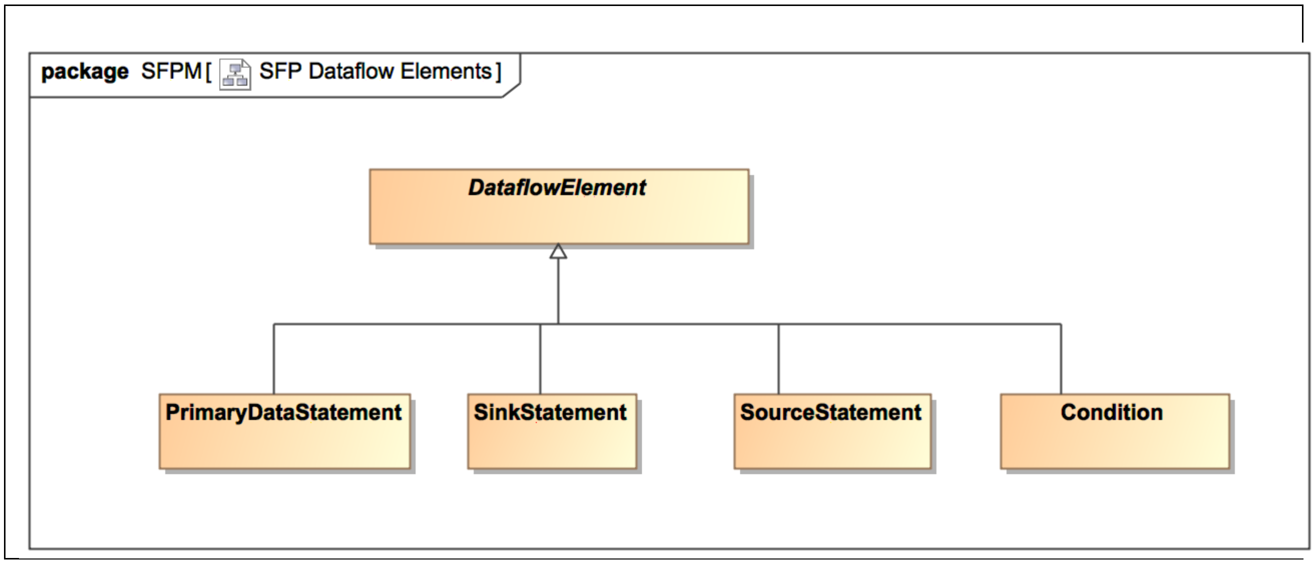

Figure 11. UML class diagram SFP Dataflow Elements

8.3.2.1 DataflowElement Class (abstract)

DataflowElement is a parent class for several elements that define the structural parts of faulty computations as an invariant of a data flow.

Superclass

SemanticElement, ClauseReference

8.3.2.2 PrimaryDataStatement Class

PrimaryDataStatement class represents the data element of the data flow that constitutes an invariant of the family of faulty computations collectively described as an SFP.

PrimaryDataStatements are usually defined as SetProjections, involving one or more variables that "range" over some concepts (see example below). Semantic formulations are based on the SBVR specification and are described in more detail in Section 8.4

Superclass

DataflowElement

Example 1. SFPM XMI

```
<element xmi:type="sfpm:PrimaryDataStatement" xmi:id="cla1">
```

```
            <definition>
              <meaning xmi:id="sem1" kind="SetProjection"
                  description="Definition of primary data statement" >
                <variable xmi:id="var1" range="nc1" name="BPBT"/>
                <variable xmi:id="var2" range="nc2" name="TBTI"/>
                <variable xmi:id="var3" range="nc2" name="BP"/>
                <variable xmi:id="var4" range="nc1" name="BPT"/>
                <variable xmi:id="var5" range="nc2" name="BPTI"/>
                <operand xmi:id="sem2" identificator="pointer"
                         kind="ExistentialQuantification" description="">
                  <variable xmi:id="var6" range="nc3" name="PT">
                    <restriction xmi:id="sem3" verb="vc1" identificator=""
                         kind="AtomicFormulation"
                        description=""> … </restriction>
                  </variable>
                  <variable xmi:id="var7" range="nc2" name="SU">
                    <restriction xmi:id="sem4" identificator=""
                              kind="Conjunction" description="">
                              <!—- body omitted -->
                        </restriction>
                  </variable>
                  <operand xmi:id="sem7" identificator="" kind="Conjunction"
                        description="">
                        <!—- body omitted -->
                        </operand>
                </operand>
              </meaning>
            </definition>
</element>
```

Example 2. Readable SFP language

```
PrimaryDataStatement

      Var BPBT: DataType [KDM]  ;;; target buffer base type (in)
      Var TBTI: DataElement [KDM]  ;;; target buffer type item (in)
      Var BP: DataElement [KDM]  ;;; buffer pointer (out)
      Var BPT: DataType [KDM]  ;;; buffer pointer type (out)
      Var BPTI: DataElement [KDM] ;;; buffer pointer item (out)

      Clause "pointer"
        # BPBT * p;
             Var PT : PointerType [KDM] such that
                   [Type is a pointer to BaseType with ItemUnit:KDM Patterns] PT,
                        BPBT, BPTI
             Var SU :DataElement [KDM] such that
                   [ KDMEntity has Name :KDM] SU, {"p": Strings}
                   [DataElement has type DataType :KDM] SU, PT
             where
               [meta] pointermode,regular
               [Thing1 is Thing2 :SBVR] BP, SU
               [Thing1 is Thing2 :SBVR] BPT, PT

End PrimaryDataStatement
```

8.3.2.3 SinkStatement Class

SinkStatement class represents the sink of the data flow that constitutes an invariant of the family of faulty computations collectively described as an SFP.

SinkStatements are defined as SetProjections, involving one or more variables that “range” over some concepts (see example below). A SetProjection “considers” another proposition. Semantic formulations are based on the SBVR specification and are described in more detail in Section 8.4.

Superclass

DataflowElement

Constraints

1. Each SinkStatement shall be defined as a SetProjection that *considers* a Disjunction in which the clauses are references to Indicator elements

Example 1. SFPM XMI

```
<element xmi:type="sfpm:SinkStatement" xmi:id="cla3">
  <definition>
    <meaning xmi:id="sem30" kind="SetProjection"
            description="Definition of sink statement" >
      <variable xmi:id="var15" range="nc4" name="S1"/>
      <variable xmi:id="var16" range="nc4" name="S2"/>
      <variable xmi:id="var17" range="nc5" name="BK"/>
      <variable xmi:id="var18" range="nc1" name="TBT"/>
      <variable xmi:id="var19" range="nc1" name="BPT"/>
      <variable xmi:id="var20" range="nc2" name="BPTI"/>
      <variable xmi:id="var21" range="nc2" name="TBTI"/>
      <variable xmi:id="var22" range="nc1" name="BPBT"/>
      <variable xmi:id="var23" range="nc1" name="DT"/>
      <variable xmi:id="var24" range="nc2" name="BP"/>
      <variable xmi:id="var25" range="nc2" name="Data"/>
      <variable xmi:id="var26" range="nc2" name="DataLength"/>
      <variable xmi:id="var27" range="nc2" name="Index"/>
      <operand xmi:id="sem31" identificator="" kind="Disjunction" description="">
         <operand xmi:id="sem32" identificator="Read Access"
                 kind="ExistentialQuantification" description="">
            <variable xmi:id="var28" range="nc1" name="F1"/>
            <operand xmi:id="sem33" identificator="" kind="Disjunction"
                        description="">
               <operand xmi:id="sem34" identificator="Explicit Access"
                        kind="ExistentialQuantification" description="">
                  <variable xmi:id="var29" range="nc1" name="F2"/>
                  <operand xmi:id="sem35" identificator="" kind="Disjunction"
                              description="">
                     <operand xmi:id="sem36" verb="cla4" kind="AtomicFormulation"
                              description="ordinary pointer dereference read">
                        <binding rolename="S1" target="var15"/>
                        <binding rolename="S2" target="var16"/>
                        <binding rolename="BP" target="var24"/>
                        <binding rolename="BPTI" target="var20"/>
```

```
                <binding rolename="Data" target="var25"/>
             </operand>
             <operand xmi:id="sem37" verb="cla5" kind="AtomicFormulation"
                 description="array with index read">
                <binding rolename="S1" target="var15"/>
                <binding rolename="S2" target="var16"/>
                <binding rolename="BP" target="var24"/>
                <binding rolename="TBTI" target="var21"/>
                <binding rolename="Index" target="var27"/>
                <binding rolename="Data" target="var25"/>
             </operand>
             <operand xmi:id="sem38" verb="cla6" kind="AtomicFormulation"
                       description="struct member read">
                <binding rolename="S1" target="var15"/>
                <binding rolename="S2" target="var16"/>
                <binding rolename="BP" target="var24"/>
                <binding rolename="BPBT" target="var22"/>
                <binding rolename="BPTI" target="var20"/>
                <binding rolename="TBTI" target="var21"/>
                <binding rolename="Data" target="var25"/>
             </operand>
             <operand xmi:id="sem39" verb="cla7" kind="AtomicFormulation"
                      description="class member read">
                <binding rolename="S1" target="var15"/>
                <binding rolename="S2" target="var16"/>
                <binding rolename="BP" target="var24"/>
                <binding rolename="TBTI" target="var21"/>
                <binding rolename="Data" target="var25"/>
             </operand>
             <operand xmi:id="sem40" verb="cla8" kind="AtomicFormulation"
                      description="cast read">
                <binding rolename="S1" target="var15"/>
                <binding rolename="S2" target="var16"/>
                <binding rolename="DT" target="var23"/>
                <binding rolename="BP" target="var24"/>
                <binding rolename="BPBT" target="var22"/>
                <binding rolename="Data" target="var25"/>
             </operand>
          </operand>
        </operand>
        <operand xmi:id="sem41" verb="cla9" kind="AtomicFormulation"
                      description="overlay struct read">
           <binding rolename="S1" target="var15"/>
           <binding rolename="S2" target="var16"/>
           <binding rolename="DT" target="var23"/>
           <binding rolename="BP" target="var24"/>
           <binding rolename="BPBT" target="var22"/>
           <binding rolename="Data" target="var25"/>
        </operand>
        <operand xmi:id="sem42" identificator="Hidden Access via api"
                kind="ExistentialQuantification" description="">
                <!—body omitted --> </operand>
</operand>
<operand xmi:id="sem46" identificator="Write Access"
         kind="ExistentialQuantification" description="">
                <!—body omitted -->  </operand>
<operand xmi:id="sem60" identificator="Call access"
```

```
                  kind="ExistentialQuantification" description="">
                  <!—body omitted -->  </operand>
      </operand>
    </meaning>
  </definition>
</element>
```

Example 2. Readable SFP language

```
SinkStatement
      Var S1 : ActionElement [KDM] ;;; segment Begin
      Var S2 : ActionElement [KDM] ;;; segment End

      Var BK: TargetBufferKind [Platform Meta]
      Var TBT: DataType [KDM]
      Var BPT: DataType [KDM]
      Var BPTI: DataElement [KDM]
# TBTI can be a Pointer ItemUnit, an Array ItemUnit or a class MemberUnit
      Var TBTI: DataElement [KDM]
      Var BPBT: DataType [KDM]
      Var DT: DataType [KDM]
      Var BP: DataElement [KDM]
      Var Data: DataElement [KDM]
      Var DataLength : DataElement [KDM]
      Var Index: DataElement [KDM]

      Disjunction
            Clause "Read Access"
                  Var F1 : DataType [KDM]
                  Disjunction
                        Clause "Explicit Access"
                              Var F2 : DataType [KDM]
                              Disjunction
                                    Clause [ordinary pointer dereference read]
                                           S1=S1, S2=S2,
                                          BP=BP, BPTI=BPTI, Data=Data
                                    Clause [array with index read] S1=S1, S2=S2,
                                           BP=BP, TBTI=TBTI,
                                           Index=Index, Data=Data
                                    Clause  [struct member read] S1=S1, S2=S2,
                                                BP=BP, BPBT=BPBT, BPTI=BPTI,
                                                 TBTI=TBTI, Data=Data
                                    Clause  [class member read] S1=S1, S2=S2,
                                                 BP=BP, TBTI=TBTI, Data=Data
                                    Clause  [cast read] S1=S1, S2=S2,
                                                DT=DT, BP=BP, BPBT=BPBT,
                                                 Data=Data
                              End Disjunction

                        Clause  [overlay struct read] S1=S1, S2=S2,
                              DT=DT, BP=BP, BPBT=BPBT, Data=Data
                        Clause "Hidden Access via api"
                              …

                  End Disjunction
            Clause "Write Access"
                  …

            Clause "Call access"
                  …

End SinkStatement
```

8.3.2.4 SourceStatement Class

SourceStatement class represents the source of the data flow that constitutes an invariant of the family of faulty computations collectively described as an SFP.

SourceStatements are usually defined as SetProjections, involving one or more variables that "range" over some concepts (see example below). A SetProjection "considers" another proposition. Semantic formulations are based on the SBVR specification and are described in more detail in Section 8.4.

