A Representational Analysis of the API4KP Metamodel

Adrian Paschke[1] Tara Athan[2] Davide Sottara[3] Elisa Kendall[4] Roy Bell[5]

[1] AG Corporate Semantic Web, Freie Universitaet Berlin, Germany
paschke@inf.fu-berlin.de
[2] Athan Services (athant.com), West Lafayette, Indiana, USA
taraathan@gmail.com
[3] Department of Biomedical Informatics, Arizona State University, USA
davide.sottara@asu.edu
[4] Thematix Partners LLC, New York, New York, USA
ekendall@thematix.com
[5] Raytheon, Fort Wayne, Indiana, USA
Roy_M_Bell@raytheon.com

Abstract. The API for Knowledge Platforms (API4KP) provides a common abstraction interface for discovery, exploration of metadata and querying of different types of knowledge bases. It targets the basic administration services as well as the retrieval and the modification of expressions in machine-readable knowledge representation and reasoning (KRR) languages, such as RDF(S), OWL, RuleML and Common Logic, within heterogeneous and possibly distributed (multi-language, multi-nature) knowledge platforms. This paper introduces typical use cases for API4KP and, based on their ontological requirements, analyses the representational completeness and clarity of its ontological metamodel.

1.1 Introduction

With the growing uptake of knowledge bases and semantically linked data and knowledge repositories on the Web, there is a strong demand for interacting and using them by exploring their metadata and by querying, writing and changing their data via a generic interface. Existing approaches provide solutions for particular domains and particular knowledge platforms and languages, such as federated query and mapping frameworks, specializes (query) interfaces or knowledge interchange and interoperation standards. However, what is missing is a standardized generic API for using such heterogeneous knowledge bases for generic knowledge processing applications in use cases such as the following:

1. An internet RDF publisher of earthquake observations in a publish-subscribe architecture sends out a stream of RDF graphs over time. Each graph describes an observation, including the date, location and observed strength. The vocabulary used in the streamed RDF graphs is specified in RDFS, OWL or Common Logic (CL) ontologies specific to the units of measure, time, geospatial and geoscience domains. Users of this knowledge resource

should be able to submit queries in SPARQL, receiving in response a stream of incremental query results, updated as new data becomes available.
2. Devices in the Internet of Things (IoT) publish in compact sensor-specific data formats, such as XMPP [1]. A "connected patient" system gathers input from a number of biomedical devices. A cognitive support application allows providers to query a patient's case using concepts defined in a modular medical ontology in OWL.
3. A reactive knowledge base contains event-condition-action (ECA) rules that describing actions that should be taken whenever events matching conditional patterns are detected. A smart home relies on ECA rules to adjusts the thermostat when the owner's car is within a given distance. Earlier in the day, it had placed an electronic order to the nearest supermarket.
4. A legal knowledge source provides access to a legal code in the knowledge representation language LegalRuleML. Legal knowledge engineers add material to the system over time as laws are enacted or annulled, and occasionally correct errors in rendering. Queries to the system are processed based on the appropriate version of the legal code.
5. Emergency response knowledge systems accept data and queries from mobile devices in the field where connectivity is unreliable. If a response to an information update or query request is not received within a reasonable time delay, the request is attempted again until it succeeds.

Given this variety of use cases, an abstraction is required to facilitate the interchange, deployment, revision and ultimately consumption of formal, declarative pieces of knowledge within a knowledge-driven application (KDA). OMGs standardization effort, called API4KP, aims at providing a standardized interface for the interaction with heterogenous Knowledge Platforms (KPs) at the core of a KDA. To provide a semantic foundation for the operations and their arguments, an abstract API4KP metamodel of knowledge sources has been created and expressed as an OWL ontology. [2] On the basis of typical use cases for API4KP this paper contributes with a representational analyses of the API4KP metamodel based on the Bunge-Wand-Weber (BWW) ontology, which has become a widely accepted represention modeling theory [3].