Superclass

DataflowElement

Example 1. SFPM XMI

```
<element xmi:type="sfpm:SourceStatement" xmi:id="cla2">
  <definition>
    <meaning xmi:id="sem11" kind="SetProjection"
            description="Definition of source statement" >
       <variable xmi:id="var8" range="nc4" name="S1"/>
       <variable xmi:id="var9" range="nc4" name="S2"/>
       <variable xmi:id="var10" range="nc5" name="BK"/>
       <variable xmi:id="var11" range="nc2" name="TB"/>
       <variable xmi:id="var12" range="nc1" name="BPT"/>
       <variable xmi:id="var13" range="nc1" name="BPBT"/>
       <variable xmi:id="var14" range="nc2" name="BP"/>
       <operand xmi:id="sem12" identificator="" kind="Disjunction"
            description="">
         <operand xmi:id="sem13" identificator="assign" kind="Conjunction"
                  description="">
           <operand xmi:id="sem14" verb="vc4" identificator=""
                        kind="AtomicFormulation" description="">
              <binding rolename="Thing1" target="ic4"/>
              <binding rolename="Thing2" target="ic5"/>
           </operand>
           <operand xmi:id="sem15" verb="vc4" identificator=""
                        kind="AtomicFormulation" description="">
              <binding rolename="Thing1" target="ic6"/>
              <binding rolename="Thing2" target="ic7"/>
           </operand>
           <operand xmi:id="sem16" verb="vc4" identificator=""
                        kind="AtomicFormulation" description="">
             <binding rolename="Thing1" target="ic8"/>
             <binding rolename="Thing2" target="ic3"/>
           </operand>
```

```
            <operand xmi:id="sem17" verb="vc5" identificator=""
                    kind="AtomicFormulation" description="">
               <binding rolename="ActionElement" target="var8"/>
            </operand>
            <operand xmi:id="sem18" verb="vc6" identificator=""
                   kind="AtomicFormulation" description="">
               <binding rolename="ActionElement" target="var8"/>
               <binding rolename="DataElement" target="var11"/>
            </operand>
            <operand xmi:id="sem19" verb="vc7" identificator=""
                   kind="AtomicFormulation" description="">
               <binding rolename="ActionElement" target="var8"/>
               <binding rolename="DataElement" target="var14"/>
            </operand>
            <operand xmi:id="sem20" verb="vc4" identificator=""
                   kind="AtomicFormulation" description="">
               <binding rolename="Thing1" target="var9"/>
               <binding rolename="Thing2" target="var8"/>
            </operand>
         </operand>
             <!—- body omitted -->
      </operand>
    </meaning>
  </definition>
</element>
```

Example 2. Readable SFP language

```
SourceStatement

      Var S1: ActionElement [KDM]
      Var S2: ActionElement [KDM]
      Var BK: TargetBufferKind [Platform Meta]
      Var TB: DataElement [KDM]
      Var BPT: DataType [KDM]
      Var BPBT: DataType [KDM]
      Var BP: DataElement [KDM]

      Disjunction
            Clause "assign"
            #  p=buf;
                  [meta] complexity.inline,no
                  [meta] isnamed,yes
                  [meta] buffermode,regular
                  [ActionElement is assign :KDM] S1
                  [ActionElement reads DataElement :KDM] S1, TB
                  [ActionElement writes DataElement :KDM] S1, BP
                  [Thing1 is Thing2 :SBVR] S2, S1

            Clause "address"
            # p=&buf;
                  [meta] complexity.inline,no
                  [meta] isnamed,yes
                  [meta] buffermode,struct
                  [ActionElement is ptr :KDM] S1
```

```
                [ActionElement addresses DataElement :KDM] S1,TB
                [ActionElement writes DataElement :KDM] S1,BP
                [Thing1 is Thing2 :SBVR] S2, S1

          Clause "release"
                [ simple Begin End releases Buffer of DataType:KDM Patterns]
                       S1, S2,  BP, BPT
      End Disjunction

End SourceStatement
```

8.3.2.5 Condition Class

Condition class represents the invariant condition of the data flow for the family of faulty computations collectively described as an SFP. While SinkStatement, SourceStatement and PrimaryDataStatement focus on program point patterns, Condition allows specification of properties that involve values (state) of the computation. In general, specification based on values assumes a more powerful class of supporting capabilities.

Superclass

DataflowElement

8.3.3 SFP Canonical Elements Class Diagram

This section describes the Canonical Elements of the SFP Catalog. In contrast to the Dataflow Element that define the invariant of some data flow, CanonicalElements define *canonical* dataflows by providing their full context. CanonicalElements are aligned with the Dataflow elements and define dataflows that exhibit the fault described by the SFP, as well as related dataflows that share certain significant parts of the invariant without exhibiting the fault. The latter dataflows are referred to as “mitigated dataflows” as they illustrate possible “mitigations” of the fault.

The intention of the Canonical Elements is to synthesize the test cases that correspond to the software weaknesses defined by the SFP items. CanonicalElements define a broader context for faulty computations, sufficient to generate complete (compilable, executable) examples of the faulty computation in the form of a test case with appropriate metadata. These test cases can be used to validate CWE compliance mappings of existing and future Static Code Analysis tools. The ContextElements represent reusable elements that are used in CanonicalElements. CanonicalElements also represent “mitigated” computations that shared significant fragments with the faulty computations but do not exhibit the fault. The latter content can be used to generate additional test cases for the “false positives” reporting in existing and future Static Code Analysis tools.

The Dataflow Element Sink and Source (when applicable) are *specifications* of the fault. On the other hand, CanonicalElements SinkSegment and SourceSegment are *rules* that describe unmitigated Sink and Source together with the context. At the same time, MitigatedSinkSegment and MitigatedSourceSergment elements represent canonical Sink and Source with mitigations, also with the context. CanonicalSegment are referred to by a CanonicalForm that provides the full context for canonical descriptions of both faulty and mitigated computations. The elements of context are introduced when needed, esp. for Source when there is a gap between the specification and the full context.

For example, an invariant specification may specify data flow as having a sink that is a dereference of a pointer. This content may be used to systematically collect evidence of such data flows in a given code. The assumed capabilities are

1) representing the code as a set of KDM facts;
2) identifying the specific facts that are instances of the invariant.

This example is simplified not to include capabilities to identify dataflows related to the identified location. From the certification perspective, a complete collection of facts is obtained by the capability #1 (for example by parsing source or binary code and representing the result as a set of standard KDM facts). The content that specifies the fault only focuses at the key KDM facts. The makes implicit assumptions and relies on the constraints defined in the KDM standard (for example, that there is a variable that declares a pointer, and that the statement representing a pointer dereference is part of some procedure that is called from some runtime entry point). Also, the concise description of the invariant relies on KDM to represent the multitude of situations where the pointer being dereferenced can be embedded into a more complex data structure, and the dereference may be also part of a more complex statement. Some (but not all) of these implicit assumptions are referred to as the "context" of the faulty computation.

The CanonicalElements define the dataflow related to the SFP fault in context by providing the necessary definitions and filling in the minimal required scopes so that the resulting KDM describes a minimal fully functional code sample. Further, the CanonicalElements provide some "hooks" for adding "code and data complexities" in a structured way so that increasingly more complex samples can be produced. For obvious reasons, it is not possible to enumerate all samples of a fault. Thus, the content of the SFP Catalog involves an interface to the external capabilities that can systematically add code and data complexities as needed. The name "canonical" emphasizes the fact that the content only represents certain select (i.e. "canonical" rather than "random") examples out of an infinite number of compliant dataflows.

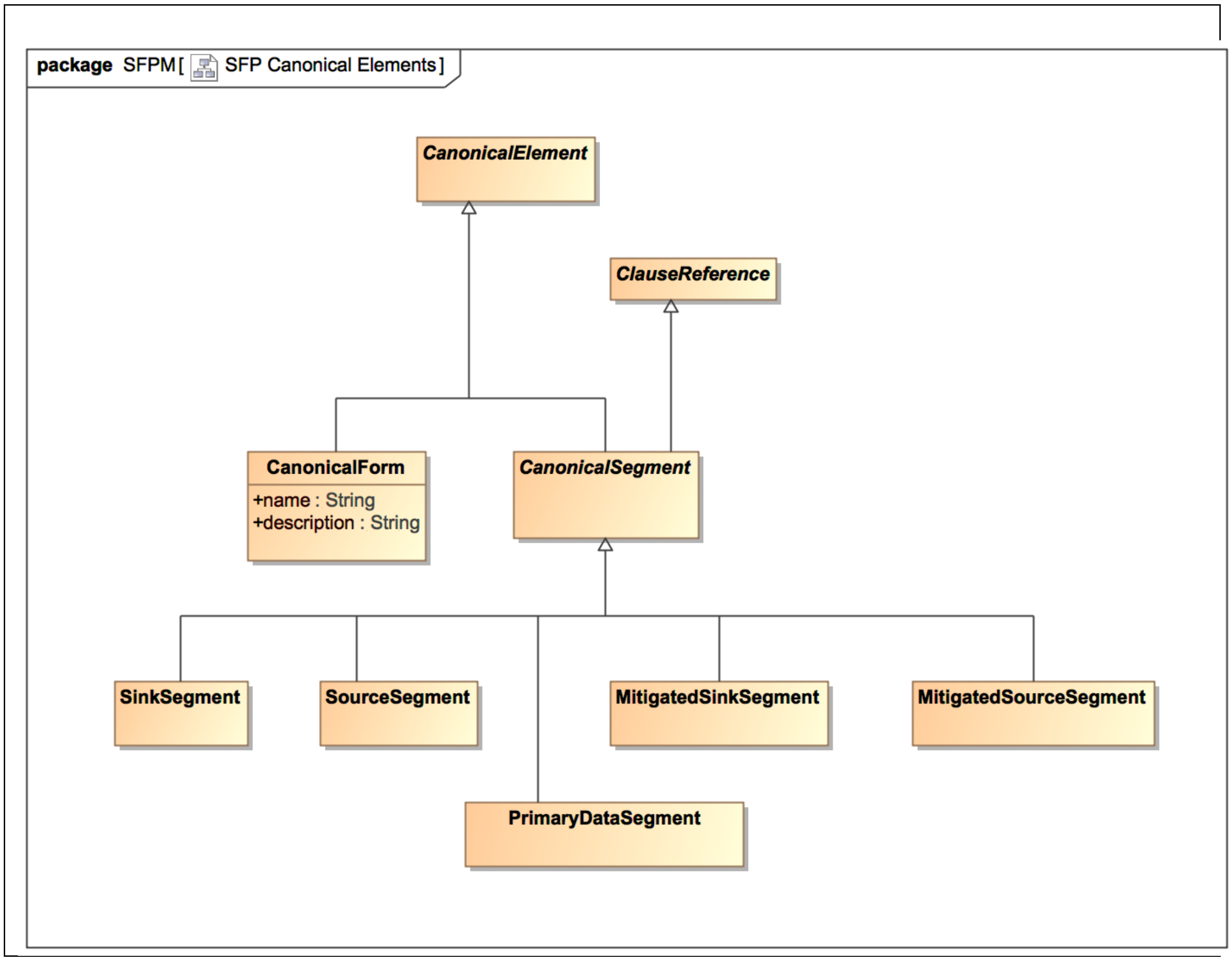

Figure 12. UML class diagram SFP Canonical Elements

8.3.3.1 CanonicalElement Class (abstract)

CanonicalElement is a parent class of the elements of SFP Catalog that provide context for the faulty computations captured as dataflow invariants.

Superclass

SemanticElement

8.3.3.2 CanonicalForm Class

Through its semantic definition, CanonicalForm defines a sequence of segments that completes the definition of fault into a full canonical representation with appropriate context. The CanonicalForm is the “blueprint” for plugging in other CanonicalSegments. CanonicalForm describes how various segments (specific to an SFP and referenced context elements) can be arranged into a coherent piece of source code which “implements” a fault in an appropriate context.

 The Canonical Form assumes few simple variation rules, such as that the “unmitigated” sample of the data flow can be obtained by plugging in Sink and Source segments, while several structurally different forms of “mitigated” dataflow can be obtained by plugging in 1) only Mitigated Sink instead of the Sink Segment; 2) only MitigatedSource instead of SourceSegment; 3) both Mitigated Sink and MitigatedSource.

CanonicalForm is closely aligned with the variation tree described in section 8.1.2.3.

A “segment” (a “KDM segment”) is a term consistently used in SFP to refer to a semantic formulation that represents one or more KDM ActionElement together with the corresponding Flows relationships is such a way that there is a single “entry” ActionElement and a single “exit” element. The “signature” of a segment includes two variables that reference these two elements. KDM segments are useful building blocks of content. SFP vocabularies built on top of KDM often define KDM segments as new “verbs”.

Specification of the CanonicalForm assumes the Flow relations between the segments of the semantic definition, connecting them in the order in which they occur in the conjunction. The intent is that the implementation capabilities may inject interleaving dataflows between the segments of the CanonicalForm.

Complexity “hooks” are introduced to the same end and use explicit tags that refer to the kinds of complexity that can be injected. The vocabulary of complexity “hooks” in not part of the SFP Metamodel, and shall be explained in the SFP catalog.