The rest of the paper is structured as follows: After providing some history and background notions in Section 1.2, we will review the primary concepts in the metamodel in Section 1.3. Section 1.4 will illustrate the use and implementation of API4KP based on the above use cases examples. On the basis of these examples, section 1.5 will perform a representational analyse the metamodel. Finally, section 1.6 will conclude this paper.

1.2 The API4KP and OntoIOp RFPs

1.2.1 The API4KP RFP + DOL

In the early to mid-2000s, a number of Object Management Group (OMG) members developed applications that explored various aspects of semantic tech-

nologies. As practitioners of Model Driven Architecture (MDA) methods [4] and information architecture (IA), they recognized limitations in the ability to modularize and reuse aspects of semantic technology in the applications they were constructing. IA in this context is focused on defining, developing, and managing information, and in particular information models, in order to separate content and context-specific concerns from other aspects of the software, process, or service architecture of a system or business. They teamed to develop an OMG Request for Proposal (RFP) [5], that asked for:

- a set of standard interfaces to knowledge bases, described in OWL and RDF Schema, that might extend to other description logic languages, RDF, and other knowledge representation languages
- access to, and modification of, knowledge base content and the ontologies (schema) that support them
- access to relevant features of the knowledge management systems themselves, including, but not limited to parsing, querying, reasoning, and other capabilities.

Work on the original RFP was postponed for a variety of reasons, including resource constraints, availability of expert architects who understood the importance of and requirements for true interoperability among knowledge-based services, and enabling infrastructure that would allow rule-based systems and description logic-based systems to be used together consistently.

The missing infrastructure, which started as an ISO specification and ultimately moved to the OMG, was provided in part through the OntoIOp (Ontology, Model And Specification Integration And Interoperability) RFP [6] and resulting DOL (The Distributed Ontology, Model, and Specification Language) language. The OntoIOp RFP asked for:

- A specification for an abstract metalanguage with an associated metamodel targeted at cross-language interoperability among a class of concrete languages used to record logical expressions found in ontologies, models and specifications.
- A list of concrete languages and translations to be recognized and correctly processed by implementations of this specification.
- A description of constraints and conformance criteria for additional concrete languages and translations between concrete languages that are not explicitly supported, but nonetheless have equivalent uses that could be recognized and correctly processed by implementations.

By early 2014, the DOL and API4KP submission teams began working together to provide a language and set of related interfaces to solve the problems identified by both groups independently. An initial submission for DOL [7] was presented to the OMG in November 2014, and is currently under revision, with a planned submission for adoption in June 2014. The API4KPs initial submission, which leverages the DOL effort, will be submitted initially in September 2015, with a planned revision for adoption in early 2016. The work described in this paper summarizes some of the findings of the API4KPs team.

1.3 API4KP Metamodel

This section will summarize the architectural and conceptual descriptions of the API4KP metamodel, as presented in more detail in [2].

1.3.1 Architectural Description

To allow for the greatest generality, we will not assume that communications are local (in the virtual address space) or synchronous - although these properties could apply in some architectures.

Communication channels may in general be many-to-many and uni- or -bidirectional, but a particular communication will have a unique sender. Each communication has a unique source; multi-source communications are not modelled directly, but are emulated by knowledge sources that are accessed as streams. We will allow for failure, either in communication or in execution, but do not specify any particular failure recovery strategy. Communicating entities may be single-sorted or many-sorted, with sorts being characterized by the kind of communications that may be initiated, forwarded or received, and by the kind of entity they may be received or forwarded from or sent to.

The API4KP architectural elements are categorized according to the following competency questions in the ontology.

1. What are the entities that are communicating in the distributed API4KP system?
2. How do they communicate, or, more specifically, what communication paradigm is used between API4KP entities?
3. What (potentially changing) roles and responsibilities do they have in the overall API4KP architecture?
4. What is the mapping of the API4KP elements into a physical distributed infrastructure?

Communicating Entities

Node: In primitive environments such as sensor networks, operating systems does not provide any abstractions, therefore (hardware) nodes communicate.