Superclass

CanonicalElement

Example 1. SFPM XMI

This example focuses on the main element of the CanonicalForm and omits the body of the semantic formulation. See Example 2 for the full content.

```
<canonical xmi:type="sfpm:CanonicalForm" xmi:id="cla40" name="CF1" >
    <definition>
       <meaning xmi:id="sem67" kind="SetProjection"
                  description="Definition of CF1" >
         <variable xmi:id="var35" range="nc4" name="S1"/>
         <variable xmi:id="var36" range="nc4" name="S2"/>
            <!—- body omitted -->
       </meaning>
    </definition>
</canonical>
```

Example 2. Readable SFP language

```
Canonical CF1
#
# CanonicalForm defines a sequence of segments that fully exemplifies a fault
# in an appropriate context
#
      Var S1: ActionElement [KDM]
      Var S2: ActionElement [KDM]
      Var DT :DataType [KDM]
      Var BK: TargetBufferKind [Platform Meta]
      Var TB: Buffer [Platform APIs]

      Var TBT : DataType [KDM]        ;; target buffer type
# TBTI can be a Pointer ItemUnit, an Array ItemUnit or a class MemberUnit
      Var TBTI : DataElement [KDM]    ;; target buffer item

      Var BPTI: DataElement [KDM]     ;; buffer pointer item
      Var BPCT: DataType [KDM]         ;; buffer pointer container type
      Var BPT: DataType [KDM]          ;; buffer pointer type
      Var BPBT: DataType [KDM]         ;; buffer pointer base type
      Var BP: DataElement [KDM]        ;; buffer pointer

      Var BufferLength: DataElement [KDM]
      Var BufferSize : IntegerValue [SBVR]
      Var Data: DataElement [KDM]
      Var DataLength : DataElement [KDM]
      Var DataSize: IntegerValue [SBVR]
      Var Index: DataElement [KDM]
      Var Offset: DataElement [KDM]

         Clause
            Var A2: ActionElement [KDM]
          Var A3: ActionElement [KDM]
            Var A4: ActionElement [KDM]
          Var A5: ActionElement [KDM]
          Var A6: ActionElement [KDM]
          Var A7: ActionElement [KDM]
          Var A8: ActionElement [KDM]
          Var A9: ActionElement [KDM]
          Var A10: ActionElement [KDM]
          Var A11: ActionElement [KDM]
            Var BE_1: DataElement [KDM]
            Var BE_2: DataElement [KDM]
            Var BE_3: DataElement [KDM]
            Var BP_1: DataElement [KDM]
          Var SA: ActionElement [KDM]
          Var SB: ActionElement [KDM]

        Clause [DataType ElementType] DT=DT

        Clause [Resource TargetBuffer] BK=BK, TB=TB
        Clause [DataType TargetBufferType] BK=BK, TBT=TBT, TBTI=TBTI, BPT=BPT,
                   DT=DT, BPBT=BPBT,  BufferSize=BufferSize

        Clause [PrimaryDataSegment] S1=S1, S2=A2, BPBT=BPBT, TBTI=TBTI,
                  BufferSize=BufferSize, BP=BP, BPT=BPT, BPTI=BPTI, BPCT=BPCT

        Clause [DataElement BufferLength] BufferLength=BufferLength,
                   BufferSize=BufferSize

        Clause [DataElement DataLengthGood] DataLength=DataLength,
DataSize=DataSize
        Clause [DataElement DefineData] DT=DT, Data=Data, DataSize=DataSize
        Clause [DataElement DefineIndex] BK=BK, Index=Index
```

```
            [container access :Hooks] A3, A4, BP, BE_1

        Clause [SourceSegment] S1=A4, S2=A5, BK=BK, TB=TB, TBT=TBT, BPT=BPT,
                  BPBT=BPBT, BP=BE_1, BufferSize=BufferSize

                  [complexity comment :Hooks] A6

            [container access :Hooks] A6, A7, BP_1, BE_2

        Clause [SinkSegment] S1=A7, S2=A8, BK=BK, TBT=TBT, BPT=BPT, BPTI=BPTI,
            TBTI=TBTI, BPBT=BPBT, DT=DT, BP=BE_2, Data=Data,
            DataLength=DataLength, Index=Index, BufferSize=BufferSize

            [container access :Hooks] A9, A10, BP_1, BE_3

        Clause [Operation Cleanup] S1=A10, S2=A11, BK=BK, BPT=BPT, BPBT=BPBT,
                        BP=BE_3

                  [complexity connect :Hooks] A2, A3
                  [complexity end :Hooks] A6, A8, SA, SB, BPT
                  [complexity connect :Hooks] SB, A9
                  [complexity path :Hooks] A5, SA, BP, BP_1, BPT, BPBT, BPCT
                  [complexity return :Hooks] A11

End Canonical
```

8.3.3.3 CanonicalSegment Class (abstract)

CanonicalSegment class represents a canonical version of a Dataflow element with full context. Further, Some CanonicalSegment represent the "mitigated" versions of the corresponding Dataflow element. Coordination of the various clauses is guided by the Property element defined as part of the clauses.

Superclass

CanonicalElement, ClauseReference

8.3.3.4 SinkSegment Class

SinkSegment class represents a canonical version of the Dataflow element SinkStatement with full context.

Superclass

CanonicalSegment

Example 1. SFPM XMI

This example focuses on the main element of the SinkSegment and omits the body of the semantic formulation. See Example 2 for the full content.

```
<canonical xmi:type="sfpm:SinkSegment" xmi:id="cla43" >
    <definition>
```

```
        <meaning xmi:id="sem104" kind="SetProjection"
                   description="Definition of segment SinkSegment" >
           <variable xmi:id="var92" range="nc4" name="S1"/>
           <variable xmi:id="var93" range="nc4" name="S2"/>
             <!-- body omitted -->
        </meaning>
    </definition>
</canonical>
```

Example 2. Readable SFP language

```
  Segment SinkSegment
#
# the version of Sink with context uses additional parameter BufferSize
# for use in mitigation;
# must be signature compatible with MitigatedSinkSegment
#
            Var S1 : ActionElement [KDM] ;;; segment Begin
            Var S2 : ActionElement [KDM] ;;; segment End

            Var BK: TargetBufferKind [Platform Meta]
            Var TBT: DataType [KDM]
            Var BPT: DataType [KDM]
            Var BPTI: DataElement [KDM]
# TBTI can be a Pointer ItemUnit, an Array ItemUnit or a class MemberUnit
            Var TBTI: DataElement [KDM]
# method unit optional
#           Var TBTM: ControlElement [KDM]
            Var BPBT: DataType [KDM]
            Var DT: DataType [KDM]
            Var BP: DataElement [KDM]
            Var Data: DataElement [KDM]
            Var DataLength : DataElement [KDM]
            Var Index: DataElement [KDM]
         Var BufferSize : IntegerValue [SBVR]

            Clause [SinkStatement] S1=S1, S2=S2, BK=BK, TBT=TBT, BPT=BPT,
                  BPTI=BPTI, TBTI=TBTI, BPBT=BPBT, DT=DT, BP=BP, Data=Data,
                   DataLength=DataLength, Index=Index

      End Segment
```

8.3.3.5 SourceSegment Class

SourceSegment class represents a canonical version of the Dataflow element SourceStatement with full context.

Superclass

CanonicalSegment

Example 1. SFPM XMI

This example focuses on the main element of the SourceSegment and omits the body of the semantic formulation. See Example 2 for the full content.

```
<canonical xmi:type="sfpm:SourceSegment" xmi:id="cla42" >
  <definition>
    <meaning xmi:id="sem96" kind="SetProjection"
            description="Definition of segment SourceSegment" >
      <variable xmi:id="var79" range="nc4" name="S1"/>
      <variable xmi:id="var80" range="nc4" name="S2"/>
            <!—- body omitted -->
       </meaning>
    </definition>
</canonical>
```

Example 2. Readable SFP language

```
Segment SourceSegment
      Var S1: ActionElement [KDM]
      Var S2: ActionElement [KDM]

      Var BK: TargetBufferKind [Platform Meta]
      Var TB: Buffer [Platform APIs]
      Var TBT : DataType [KDM]
      Var BPT: DataType [KDM]
      Var BPBT: DataType [KDM]
      Var BP: DataElement [KDM]
      Var BufferSize : IntegerValue [SBVR]

      Clause
            Var A2: ActionElement [KDM]
            Var A3: ActionElement [KDM]
            Var A4: ActionElement [KDM]
            Var A5: ActionElement [KDM]

            where
            Clause [Operation DefineTargetBuffer] S1=S1, S2=A2, BK=BK, TBT=TBT,
                   BPT=BPT, BPBT=BPBT, TB=TB, BP=BP, BufferSize=BufferSize
            Clause [Operation BindPointerToTargetBuffer] S1=A3, S2=A4,
                  BK=BK,BPT=BPT,BPBT=BPBT,BP=BP, TB=TB
            Clause [Operation ReleaseTargetBuffer] S1=A5, S2=S2, BK=BK, BPT=BPT,
                   BPBT=BPBT, BP=BP
            [ActionElement1 flows into ActionElement2 :KDM] A2, A3
            [ActionElement1 flows into ActionElement2 :KDM] A4, A5

End Segment
```

8.3.3.6 PrimaryDataSegment Class

PrimaryDataSegment class represents a canonical version of the Dataflow element PrimaryDataStatement with full context.

Superclass

CanonicalSegment

Example 1. SFPM XMI

This example focuses on the main element of the PrimaryDataSegment and omits the body of the semantic formulation. See Example 2 for the full content.

```
<canonical xmi:type="sfpm:PrimaryDataSegment" xmi:id="cla41" >
            <definition>
              <meaning xmi:id="sem90" kind="SetProjection"
                  description="Definition of segment PrimaryDataSegment" >
                <variable xmi:id="var70" range="nc4" name="S1"/>
                <variable xmi:id="var71" range="nc4" name="S2"/>
            <!—- body omitted -->
       </meaning>
    </definition>
</canonical>
```

Example 2. Readable SFP language

```
Segment PrimaryDataSegment
      Var S1: ActionElement [KDM]
      Var S2: ActionElement [KDM]

      Var BPBT: DataType [KDM]  ;;; target buffer base type (in)
      Var TBTI: DataElement [KDM]  ;;; target buffer type item (in)
      Var BufferSize: DataType [KDM] ;;   target buffer size (in)
      Var BP: DataElement [KDM]  ;;; buffer pointer (out)
      Var BPT: DataType [KDM]  ;;; buffer pointer type (out)
      Var BPTI: DataElement [KDM] ;;; buffer pointer item (out)
      Var BPCT: DataType [KDM]     ;;; buffer pointer container type (out)

      [meta] complexity.inline,no
      Clause [DataType BufferPointerType] BPT=BPT, BPBT=BPBT, BPTI=BPTI,
                  TBTI=TBTI
      [key data type pointer :Hooks] BPBT, BPT, BPTI, BufferSize
      Clause [DataElement BufferPointer] S1=S1, S2=S2, BP=BP, BPT=BPT,
            BPBT=BPBT, BPCT=BPCT
End Segment
```

8.3.3.7 MitigatedSinkSegment Class

MitigatedSinkSegment class represents a canonical mitigated version of the Dataflow element SinkStatement with full context. See additional descriptions in the CanonicalForm section and the introduction to section 8.3.

Superclass

CanonicalSegment

Example 1. SFPM XMI

This example focuses on the main element of the MitigatedSinkSegment and omits the body of the semantic formulation. See Example 2 for the full content.

```
<canonical xmi:type="sfpm:MitigatedSinkSegment" xmi:id="cla45" >
            <definition>
```

```
                <meaning xmi:id="sem117" kind="SetProjection"
                    description="Definition of segment MitigatedSinkSegment" >
                  <variable xmi:id="var124" range="nc4" name="S1"/>
                  <variable xmi:id="var125" range="nc4" name="S2"/>
              <!-- body omitted -->
         </meaning>
      </definition>
</canonical>
```

Example 2. Readable SFP language

```
Segment MitigatedSinkSegment
      Var S1 : ActionElement [KDM] ;;; segment Begin
      Var S2 : ActionElement [KDM] ;;; segment End

      Var BK: TargetBufferKind [Platform Meta]
      Var TBT: DataType [KDM]
      Var BPT: DataType [KDM]
      Var BPTI: DataElement [KDM]
# TBTI can be a Pointer ItemUnit, an Array ItemUnit or a class MemberUnit
      Var TBTI: DataElement [KDM]
      Var BPBT: DataType [KDM]
      Var DT: DataType [KDM]
      Var BP: DataElement [KDM]
      Var Data: DataElement [KDM]
      Var DataLength : DataElement [KDM]
      Var Index: DataElement [KDM]
      Var BufferSize : IntegerValue [SBVR]

      Disjunction
            Clause "null dereference"
                  Var A1 : ActionElement [KDM] ;;; segment Begin
                  Var A2 : ActionElement [KDM] ;;; segment End
                  where
                        [meta] isnull,yes
                        Clause [SinkStatement] S1=A1, S2=A2, BK=BK, TBT=TBT,
                               BPT=BPT, BPTI=BPTI, TBTI=TBTI, BPBT=BPBT,
                              DT=DT, BP=BP, Data=Data, DataLength=DataLength,
                               Index=Index
                        [simple Begin2 End2 mitigates null of DataType in segment
                               Begin1 End1 DataElement:KDM Patterns] S1, S2,
                                    BPT, A1, A2, BP

            Clause "other - use an alternative source"
                  Var TB: Buffer [Platform APIs]

                  Var A2: ActionElement [KDM]
                  Var A3: ActionElement [KDM]

                  where
                        [meta] isnull,no
                        [meta] complexity.inline,no

                         Clause [Operation DefineValidReference] S1=S1, S2=A2,
                               BK=BK, TBT=TBT, BPT=BPT, BPBT=BPBT, TB=TB,
                              BP=BP, BufferSize=BufferSize
                        Clause [SinkStatement] S1=A3, S2=S2, BK=BK, TBT=TBT,
                              BPT=BPT, BPTI=BPTI, TBTI=TBTI, BPBT=BPBT, DT=DT,
                               BP=BP, Data=Data, DataLength=DataLength,
                              Index=Index

                        [ActionElement1 flows into ActionElement2 :KDM] A2, A3

      End Disjunction
End Segment
```

8.3.3.8 MitigatedSourceSegment Class

MitigatedSourceSegment class represents a canonical mitigated version of the Dataflow element SourceStatement with full context. See additional descriptions in the CanonicalForm section and the introduction to section 8.3.

Superclass

CanonicalSegment

Example 1. SFPM XMI

This example focuses on the main element of the MitigatedSourceSegment and omits the body of the semantic formulation. See Example 2 for the full content.

```
<canonical xmi:type="sfpm:MitigatedSourceSegment" xmi:id="cla44" >
            <definition>
              <meaning xmi:id="sem106" kind="SetProjection"
                  description="Definition of segment MitigatedSourceSegment" >
                <variable xmi:id="var106" range="nc4" name="S1"/>
                <variable xmi:id="var107" range="nc4" name="S2"/>
            <!—- body omitted -->
       </meaning>
    </definition>
</canonical>
```

Example 2. Readable SFP language

```
Segment MitigatedSourceSegment
#
# generates a valid reference for each variant of the Parameter Incorrect Value
Kind
#
       Var S1: ActionElement [KDM]
         Var S2: ActionElement [KDM]

         Var BK: TargetBufferKind [Platform Meta]
         Var TB: Buffer [Platform APIs]
         Var TBT : DataType [KDM]
         Var BPT: DataType [KDM]
         Var BPBT: DataType [KDM]
         Var BP: DataElement [KDM]
         Var BufferSize : IntegerValue [SBVR]
```

```
            Clause
                  Var A1: ActionElement [KDM]
                  Var A2: ActionElement [KDM]
            Var A3: ActionElement [KDM]
                  Var A4: ActionElement [KDM]
            Var A5: ActionElement [KDM]
            Var A6: ActionElement [KDM]

            Var A7: ActionElement [KDM]
                  Var A8: ActionElement [KDM]
#           Var A9: ActionElement [KDM]
#           Var A10: ActionElement [KDM]
            Var ValidTB: DataElement [KDM] such that
                  [DataElement is a temporary variable of DataType :KDM]
                        ValidTB, BPT

         where
                   Clause [Operation DefineTargetBuffer] S1=A1, S2=A2,
                              BK=BK, TBT=TBT, BPT=BPT, BPBT=BPBT, TB=TB,
                              BP=BP, BufferSize=BufferSize
               Clause [Operation BindPointerToTargetBuffer] S1=A3, S2=A4,
                               BK=BK,BPT=BPT,BPBT=BPBT,BP=BP, TB=TB
               Clause [Operation ReleaseTargetBuffer] S1=A5, S2=A6,
                              BK=BK, BPT=BPT, BPBT=BPBT, BP=BP
               [ActionElement1 flows into ActionElement2 :KDM] A2, A3
               [ActionElement1 flows into ActionElement2 :KDM] A4, A5

               Clause [Operation DefineValidReference] S1=A7, S2=A8,
                        BK=BK, TBT=TBT, BPT=BPT, BPBT=BPBT, TB=ValidTB,
                        BP=BP, BufferSize=BufferSize
#              Clause [Operation BindMitigatedPointerToTargetBuffer]
                        S1=A9, S2=A10, BK=BK,BPT=BPT,BPBT=BPBT,BP=BP, TB=TB
#              [ActionElement1 flows into ActionElement2 :KDM] A8, A9

                  [segment Begin3 End3 mitigates segments Begin1 End1 and
                        Begin2 End2:KDM Patterns] S1, S2, A1, A6, A7, A8
    End Segment
```

8.3.4 SFP Context Elements Class Diagram

This section describes the Context Elements of the SFP Catalog. As ContextElement represent significant referenced clauses that are used mainly by the CanonicalElement they determine the common characteristics of an SFP and constitute an important part of the overall SFP content. Based on the shared ContextElement, SFP can be systematically grouped into clusters, and the nature of the relationships between different SFPs can be formally described.

SFP ContextElement are represent the conceptual level of the SFP description in contrast to the technical level represented by KDM vocabulary and KDM patterns. SFP ContextElement focus at the essential Resource, Operations, DataTypes and DataElements, as well as APIs and Decision involved in SFPs. By adding few more abstractions, SFP Catalog extends the standard vocabularies and accumulates more useful content towards the advanced analytics of the software weaknesses.

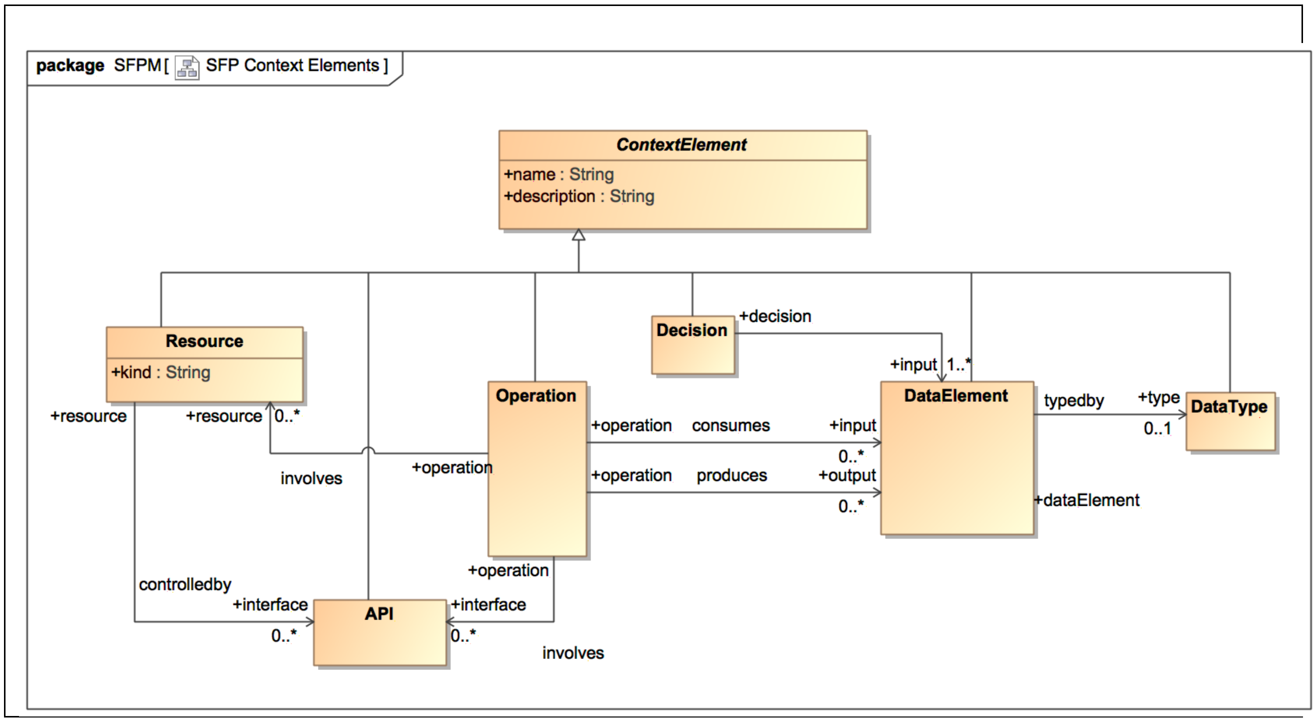

Figure 13. UML class diagram SFP Context Elements

8.3.4.1 ContextElement Class (abstract)

ContextElement class represents the common parent for context elements.

Superclass

SemanticElement

Attributes

name:String[1]	Name of the context elements
description: String[1]	Informal description of the context element

8.3.4.2 Resource Class

Resource class represents a resource provided by the operating system or by one of the frameworks. Resource can also be implemented by the software under assessment. Several weaknesses are directly related to manipulations of resources in non-secure ways. Describing these situations in terms of the programming constructs fails to communicate the essence of the related software faults, as from the programming language perspective manipulations of resources looks like API calls. KDM specification already introduces a Resource Layer to provide a more meaningful representation of common resources. SFP Resource makes this framework more visible as part of the content of the SFP Catalog. As an SFP ContextElement DataType can serve as a variation point for synthesis of test cases with appropriate metadata, as well as a characteristic of an SFP while analyzing relationships between multiple SFP.

Superclass

ContextElement

Attributes

kind:String[1]	Kind of the resource (defined in the KDM specification)

Associations

interface:API[0..*]	Set of API of the resource

Example 1. SFPM XMI

```
<element xmi:type="sfpm:Resource" xmi:id="shared2"  name="TargetBuffer">
  <definition>
    <meaning xmi:id="sem188" kind="SetProjection"
            description="Definition of Resource TargetBuffer" >
      <variable xmi:id="var163" range="nc5" name="BK"/>
      <variable xmi:id="var164" range="nc6" name="TB"/>
      <operand xmi:id="sem189" identificator="" kind="Conjunction"
                  description="">
        <operand xmi:id="sem190" verb="vc27" identificator=""
                  kind="AtomicFormulation" description="">
          <binding rolename="TargetBufferKind" target="var163"/>
          <binding rolename="Buffer" target="var164"/>
        </operand>
        <operand xmi:id="sem191" verb="vc4" identificator=""
                  kind="AtomicFormulation" description="">
          <binding rolename="Thing1" target="ic22"/>
          <binding rolename="Thing2" target="ic5"/>
        </operand>
      </operand>
    </meaning>
  </definition>
</element>
```

Example 2. Readable SFP language

```
Resource TargetBuffer
      Var BK: TargetBufferKind [Platform Meta]
      Var TB: Buffer [Platform APIs]

      [TargetBufferKind represents Buffer : Platform APIs] BK, TB
      [meta] ooapi,no
End Resource
```

8.3.4.3 Operation Class

Operation class represents a logical operation on a resource. Usually an operation changes the state of the resource. Operations are often implemented by the operating systems or by one of the software frameworks. Operation can also be defined as part of the software. As an SFP ContextElement Operation can serve as a variation point for synthesis of test cases with appropriate metadata, as well as a characteristic of an SFP while analyzing relationships between multiple SFP. From the formalization perspective, an Operation is usually a KDM *segment*.

Superclass

ContextElement

Associations

resource:Resource[0..*]	Resource manipulated by the operation, if any
input: DataElement[0..*]	Input to the operation, if any
output: DataElement[0..*]	Output of the operation, if any
interface: API[0..*]	API of this operation

Example 1. SFPM XMI

```
<element xmi:type="sfpm:Operation" xmi:id="shared13" name="ReleaseTargetBuffer"
resource="shared2" >
  <definition>
    <meaning xmi:id="sem707" kind="SetProjection"
                  description="Definition of operation ReleaseTargetBuffer" >
       <variable xmi:id="var255" range="nc4" name="S1"/>
       <variable xmi:id="var256" range="nc4" name="S2"/>
            <!-- body omitted -->
       </meaning>
    </definition>
</element>
```

Example 2. Readable SFP language

```
Operation ReleaseTargetBuffer
      involves TargetBuffer

      Var S1: ActionElement [KDM]
      Var S2: ActionElement [KDM]
      Var BK: TargetBufferKind [Platform Meta]
      Var BPT: DataType [KDM]
      Var BPBT: DataType [KDM]
      Var BP: DataElement [KDM]

      Disjunction
            Clause "buffer is available"
                  # no release
                  [meta] release, no
                  [ActionElement is nop :KDM] S1
                  [Thing1 is Thing2 :SBVR] S2, S1

            Clause "buffer is in released state"
                  # explicit release;

                  [meta] release, yes
                  [ simple Begin End releases Buffer of DataType:KDM Patterns]
                              S1, S2,  BP, BPT

End Operation
```

8.3.4.4 DataType Class

DataType class represents a data type in the software under assessment. A DataType class is a more powerful construct in comparison to a KDM fact, since it allows disjunction of KDM types, complex KDM types, that involve multiple facts, as well as combinations of KDM statements and properties. As an SFP ContextElement DataType can serve as a variation point for synthesis of test cases with appropriate metadata, as well as a characteristic of an SFP while analyzing relationships between multiple SFP.

Superclass

ContextElement

Example 1. SFPM XMI

```
<element xmi:type="sfpm:DataType" xmi:id="shared1"  name="ElementType">
  <definition>
    <meaning xmi:id="sem131" kind="SetProjection"
                  description="Definition of DataType ElementType" >
      <variable xmi:id="var143" range="nc1" name="DT"/>
      <operand xmi:id="sem132" identificator="" kind="ExistentialQuantification"
                  description="">
        <variable xmi:id="var144" range="nc8" name="T">
          <restriction xmi:id="sem133" verb="vc2" identificator=""
                  kind="AtomicFormulation" description="">
            <binding rolename="KDMEntity" target="var144"/>
            <binding rolename="Name" target="ic11"/>
          </restriction>
        </variable>
        <operand xmi:id="sem134" identificator="" kind="Conjunction"
                  description="">
          <operand xmi:id="sem135" verb="vc4" identificator=""
                  kind="AtomicFormulation" description="">
            <binding rolename="Thing1" target="ic12"/>
            <binding rolename="Thing2" target="ic13"/>
          </operand>
          <operand xmi:id="sem136" verb="vc4" identificator=""
                  kind="AtomicFormulation" description="">
            <binding rolename="Thing1" target="var143"/>
            <binding rolename="Thing2" target="var144"/>
          </operand>
        </operand>
      </operand>
    </meaning>
  </definition>
```

```
</element>
```

Example 2. Readable SFP language

```
DataType ElementType

       Var DT : DataType [KDM]
       Clause
             Var T : CharType [KDM] such that
                   [KDMEntity has Name :KDM] T, {"char": Strings}
             where
                  [meta] complexity.datatype,char
                   [Thing1 is Thing2 :SBVR] DT, T

End DataType
```

8.3.4.5 DataElement Class

DataElement class represents a data element in the software under assessment. A DataElement class is a more powerful construct in comparison to a KDM fact, since it allows disjunction of KDM facts, complex KDM facts, as well as combinations of KDM statements and properties. As an SFP ContextElement DataElement can serve as a variation point for synthesis of test cases with appropriate metadata, as well as a characteristic of an SFP while analyzing relationships between multiple SFP.

Superclass

ContextElement

Associations

type:DataType[0..*]	Type of the data element, if available

Example 1. SFPM XMI

```
<element xmi:type="sfpm:DataElement" xmi:id="shared7"  name="BufferLength">
      <definition>
        <meaning xmi:id="sem137" kind="SetProjection"
                  description="Definition of DataElement BufferLength" >
          <variable xmi:id="var145" range="nc2" name="BufferLength"/>
          <variable xmi:id="var146" range="nc7" name="BufferSize"/>
          <operand xmi:id="sem138" identificator="" kind="Conjunction"
                  description="">
            <operand xmi:id="sem139" verb="vc22" identificator=""
                  kind="AtomicFormulation" description="">
              <binding rolename="DataElement" target="var145"/>
              <binding rolename="Datatype" target="ic14"/>
              <binding rolename="Name" target="ic15"/>
            </operand>
            <operand xmi:id="sem140" verb="vc4" identificator=""
                        kind="AtomicFormulation" description="">
              <binding rolename="Thing1" target="var146"/>
              <binding rolename="Thing2" target="ic15"/>
            </operand>
          </operand>
        </meaning>
      </definition>
```

```
    </element>
```

Example 2. Readable SFP language

```
DataElement BufferLength

        Var BufferLength: DataElement [KDM]
        Var BufferSize : IntegerValue [SBVR]

        [ DataElement is a constant of Datatype with Name :KDM Patterns]
                   BufferLength, defaultInt, {"10":SBVR}
        [Thing1 is Thing2 :SBVR] BufferSize, {"10":SBVR}

End DataElement
```

8.3.4.6 API Class

API class represents an external function provided by the operating system or a software library. An API class is a more powerful construct in comparison to a KDM fact, since it allows disjunction of KDM facts, complex KDM facts, as well as combinations of KDM statements and properties. API provides more visibility to certain external functions as part of the SFP Content. As an SFP ContextElement API can serve as a variation point for synthesis of test cases with appropriate metadata, as well as a characteristic of an SFP while analyzing relationships between multiple SFP.

Superclass

ContextElement

8.3.4.7 Decision Class

Decision class represents one or more statements in the software under assessment that implement a decision. While such content can be represented as KDM, the use of a special ContextElement is warranted for important decisions in the code. A Decision class is a more powerful construct in comparison to a KDM fact, since it allows complex KDM facts, as well as combinations of KDM statements and properties. As an SFP ContextElement Decision can serve as a variation point for synthesis of test cases with appropriate metadata, as well as a characteristic of an SFP while analyzing relationships between multiple SFP.

Superclass

ContextElement

Associations

input:DataElement[1..*]	Input into the decision, if any

8.4 Semantic Formalization Apparatus

This section describes the set of elements that provide formal definitions to the SFP Formalization elements. The formalization apparatus defined in this section is aligned with existing ISO and OMG standards.

The formalization approach of the SFP Catalog used the ISO/OMG Knowledge Discovery Metamodel (KDM) as the foundation of the discourse related to software faults. Statements in KDM vocabulary represent basic semantic fragments. A basic KDM fragment is interpreted as a set of facts. More complex logical statements are made on top of the KDM fragments using the semantic formulations language defined in the ISO/OMG Semantics of Business Vocabularies and Rules (SBVR). Basic KDM fragments are the atomic propositions. There are two kinds of semantic formulations. The first kind, logical formulation, structures propositions, both simple and complex. Specializations of that kind are given for various logical operations, quantifications, atomic formulations based on verb concepts and other formulations for special purposes such as objectifications and nominalizations.

The second kind of semantic formulation is projection. It structures intensions as sets of things that satisfy constraints. Projections formulate definitions, aggregations, and questions.

Semantic Formulations allow building complex logical propositions in the well-understood formalism of propositional logic.

For the purposes of the SFP Catalog, SBVR provides a means for describing the structure of the meaning of software faults expressed in the natural language. Semantic formulations are not expressions or statements. They are structures that make up meaning. Using SBVR, the meaning of a definition or statement is communicated as facts about the semantic formulation of the meaning, not as a restatement of the meaning in a formal language.

8.4.1 Semantic Elements Class Diagram

This section describes the semantic elements of the SFP Catalog.

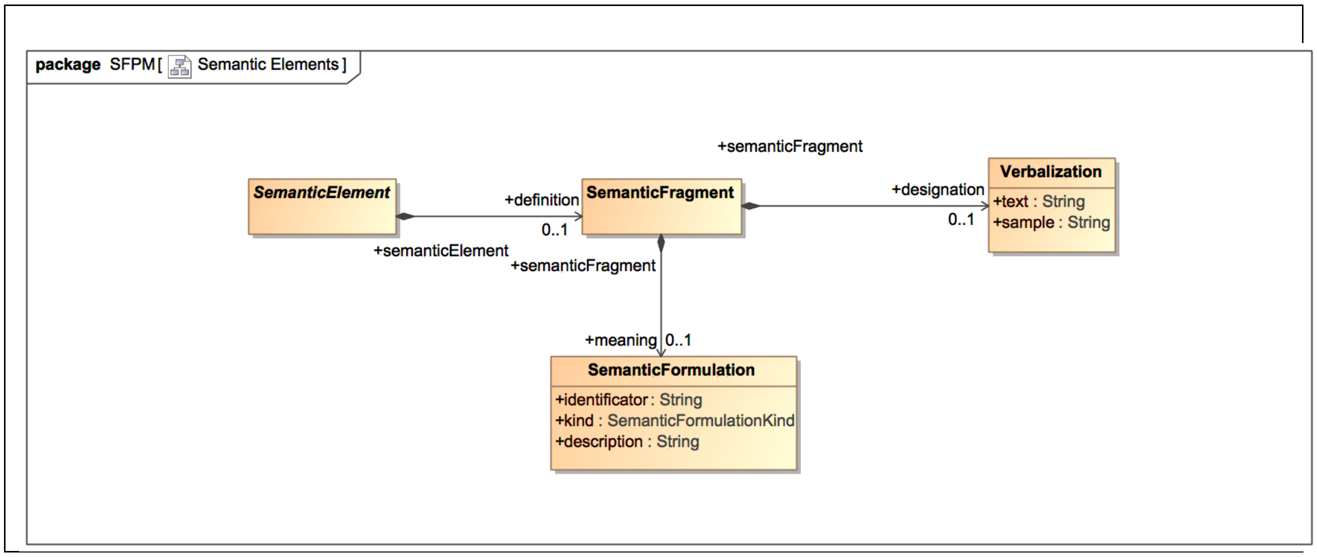

Figure 14. UML class diagram Semantic Elements

8.4.1.1 SemanticElement Class (abstract)

SemanticElement is the parent class for all elements of SFP Catalog that have a formal definition.

Superclass

Associations

definition:SemanticFragment[0..1]	Formal definition of a semantic element

8.4.1.2 SemanticFragment Class

SemanticFragment class represents the formal meaning of the element together with the designated structure text that describes the element. The top SemanticFormulation is typically a SetProjection that introduces zero or more variables which are considered the signature of the semantic element; any references to the semantic element shall match the signature. For a SFP Condition the top element is usually a quantification.

Superclass

Associations

designation:Verbalization[0..1]	Designated structured text describing the element
meaning: SemanticFormulation[0..1]	The formally defined meaning of the element

8.4.1.3 Verbalization Class

Verbalization class represents designated text that represent a semantic element in addition to its formal definition. For example, the verbalization can be provided as a structured English text according to the rules of SBVR.

Superclass

Attributes

text:String[1]	Designation of the element, for example Structured English text
sample: String[1]	Sample of the element, for example a fragment in selected programming language

8.4.2 Statements Class Diagram

This section describes the semantic formulations of the SFP Catalog. Semantic formulations provide conceptual structure of meaning [SBVR]. In SFPM semantic formulations are represented by a single class SemanticFormulation with a property kind of type SemanticFormulationKind that determines the associations and the meaning of the SemanticFormulation. The constraints of individual 'semantic formulation' kinds explain what meaning is formulated. A meaning is directly formulated only for a closed semantic formulation. In the case of variables being free within a semantic formulation, a meaning is formulated with respect there being exactly one referent thing given for each free variable.

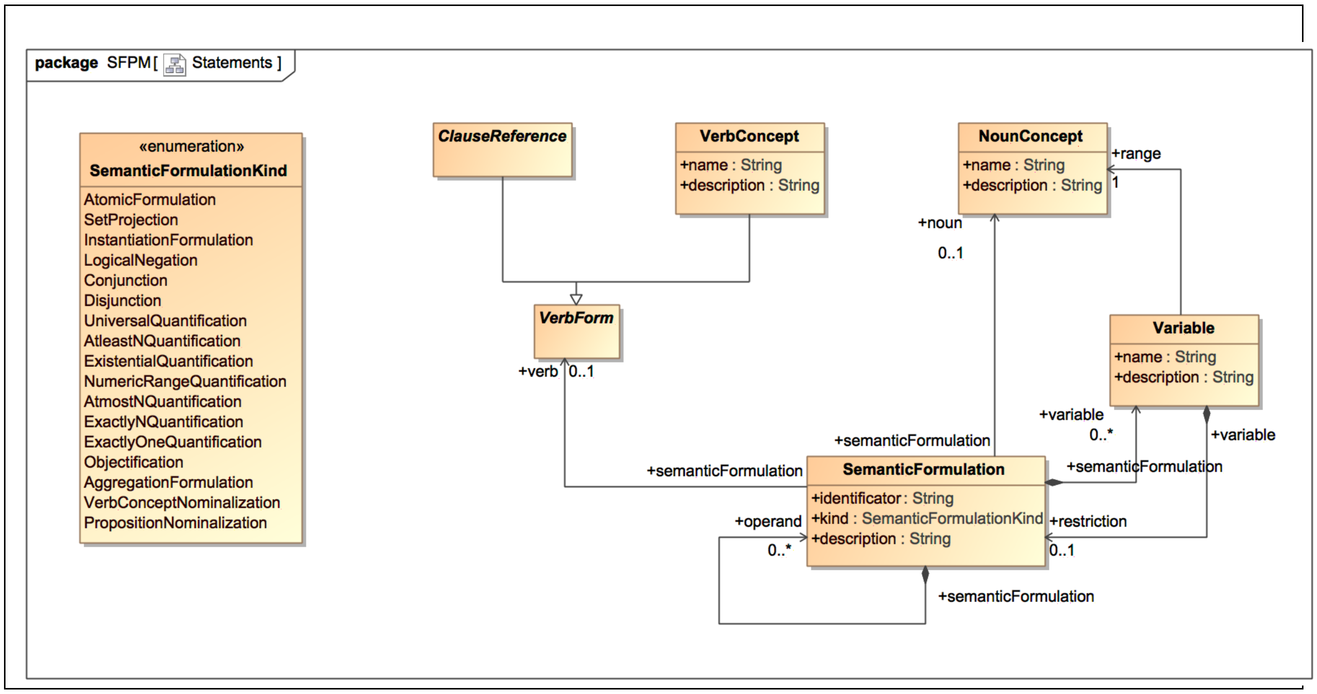

Figure 15. UML class diagram Statements

8.4.2.1 SemanticFormulation Class

SemanticFormulation class represents structure of meaning. Property kind of type SemanticFormulationKind determines the associations and the meaning of the SemanticFormulation element. The constraints of individual 'semantic formulation' kinds explain what meaning is formulated.

Superclass

Attributes

identificator:String[1]	Unique identifier of the element
kind: SemanticFormulationKind[1]	Literal that defines the kind of the Semantic Formulation element and constrains its associations

Associations

verb:VerbForm[0..1]	Verb used in some semantic formulation as determined by the SemanticFormulationKind
operand:SemanticFormulation[0..*]	Owned operand used in some semantic formulation as determined by the SemanticFormulationKind
noun:nounConcept[0..1]	Noun used in some semantic formulation as determined by the SemanticFormulationKind
variable:Variable[0..*]	Owned variable introduced by some semantic formulation as determined by the SemanticFormulationKind
description:String[0..1]	Description of the element
binding:RoleBinding[0..*]	Owned role bindings used in some semantic formulation as determined by the SemanticFormulationKind

Constraints

1. Each SemanticFormulation element shall have a set of associations determined by its kind as follows
   a. If Kind=AtomicFormulation then the SemanticFormulation element shall have exactly one *verb* and zero or more *binding* elements where each RoleBinding corresponds to a free variable of the VerbForm. The *rolename* property of the RoleBinding corresponds to the name of the role in the VerbForm.
      i. The AtomicFormulation formulates the meaning: there is an actuality that involves in each role of the verb concept the thing to which the bindable target of the corresponding role binding refers [SBVR].
   b. If Kind=SetProjection then the SemanticFormulation element shall have at most one *operand* (the constraint of the projection) and one or more *variable* elements. The SemanticFormulation is a Projection. The constraint of the projection shall not be a Projection.
      i. Projection introduces one or more variables corresponding to involvements in actualities. If the projection is constrained by a logical formulation, then for each combination of variables, one referent for each variable, the actuality is that the meaning of the constraining formulation is true. If the projection has no constraining formulation, then for each combination of variables, one referent for each variable, the actuality is that the referents exist [SBVR].
      ii. A Projection can be opened or closed. An opened projection refers to variables that are introduced outside of the projection. A closed projection refers only to the variables introduced by the projection.

    iii. Projection is used in ProjectingFormulation and as the element of meaning in SemanticFragment elements.

c. If Kind=InstantiationFormulation then the SemanticFormulation element shall have exactly one noun element and exactly one *binding* element where the *rolename* property of the RoleBinding is shall be ignored.

    i. InstantiationFormulation formulates the meaning: the thing to which the bindable target refers is an instance of the concept [SBVR]

d. If Kind=LogicalNegation then the SemanticFormulation element shall have exactly one *operand* element.

    i. LogicalNegation formulates that the meaning: the logical operand is false [SBVR]

e. If Kind=Conjunction or Kind=Disjunction then the SemanticFormulation element shall have two or more *operand* elements.

    i. Conjunction formulates that the meaning: each of its logical operands is true [SBVR]

    ii. Disjunction formulates that the meaning: at least one of its logical operands is true [SBVR]

f. If Kind=UniversalQuantification then the SemanticFormulation element shall exactly one *operand* element (the scope formulation) and exactly one *variable* element.

    i. UniversalQuantification formulated the meaning: for each referent of the variable introduced by the quantification the meaning formulated by the logical formulation for the referent is true [SBVR]

g. If Kind=AtLeastNQuantification or Kind=ExistentialQuantification or Kind=AtmostNQuantification or Kind=AtmostNQuantification or Kind=ExactlyNQuantification or Kind=ExactlyOneQuantification then the SemanticFormulation element shall have exactly one *operand* element and exactly one *variable* element and exactly one binding element where the *rolename* shall be ignored and the *target* is an individual concept representing a non-negative number. The SemanticFormulation is a Quantification.

    i. Quantification formulates the meaning: a bounded number of referents of the variable exist and satisfy a scope formulation [SBVR]

h. If Kind=NumericRangeQuantification then the SemanticFormulation element shall have exactly one *operand* element and exactly one *variable* element and exactly two binding elements where the *rolename* of the first RoleBinding is a string "min" and the rolename of the second RoleBinding is a string "max" and the *target* of either RoleBinding an individual concept representing a non-negative number.

    i. NumericRangeQuantification formulates the meaning: the number of referents of the variable introduced by the quantification that exist and that satisfy a scope formulation, is not less then the minimum cardinality and is not greater then the maximum cardinality [SBVR]

i. If Kind=Objectification then the SemanticFormulation element shall have exactly one *operand* element (the considered logical formulation) and exactly one *binding* element where the *rolename* property of the RoleBinding is an empty string. The considered formulation shall not be a Projection.

   i. Objectification formulates the meaning: the thing to which the bindable target refers is a state of affairs to which the meaning of the considered logical formulation corresponds [SBVR]

j. If Kind=AggregationFormulation or Kind=VerbConceptNominalization then the SemanticFormulation element shall have exactly one *operand* element (the considered projection) and exactly one *binding* element where the *rolename* property of the RoleBinding shall be ignored. The *operand* element shall be a Projection.

   i. AggregationFormulation formulates the meaning: the thing to which the bindable target bound to the projecting formulation refers is the result of the projection of the projecting formulation [SBVR]. The aggregation formulation is used primarily to associate a variable with a set of things, involvements, or actualities that satisfy some condition.

   ii. VerbConceptNominalization formulates the meaning: the thing to which the bindable target bound to the projecting formulation refers is a verb concept that is defined by the projection of the projecting formulation [SBVR]. A verb concept nominalization formulates the (anonymous) verb concept defined by a projection. In most uses of verb concept nominalizations, the bindable target is a unitary variable, and the effect is to define the variable to refer to the anonymous verb concept defined by the projection. It is the only referent for which the verb concept nominalization will hold.

k. If Kind=PropositionNominalization then the SemanticFormulation element shall have exactly one *operand* element (the considered logical formulation) and exactly one *binding* element where the *rolename* property of the RoleBinding is an empty string. The considered logical formulation shall not be a Projection.

   i. PropositionNominalization formulates the meaning: the thing to which the bindable target refers is the proposition that is formulated by the considered logical formulation [SBVR]

Example 1. SFPM XMI

```
<indicator xmi:type="sfpm:Indicator" xmi:id="cla4" name="ordinary pointer
dereference read">
  <definition>
     <meaning xmi:id="sem1388" kind="SetProjection"
                 description="Definition of indicator ordinary pointer
                 dereference read" >
        <variable xmi:id="var514" range="nc4" name="S1"/>
        <variable xmi:id="var515" range="nc4" name="S2"/>
        <variable xmi:id="var516" range="nc2" name="BP"/>
        <variable xmi:id="var517" range="nc2" name="BPTI"/>
        <variable xmi:id="var518" range="nc2" name="Data"/>
        <operand xmi:id="sem1389" identificator="ordinary pointer dereference
                 read" kind="Conjunction" description="">
```

```
          <operand xmi:id="sem1395" verb="vc109" identificator=""
                  kind="AtomicFormulation" description="">
              <binding rolename="ActionElement" target="var514"/>
            </operand>
          <operand xmi:id="sem1396" verb="vc9" identificator=""
                  kind="AtomicFormulation" description="">
              <binding rolename="ActionElement" target="var514"/>
              <binding rolename="DataElement" target="var516"/>
          </operand>
          <operand xmi:id="sem1397" verb="vc6" identificator=""
                  kind="AtomicFormulation" description="">
              <binding rolename="ActionElement" target="var514"/>
              <binding rolename="DataElement" target="var517"/>
          </operand>
          <operand xmi:id="sem1398" verb="vc7" identificator=""
                  kind="AtomicFormulation" description="">
              <binding rolename="ActionElement" target="var514"/>
              <binding rolename="DataElement" target="var518"/>
          </operand>
          <operand xmi:id="sem1399" verb="vc4" identificator=""
                   kind="AtomicFormulation" description="">
              <binding rolename="Thing1" target="var515"/>
              <binding rolename="Thing2" target="var514"/>
          </operand>
        </operand>
      </meaning>
    </definition>
</indicator>
```

Example 2. Readable SFP language

```
Indicator "ordinary pointer dereference read"

      Var S1 : ActionElement [KDM] ;;; segment Begin
      Var S2 : ActionElement [KDM] ;;; segment End
      Var BP: DataElement [KDM]
      Var BPTI: DataElement [KDM]
      Var Data: DataElement [KDM]

      Clause "ordinary pointer dereference read"
        # data=*p;
                    [ActionElement is ptrselect :KDM] S1
                    [ActionElement addresses DataElement :KDM] S1,BP
                    [ActionElement reads DataElement :KDM] S1, BPTI
                    [ActionElement writes DataElement :KDM] S1,Data
                    [Thing1 is Thing2 :SBVR] S2, S1

End Indicator
```

8.4.2.2 SemanticFormulationKind Enumeration

Enumeration that determines the structure and meaning of a SemanticFormulation element.

Literals

AtomicFormulation

SetProjection

InstantiationFormulation

LogicalNegation

Conjunction

Disjunction

UniversalQuantification

AtleastNQuantification

ExistentialQuantification

NumericRangeQuantification

AtmostNQuantification

ExactlyOneQuantification

Objectification

AggregationFormulation

PropositionNominalization

8.4.2.3 ClauseReference Class (abstract)

ClauseReference class represents the proposition based on a formally defined "clause" instead of a "VerbConcept" from one of the referenced vocabularies. A "clause" is a proposition that is part of one of the formally defined elements of the SFP Catalog, such as an Indicator, or one of the DataflowElement, or a ContextElement. Referenced clauses can be used in SemanticFormulation in the same way as VerbConcept. Note, that a VerbConcept can be also formally described. The ability to directly reference a clause allows preserving its primary role in the SFP Catalog.

Superclass

VerbForm

8.4.2.4 VerbForm Class (abstract)

VerbForm is either a VerbConcept or a ClauseReference. A VerbForm is the basis of propositions as defined in section 8.4.2.1.

8.4.2.5 Variable Class

Variable class represents logical variables introduced by certain semantic formulations. A variable is reference to an element of a set, whose referent may vary or is unknown [SBVR]. The set of referents of a variable is defined by the two verb concepts 'variable ranges over concept' and 'logical formulation restricts variable'. The set is limited to instances of the concept. If the variable is restricted by a logical formulation, the set is further limited to those things for which the meaning formulated by that logical formulation is true when the thing is substituted for each occurrence of the variable in the formulation.

Superclass

BindableTarget

Associations

name:String[1]	Name of the variable
description: String[1]	Description of the variable

Associations

range:NounConcept[1]	Range of the variable
restriction:SemanticFormulation[0..1]	Restriction on the set of instances

Example 1. SFPM XMI

```
<indicator xmi:type="sfpm:Indicator" xmi:id="cla6" name="struct member read">
  <definition>
    <meaning xmi:id="sem1411" kind="SetProjection" description="Definition of
indicator struct member read" >
      <variable xmi:id="var525" range="nc4" name="S1"/>
      <variable xmi:id="var526" range="nc4" name="S2"/>
      <variable xmi:id="var527" range="nc2" name="BP"/>
      <variable xmi:id="var528" range="nc1" name="BPBT"/>
      <variable xmi:id="var529" range="nc2" name="BPTI"/>
      <variable xmi:id="var530" range="nc2" name="TBTI"/>
      <variable xmi:id="var531" range="nc2" name="Data"/>
      <operand xmi:id="sem1412" identificator="struct member read"
            kind="ExistentialQuantification" description="">
        <variable xmi:id="var532" range="nc2" name="Tmp">
          <restriction xmi:id="sem1413" verb="vc19" identificator=""
                kind="AtomicFormulation" description="">
             <binding rolename="DataElement" target="var532"/>
             <binding rolename="DataType" target="var528"/>
          </restriction>
        </variable>
        <operand xmi:id="sem1414" identificator="" kind="Conjunction"
                description="">
           <operand xmi:id="sem1421" verb="vc109" identificator=""
                kind="AtomicFormulation" description="">
              <binding rolename="ActionElement" target="var525"/>
           </operand>
```

```
            <operand xmi:id="sem1422" verb="vc9" identificator=""
                   kind="AtomicFormulation" description="">
               <binding rolename="ActionElement" target="var525"/>
               <binding rolename="DataElement" target="var527"/>
            </operand>
            <operand xmi:id="sem1423" verb="vc6" identificator=""
                    kind="AtomicFormulation" description="">
               <binding rolename="ActionElement" target="var525"/>
               <binding rolename="DataElement" target="var529"/>
            </operand>
            <operand xmi:id="sem1424" verb="vc7" identificator=""
                   kind="AtomicFormulation" description="">
               <binding rolename="ActionElement" target="var525"/>
               <binding rolename="DataElement" target="var532"/>
            </operand>
            <operand xmi:id="sem1425" verb="vc18" identificator=""
                    kind="AtomicFormulation" description="">
               <binding rolename="ActionElement1" target="var525"/>
               <binding rolename="ActionElement2" target="var526"/>
            </operand>
            <operand xmi:id="sem1426" verb="vc111" identificator=""
                    kind="AtomicFormulation" description="">
               <binding rolename="ActionElement" target="var526"/>
            </operand>
            <operand xmi:id="sem1427" verb="vc9" identificator=""
                   kind="AtomicFormulation" description="">
               <binding rolename="ActionElement" target="var526"/>
               <binding rolename="DataElement" target="var532"/>
            </operand>
            <operand xmi:id="sem1428" verb="vc6" identificator=""
                   kind="AtomicFormulation" description="">
               <binding rolename="ActionElement" target="var526"/>
               <binding rolename="DataElement" target="var530"/>
            </operand>
            <operand xmi:id="sem1429" verb="vc7" identificator=""
                   kind="AtomicFormulation" description="">
               <binding rolename="ActionElement" target="var526"/>
               <binding rolename="DataElement" target="var531"/>
            </operand>
          </operand>
        </operand>
      </meaning>
    </definition>
</indicator>
```

Example 2. Readable SFP language

```
Indicator "struct member read"
  Var S1 : ActionElement [KDM] ;;; segment Begin
  Var S2 : ActionElement [KDM] ;;; segment End
  Var BP: DataElement [KDM]
  Var BPBT: DataType [KDM]
  Var BPTI: DataElement [KDM]
  Var TBTI: DataElement [KDM]
  Var Data: DataElement [KDM]

  Clause  "struct member read"
```

```
   # assuming  struct sx { char a; } *bpt; bpt p; data=p->a;
   Var Tmp : DataElement [KDM] such that
    [DataElement is a temporary variable of DataType :KDM] Tmp, BPBT
    where
      [ActionElement is ptrselect :KDM] S1
      [ActionElement addresses DataElement :KDM] S1,BP
      [ActionElement reads DataElement :KDM] S1, BPTI
      [ActionElement writes DataElement :KDM] S1,Tmp
      [ActionElement1 flows into ActionElement2 :KDM] S1,S2
      [ActionElement is fieldselect :KDM] S2
      [ActionElement addresses DataElement :KDM] S2,Tmp
      [ActionElement reads DataElement:KDM] S2, TBTI
      [ActionElement writes DataElement :KDM] S2,Data
End Indicator
```

8.4.3 Variable Bindings Class Diagram

This section describes the variable bindings used by semantic formulations of the SFP Catalog.

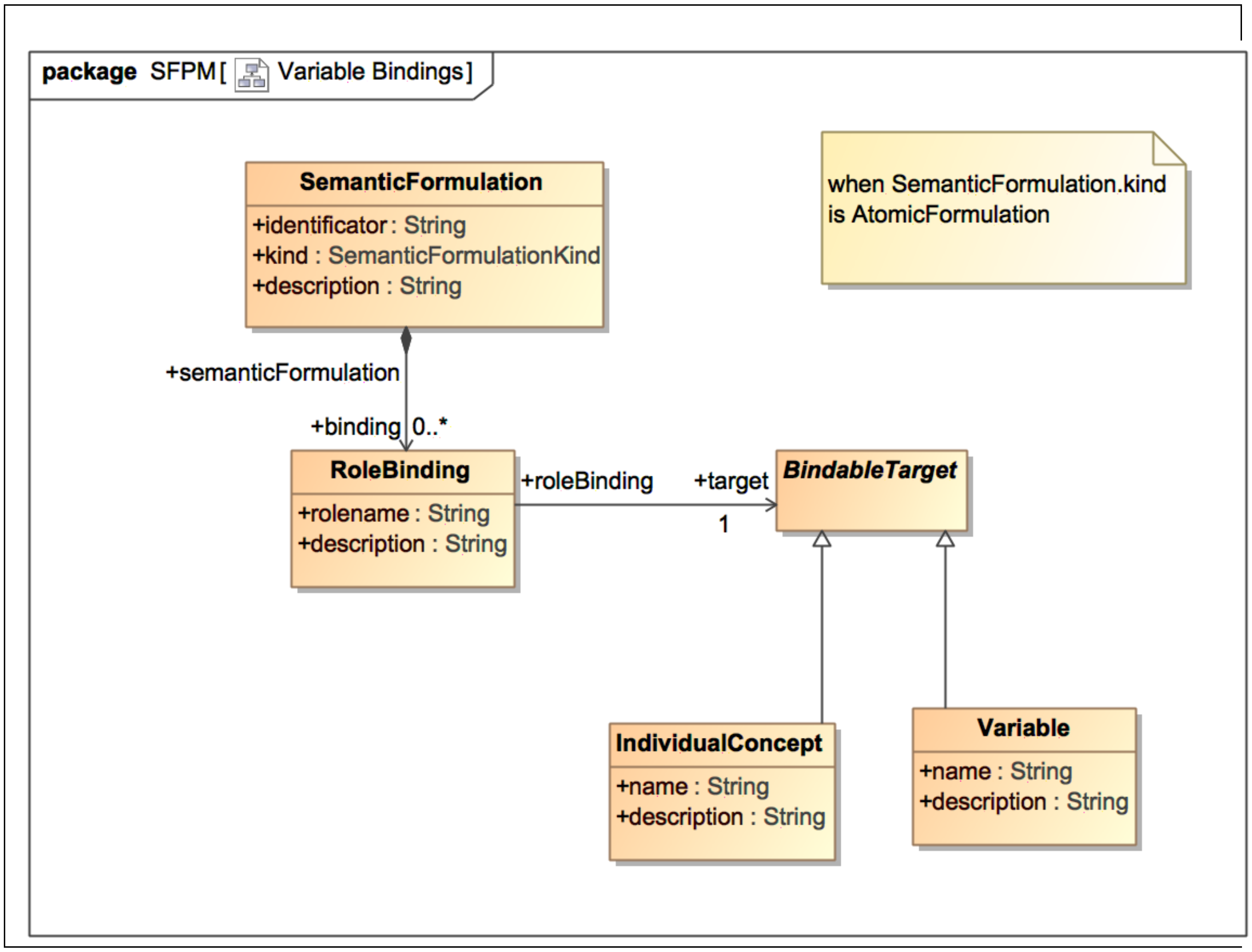

Figure 16. UML class diagram Variable Bindings

8.4.3.1 RoleBinding Class

RoleBinding class represents a connection of an atomic formulation to a bindable target. The rolename property of the RoleBinding refers to one of the roles in the atomic formulation.

RoleBinding also represents a connection of certain other SemanticFormulation to its elements. In this case, the rolename property identifies the element, if there is more than one. For example, NumericRangeQuantification has two elements that represent the minimum and maximum cardinality. The rolename "min" refer to the minimum cardinality, and the rolename "max" refers to the maximum cardinality. Other SemanticFormulation may have a single element, in which case the rolename is ignored.

Superclass

Attributes

rolename:String[1]	Unique reference to the role of the SemanticFormulation
description: String[0..1]	Description of the binding

Associations

target:BindableTarget[1]	BindableTarget

Example 1. SFPM XMI

```
  <operand xmi:id="sem1422" verb="vc9" identificator="" kind="AtomicFormulation"
                description="">
     <binding rolename="ActionElement" target="var525"/>
     <binding rolename="DataElement" target="var527"/>
  </operand>

      <verb xmi:id="vc9" name="ActionElement addresses DataElement"/>

      <variable xmi:id="var525" range="nc4" name="S1"/>
      <variable xmi:id="var527" range="nc2" name="BP"/>
```

Example 2. Readable SFP language

```
      [ActionElement addresses DataElement :KDM] S1,BP
```

Example 3. SFPM XMI

```
<variable xmi:id="var450" range="nc2" name="C2">
    <restriction xmi:id="sem1125" verb="vc22" identificator=""
                  kind="AtomicFormulation" description="">
         <binding rolename="DataElement" target="var450"/>
         <binding rolename="Datatype" target="ic14"/>
         <binding rolename="Name" target="ic135"/>
     </restriction>
</variable>

   <verb xmi:id="vc22" name="DataElement is a constant of Datatype with Name"/>
```

```
      <individual xmi:id="ic14" name="defaultInt"/>
      <individual xmi:id="ic135" name="NULL"/>
```

Example 4. Readable SFP language

```
Var C2: DataElement [KDM] such that
   [DataElement is a constant of Datatype with Name :KDM Patterns]
            C2, defaultInt, {"NULL" :Strings}
```

Example 5. SFPM XMI

```
  <operand xmi:id="sem1705" verb="vc4" identificator="" kind="AtomicFormulation"
             description="">
     <binding rolename="Thing1" target="ic143"/>
     <binding rolename="Thing2" target="ic168"/>
  </operand>

      <verb xmi:id="vc4" name="Thing1 is Thing2"/>

      <individual xmi:id="ic143" name="core.indicator"/>
      <individual xmi:id="ic168" name="callback_call"/>
```

Example 6. Readable SFP language

```
  [meta] core.indicator,callback_call
```

8.4.3.2 BindableTarget Class (abstract)

BindableTarget is either an IndividualConcept or a Variable.

Superclass

Example

See 8.4.3.1

8.5 Referenced Vocabularies

This section describes the representation of the referenced vocabularies of the SFP Catalog. The formalization apparatus of the SFP Catalog (defined in section 8.4) does not define the meaning of constructs involved in the definitions of the data flows and their invariants. Instead, this apparatus defines the structure of the *meaning*. The elements of meaning, identified as “atomic formulations” in section 8.4, are supplied by one or more referenced vocabularies. The SFP Catalog assumes the use of the ISO/OMG Knowledge Discovery Metamodel (KDM) vocabulary as the foundation for the formalizations, and some generic parts of the vocabulary described in the Semantics of Business Vocabularies and Rules (SBVR) specification. Other vocabularies are introduced by a given SFP Catalog to represent:

- Entire fragments of KDM constructs based entirely on the KDM vocabulary
- Vocabulary of tags for SFP Properties
- Interfaces to the supporting capabilities of the SFP Catalog

The elements of the referenced vocabularies can be "basic" or "structured". Both types of elements are meant to be used as part of structured semantic statements by the content of the SFP Catalog. Basic elements are informally described in the vocabulary. In contrast, the "Structured" elements are formally defined using the formalization apparatus of section 8.4.

8.5.1 Vocabularies Class Diagram

This section describes the organization of referenced vocabularies of the SFP Catalog.

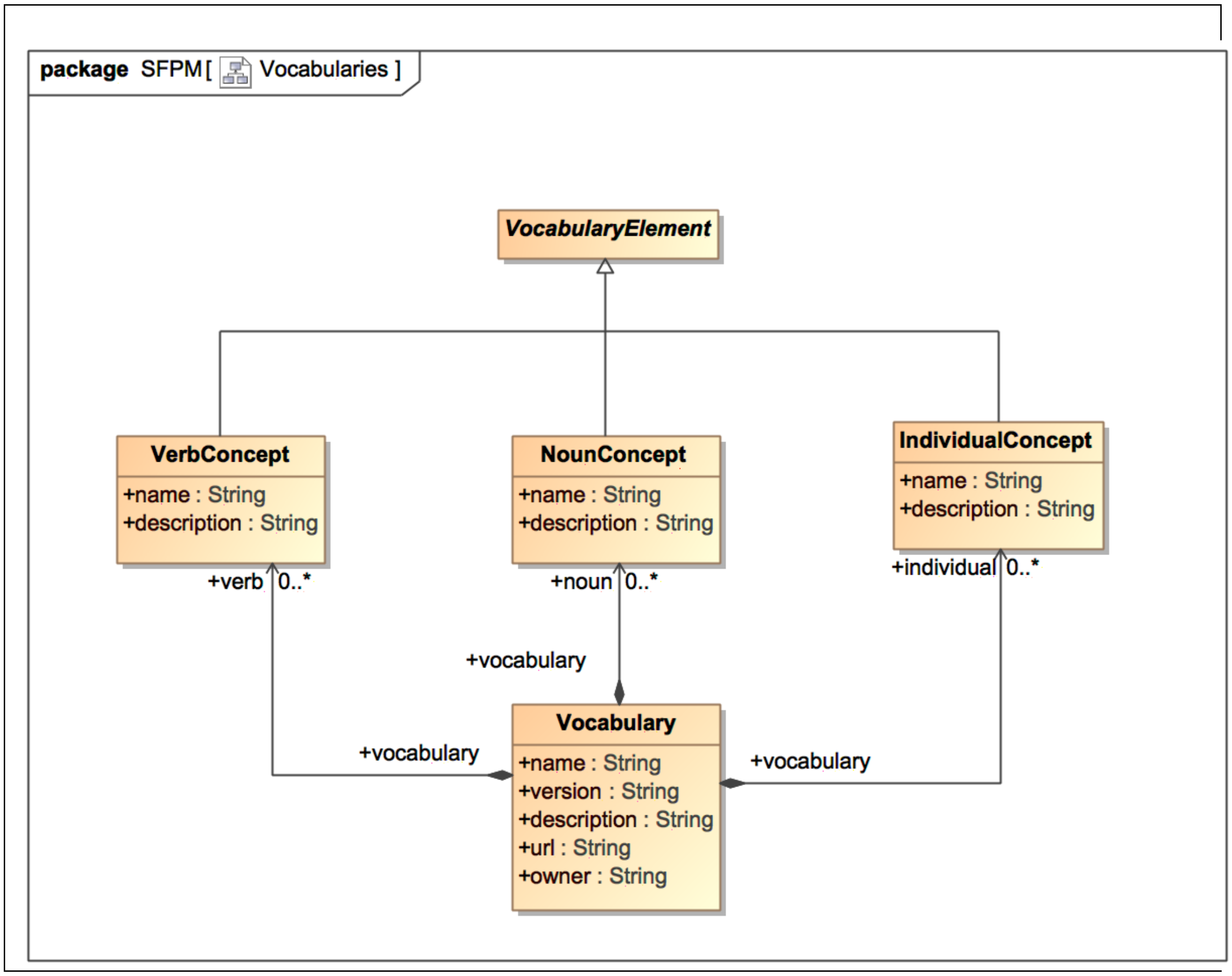

Figure 17. UML class diagram Vocabularies

8.5.1.1 NounConcept Class

NounConcept class represents a noun concept - concept that is the meaning of a noun or noun phrase. A noun concept describes a "class" of some objects. Concept is a unit of knowledge created by a unique combination of characteristics. Characteristic is abstraction of a property of an object [thing] or of a set of objects [ISO 1087-1, SBVR]. Noun concepts are used as restrictions on the ranges of values for variables and roles of verb concepts. Noun concepts can be also considered in some logical formulations (see section 8.4.2.1).

Superclass

VocabularyElement

Attributes

name:String[1]	Name of the noun concept
description: String[1]	Informal description of the noun concept

Example 1. SFPM XMI

```
<vocabulary name="KDM">
  <noun xmi:id="nc2" name="DataElement"/>
  <noun xmi:id="nc22" name="MethodUnit"/>
  <noun xmi:id="nc11" name="ValueList"/>
  <noun xmi:id="nc24" name="Name"/>
  <noun xmi:id="nc23" name="IndexUnit"/>
  <noun xmi:id="nc28" name="BooleanType"/>
  <noun xmi:id="nc21" name="Signature"/>
  <noun xmi:id="nc8" name="CharType"/>
…
</vocabulary>
```

8.5.1.2 VerbConcept Class

VerbConcept class represents a verb concept - concept that specializes the concept 'state of affairs' and that is the meaning of a verb phrase that involves one or more verb concept roles. Each instance of a verb concept is a state of affairs. For each instance, each role of the verb concept is one point of involvement of something in that state of affairs. A verb concept role is played by a thing in the domain of discourse - the world of interest. A verb concept is 'bound' by specifying the thing(s) that play the verb concept role. Linguistically those things can be specified by a quantified noun phrase or by an individual noun concept or by a pronoun that refers to a specific thing [SBVR].

An integral part of a verb concept is one or more *verb concept roles*. A verb concept role is a role that specifically characterizes its instances by their involvement in an actuality that is an instance of a given verb concept. A verb concept role is fundamentally understood as a point of involvement in actualities that correspond to a verb concept. Its incorporated characteristics come from the verb concept - what the verb concept requires of instances of the role [SBVR].

The SFPM takes a simplified approach to representing the roles of a verb concept as the words of the name of the concept starting with an uppercase letter. The corresponding RoleBinding element refers to the name of the role (see section 8.4.3.1). This convention makes the description of the SFP content less verbose (in comparison to a more complete representation based on SBVR).

Superclass

VerbForm, VocabularyElement

Attributes

name:String[1]	Name of the verb concept
description: String[1]	Informal description of the verb concept

Meaning

Consider KDM verb "ActionElement is ptrselect". The VerbFormWithRoles includes a single role "ActionElement". The extent of this verb is zero or more systems that have code such that when the code its represented as the set of KDM facts (referred to as a KDM representation), such set contains at least one fact as follows (showing KDM XMI fragment for PtrSelect):

```
<codeElement xmi:id="id.92" xmi:type="action:ActionElement" name="b2.9"
       kind="PtrSelect">
      <actionRelation xmi:id="id.93" xmi:type="action:Addresses"
            to="id.49" from="id.92"/>
      <actionRelation xmi:id="id.94" xmi:type="action:Reads"
            to="id.104" from="id.92"/>
      <actionRelation xmi:id="id.95" xmi:type="action:Writes"
            to="id.98" from="id.92"/>
</codeElement>
```

In this fragment, the role ActionElement matches the xmi:id "id.92", which is the xmi:id of the ActionElement. According to KDM constraints, the same xmi:id is the "from" property of the 3 "actionRelation" elements owned by the ActionElement. Other xmi:id in this example can be any. Also, the ActionElement may contain other associated KDM facts, such as the location, etc.

The meaning of the verb is formalized as follows:
"Any KDM representation K such that exists ID, and
also exist AR1, AR2, AR3, and
also exist N, R, W, A, such that
K contains at least one fragment

```
<codeElement xmi:id=ID xmi:type="action:ActionElement"
      name=N kind="PtrSelect">
      <actionRelation xmi:id=AR1 xmi:type="action:Addresses" to=A from=ID/>
      <actionRelation xmi:id=AR2 xmi:type="action:Reads" to=R from=ID/>
      <actionRelation xmi:id=AR3 xmi:type="action:Writes" to=W from=ID/>
</codeElement>
```

"

This fragment includes 4 KDM "existential facts" and 3 "owns facts". The references N, R, W and A provide the "context" of the fragment, into which it is "embedded". The "location" of the finding is determined by the parameter ID. For example, this can be the line in the KDM XMI file, or the associated KDM source location fact. The meaning of the verb is formalized as a Projection.
Usually, KDM fragments take the form of a "segment" of connected ActionElement. A KDM segment is determined by the two ID of its first and last ActionElement.
Connections between ActionElement in KDM is represented by an actionRelation "Flow". For example (showing KDM XMI):

```
<actionRelation xmi:id="id.91" xmi:type="action:Flow" to="id.92" from="id.86"/>
```

These facts may have special meaning when interleaving of the segments needs to be considered. Usually, the connections between segments is represented by the SFP Dataflow elements. Each Dataflow element is assumed to be a non-interleaving segment, ie the Flow shall match to exactly one KDM fact.

Example 1. SFPM XMI

```
<vocabulary name="KDM">
  <verb xmi:id="vc90" name="KDMEntity has Kind"/>
  <verb xmi:id="vc80" name="Class extends Class"/>
  <verb xmi:id="vc109" name="ActionElement is ptrselect"/>
  <verb xmi:id="vc87" name="Array has Size"/>
  <verb xmi:id="vc40" name="MemberUnit is static"/>
  <verb xmi:id="vc98" name="ActionElement reads DataElement1 and DataElement2"/>
      <!—body omitted -->
</vocabulary>
```

Example 2. SFPM XMI

```
<verb xmi:id="vc66" name="segment Begin End copies Data to Buffer of DataType">
  <definition>
    <meaning xmi:id="sem915" kind="SetProjection" description="Definition of verb
             segment Begin End copies Data to Buffer of DataType" >
       <variable xmi:id="var329" range="nc4" name="S1"/>
       <variable xmi:id="var330" range="nc4" name="S2"/>
       <variable xmi:id="var331" range="nc2" name="Data"/>
       <variable xmi:id="var332" range="nc2" name="BP"/>
       <variable xmi:id="var333" range="nc1" name="BPT"/>
       <operand xmi:id="sem916" identificator="" kind="ExistentialQuantification"
             description="">
         <variable xmi:id="var334" range="nc7" name="ArgCount"/>
         <variable xmi:id="var335" range="nc21" name="Sig"/>
         <variable xmi:id="var336" range="nc13" name="Api">
           <restriction xmi:id="sem917" verb="vc67" identificator=""
                  kind="AtomicFormulation" description="">
             <binding rolename="ControlElement" target="var336"/>
             <binding rolename="Signature" target="var335"/>
             <binding rolename="ArgCount" target="var334"/>
           </restriction>
         </variable>
         <operand xmi:id="sem918" identificator="" kind="Conjunction"
                   description="">
           <operand xmi:id="sem919" verb="vc68" identificator=""
                  kind="AtomicFormulation" description="">
             <binding rolename="Par1" target="var332"/>
             <binding rolename="Par2" target="var331"/>
           </operand>
           <operand xmi:id="sem920" verb="vc69" identificator=""
                  kind="AtomicFormulation" description="">
             <binding rolename="Begin" target="var329"/>
             <binding rolename="End" target="var330"/>
             <binding rolename="ControlElement" target="var336"/>
             <binding rolename="Signature" target="var335"/>
             <binding rolename="ArgCount" target="var334"/>
             <binding rolename="DataElement" target="var332"/>
             <binding rolename="DataType" target="var333"/>
           </operand>
         </operand>
       </operand>
    </meaning>
  </definition>
</verb>
```

Example 3. Readable SFP language

```
Verb segment Begin End copies Data to Buffer of DataType [KDM Patterns]
      Var S1 : ActionElement [KDM]
      Var S2 : ActionElement [KDM]
      Var Data : DataElement [KDM]
      Var BP: DataElement [KDM]
      Var BPT : DataType [KDM]
      Clause
                  Var ArgCount: IntegerValue [SBVR]
                  Var Sig: Signature [KDM]
                  Var Api: ControlElement [KDM] such that
                     [ControlElement of Signature with ArgCount copies data
                            :Platform APIs]  Api, Sig, ArgCount
               where
                  [two actual parameters Par1 Par2 :KDM] BP, Data
                  [segment Begin End calls ControlElement of Signature with
                         ArgCount with DataElement of DataType :KDM Patterns]
                                    S1, S2, Api, Sig, ArgCount, BP, BPT

End Verb
```

8.5.1.3 IndividualConcept Class

IndividualConcept class represents an individual noun concept - noun concept that corresponds to at most one thing in all possible worlds [ISO-1087-1, SBVR]. An example of an individual concept is an integer number "1".

Superclass

BindableTarget, VocabularyElement

Attributes

name:String[1]	Name of the individual concept
description: String[1]	Informal description of the individual concept

Example 1. SFPM XMI

```
<vocabulary name="SBVR">
   <noun xmi:id="nc7" name="IntegerValue"/>
   <noun xmi:id="nc26" name="String"/>
   <verb xmi:id="vc4" name="Thing1 is Thing2"/>
   <individual xmi:id="ic76" name="0xabad1dea"/>
   <individual xmi:id="ic15" name="10"/>
   <individual xmi:id="ic17" name="a"/>
   <individual xmi:id="ic137" name="false"/>
   <individual xmi:id="ic158" name="1.1"/>
   <individual xmi:id="ic80" name="null"/>
   <individual xmi:id="ic21" name="0"/>
   <individual xmi:id="ic68" name="1"/>
   <individual xmi:id="ic131" name="2"/>
   <individual xmi:id="ic64" name="3"/>
   <individual xmi:id="ic96" name="5"/>
</vocabulary>
```

8.5.1.4 Vocabulary Class

Vocabulary class represents a single reference vocabulary, including its version and authority. A vocabulary is a container for a collection of noun and verb concepts. The alignment with the ISO/OMG SBVR standard facilitates the use of externally defined ontologies, vocabularies, and models for the purposes of defining the content of the SFP Catalog.

Superclass

Attributes

name:String[1]	Name of the referenced vocabulary
version: String[1]	Version of the vocabulary
description:String[1]	Description of the vocabulary
url:String[1]	url to the official location of the vocabulary
owner:String[1]	Owner of the vocabulary

Example

See 8.5.1.1-3

8.5.1.5 VocabularyElement Class (abstract)

VocabularyElement class is a common parent for the elements owned by a vocabulary. This includes noun concepts, verb concepts and individual concepts.

Superclass

SemanticElement

9 Appendix A (Informative)

This section defines a simple textual language that is can be used to represent SFP context in a readable form. This language is referred to as "readable SFP language" throughout this specification. The formal grammar of the readable SFP language is given in this specification using a simple Extended Backus-Naur Form (EBNF) notation, used in the W3C specification of XML [xml].

The SFPM XMI representation can be automatically generated from this language. Examples in section 8 are given both in SFPM XMI and in this readable SFP language.

```
SFPCatalog ::= `Catalog` Version CatalogClause*
            PrimaryCluster+ CommonSection+
            `End` `Catalog`
CatalogClause ::=
      'description' '=' Text |
      'owner' '=' Text
PrimaryCluster ::= 'Cluster' Name
            SecondaryCluster+ (CWESection | ClusterSection )+
            `End' 'Cluster'
SecondaryCluster ::= 'Secondary' Name
            SFP+
            ( CWESection | ClusterSection)+
            'End' 'Secondary'

CWESection ::= 'CWEs' CWE+ 'End' 'CWEs'
CWE ::= 'CWE' CWEID Name CWEClause+ Note* 'End' 'CWE'
CWEClause ::= 'description' '=' Text |
      'details' '=' Text |
      'status' '=' Text |
      'url' '=' Text |
      'discernible' '=' DiscernibilityLevel |
      'Mapping:' VariantId+
DiscernibilityLevel ::= 'Very High' | 'High' | "Medium' | 'Low' | 'Very Low'
Note ::= 'Note:' Text

SFP ::= 'SFP' SFPID Name Description
```

```
      RootCauses Injuries
      (SFPSection | ClusterSection | CWESection)+
      ‘End’ ‘SFP’
RootCauses ::= ‘Rootcauses’ Name+ ‘End’ ‘RootCauses’
Injuries ::= ‘Injuries’ Name+ ‘End’ ‘Injuries’

ClusterSection ::= ‘Characteristics’ ReferencedContextElement*
                  ‘End’ ‘Characteristics’
ReferencedContextElement ::= ‘Ref’ ContextElementKind Name
ContextElementKind ::= ‘Resource’ | ‘Operation’ | ‘DataType’ | ‘DataElement’ |
‘API’ | ‘Decision’

SFPSection ::= ParameterSection | VariationSection | ElementSection |
CanonicalSection

ParameterSection ::= ‘Parameters’ Parameter+ ‘End’ ‘Parameters’
Parameter ::= ‘Parameter’ Name Variant+ ‘End’ ‘Parameter’
Variant ::= ‘Variant’ VariantId Name ‘->’ ‘Property’ Name InjuryMapping
InjuryMapping ::= ‘Injuries:’ Name*

VariationSection ::= ‘Variations’ Variation+ ‘End’ ‘Variations’
Variation ::= VariantRef Variation* |
      Name NL Variation+ LF
VariantRef ::= Name ‘->’ VariantId LF

ElementSection ::= ‘Elements’ DataflowElement+ ‘End’ ‘Elements’
DataflowElement ::= PrimaryDataStatement | SourceStatement | SinkStatement |
            Condition
PrimaryDataStatement ::= ‘PrimaryDataStatement’ Definition
                  ‘End’ ‘PrimaryDataStatement’
SourceStatement ::= ‘SourceStatement’ Definition
                  ‘End’ ‘’SourceStatement’
SinkStatement ::= ‘SinkStatement’ Definition
                  ‘End’ SinkStatement’
Condition ::= ‘PrimaryDataStatement’ Definition
```

```
                  'End' 'Condition'

CanonicalSection ::= 'Canonicals' CanonicalElement* 'End' 'Canonicals'
CanonicalElement ::= CanonicalForm | PrimaryDataSegment | SourceSegment |
      SinkSegment | MitigatedSourceSegment | MitigatedSinkSegment
PrimaryDataSegment ::= 'PrimaryDataSegment' Definition
                  'End' 'PrimaryDataSegment'
SourceSegment ::= 'SourceSegment' Definition
                  'End' ''SourceSegment'
SinkSegment ::= 'SinkSegment' Definition
                  'End' SinkSegment'
MitigatedSourceSegment ::= 'MitigatedSourceSegment' Definition
                  'End' 'MitigatedSourceSegment'
MitigatedSinkSegment ::= 'MitigatedSinkSegment' Definition
                  'End' 'MitigatedSinkSegment'

CommonSection ::= RootCauseSection | InjurySection | PropertySection |
      IndicatorSection | ContextSection | VocabularySection

RootCauseSection ::= 'RootCauses' Name+ 'End' 'RootCauses'
InjurySection ::= 'Injuries' Name+ 'End' 'Injuries'

PropertySection ::= 'Properties' Property+ 'End' 'Properties'
Property ::= 'Property' Name Definition 'End' 'Property'

IndicatorSection ::= 'Indicators' Indicator+ 'End' 'Indicators'
Indicator ::= 'Indicator' Name Definition 'End' 'Indicator'

ContextSection ::= 'SharedContextElements'
      ContextElement*
      'End' 'SharedContextElements'
ContextElement ::= ContextElementKind Name Definition

VocabularySection ::= 'Vocabularies'
```

```
      (Vocabulary | Definitions)*
      ‘End’ ‘Vocabularies’
Vocabulary ::= ‘Vocabulary’ VocabularyName VocabularyClause* ‘End’ ‘Vocabulary’
VocabularyClause ::= ‘description’ ‘=’ Text |
      ‘version’ ‘=’ Text |
       ‘url’ ‘=’ Text |
      ‘owner’ ‘=’ Text
Definitions ::= ‘Definitions’ VocabularyName VocabularyElement* ‘
            End’ ‘Definitions’
VocabularyElement ::= NounConcept | VerbConcept | IndividualConcept
NounConcept ::= ‘Noun’ Name Definition ‘End’ ‘Noun’
VerbConcept ::= ‘Verb’ VerbFormWithRoles Definition ‘End’ ‘Verb’
IndividualConcept ::= ‘Individual’ Name Definition ‘End’ ‘Individual’

Definition ::= Verbalization Meaning
Verbalization ::= Text
Meaning ::= Projection
Projection ::= Variable* LogicalFormulation
Variable ::= ‘Var’ Name ‘:’ NounRef ( ‘such’ ‘that’ LogicalFormulation )?
NounRef ::= Name ‘[‘ VocabularyName ’]’
LogicalFormulation ::= AtomicFormulation |
      Instantiation | LogicalOperation |
      Quantification | Objectification | AggregationFormulation |
      VerbConceptNominalization | PropositionNominalization
AtomicFormulation ::= VerbRef BindableTarget*
VerbRef ::= ‘[‘ VerbFormWithRoles ‘:’ VocabularyName ‘]’ |
      ‘[‘ContextElementKind Name ‘]’
BindableTarget ::= VarRef | IndividualRef
VarRef ::= Name
IndividualRef ::= ‘{‘ [#”] Name [#”] ‘:’ VocabularyName ‘}’
VocabularyName ::= Name

Clause ::= Identificator LogicalFormulation
Identificator ::= Name
```

```
LogicalOperation ::= LogicalNegation | LogicalBinaryOperation

LogicalNegation ::= ‘not’ LogicalFormulation

LogicalBinaryOperation ::= Disjunction | Conjunction

Disjunction ::=’Disjunction’ Clause+ ‘End’ ‘Disjunction’

Conjunction ::= Clause+

Quantification ::= UniversalQuantification | ExistentialQuantification |
      BoundedQuantification

UniversalQuantification ::= ‘for’ ‘all’ Variable+ ‘where’ LogicalFormulation

ExistentialQuantificaiton ::= Variable+ ‘where’ LogicalFormulation


BoundedQuantification ::= Bound Variable ‘where’ LogicalFormulation

Bound ::= AtLeastNBound | AtMostNBound | ExactlyNBound | ExactlyOneBound |
      NumericRangeBound

AtLeastNBound ::= ‘at’ ‘least’ Number

AtMostNBound ::= ‘at’ ‘most’ Number

ExactlyNBound ::= ‘exactly’ Number

ExactlyOneBound ::= ‘exactly’ ‘one’

NumericRangeBound ::= ‘exists’ ‘range’ MinNumber MaxNumber

MinNumber ::= Number

MaxNumber ::= Number


Instantiation ::= `instance of’ NounRef BindableTarget

Objectification ::= BindableTarget ‘objectifying’ LogicalFormulation

AggregationFormulation ::= BindableTarget ‘representing’ ‘set’ ‘of’
      Projection

VerbConceptNominalization ::= BindableTarget ‘representing’ Projection

PropositionNominalization ::= BindableTarget ‘representing’ LogicalFormulation


NCName   ::= [http://www.w3.org/TR/xml-names/#NT-NCName]

Name ::= NCNAME | (NCNAME ( #x20 )* NCNAME )*

Text ::= NCNAME | (NCNAME S NCNAME )*

Number ::= [0-9]*

VerbFormWithRoles ::= Name

VariantId ::= [0-9.]+
CWEID ::= [0-9]+ [a-z]*

SFPID ::= [0-9]+

URL      ::= [^#x5D:/?#]+ '://' [^#x5D#]+ ('#' NCName)?

Whitespace

         ::= S | Comment

S        ::= #x9 | #xA | #xD | #x20

Comment  ::= ('#' | '#' ) ( [^#xA #xD])*  [#xA #xD]
```