Process/Thread: In most environments, processes are supplemented by threads, so threads are more often the endpoints of communications.

Object: Computation is encapsulated by a number of interacting distributed objects representing units of decomposition for the problem domain. Objects are accessed via interfaces.

Component: Resemble objects in that they offer problem-oriented abstractions, also accessed via interfaces. They specify not only their interfaces but also the assumptions they make in terms of other components/interfaces that must be present for a component to fulfill its function.

Service and Agent: Software application which is identified via URI and can interact with other software agents (typically using a higher-level coordination and negotiation protocol).

Communication Paradigms

Strongly Coupled Communication: Low-level, direct API access requiring direct knowledge of the (downloaded) API4KP [Artifacts] or inter-process communication in distributed systems with ad-hoc network programming including message parsing-primitives.

Loosely Coupled Communication: Loosely coupled remote invocation in a two-way exchange via an [Interface] (RPC/RMI/Component/Agent) between communicating entities.

Decoupled Communication: Indirect communication, where sender and receiver are time and space uncoupled via an Intermediary.

Roles

For instance, client-server, peer-to-peer, agent architecture styles.

Client: Client knows server (after discovery) and requests a particular knowledge resource service

Server: Server is the central entity and provider of services and knowledge resources

Peer: Peer is client/requester and provider (Servant) at the same time. Knowledge resources are shared between peers and can be accessed from other peers.

Agent: an abstraction from the client-server or peer-to-peer architecture style into orchestrated or choreography style agent architectures.

Placement

Partitioning: API4KP services provided by multiple servers by partitioning a set of objects in which the service is based and distribute them between multiple-services.

Replication: Server maintain replicated API4KP server copies of them on several hosts (horizontal replication) or distributed API4KP functions into distributed peers.

Caching and Proxying: A cache stores recently used knowledge resources. Caches might be co-located with each client or located in a Proxy. Proxy provides a surrogate or placeholder for a API4KP knowledge object to control access to it.

Mobile: Mobile (executable) code that is downloaded to a client or mobile components/agents, which are running programs (both code and data/resources + state) that travel from one computer / environment to another.

1.3.2 Upper Levels of the Metamodel

The API4KP metamodel is hierarchical, with a few under-specified concepts at the upper levels, and more precisely defined concepts as subclasses.

The principle upper-level concepts in the API4KP metamodel are

Communicating Entity: architectural elements, as described above, that may communicate (initiate, forward or receive communications) with other elements.

Knowledge Source: source of machine-readable information with semantics. Examples: an RDF graph, a mutable RDF source, an RDF dataset with specified semantics, or a relational database with a mapping to an ontology.

Environment: mathematical structure of mappings and members, where the domain and codomains of the mappings are members of the environment. Example: a KRR language environment containing semantics-preserving translations from RDF and OWL into Common Logic.

Knowledge Operation: function (possibly with side-effects. i.e. effects beyond the output value returned) having a knowledge source, environment or operation type in its signature. Examples: creating, modifying, deleting, parsing, translating, querying, downloading, or reasoning about a knowledge source.

Knowledge Event: successful evaluation or execution of a knowledge operation by a particular application at a particular time.

These definitions are intentionally vague so as to be adaptable to a variety of implementation paradigms.

The fundamental building blocks of knowledge sources are *basic knowledge resources*, which are immutable knowledge sources without structure. Subclasses of basic knowledge resources are defined according to their knowledge source level.

Basic Knowledge Expression: well-formed formula in the abstract syntax of a machine-readable language.

Basic Knowledge Manifestation: character-based embodiment of a basic knowledge expression in a concrete dialect.

Basic Knowledge Item: single exemplar of a basic knowledge manifestation in a particular location.

Basic Knowledge Asset: equivalence class of basic expressions determined by the equivalence relation of an asset environment.

API4KP lifting/lowering operations provide transformations from one level to another. Knowledge sources are characterized as mutable or immutable, and immutable knowledge sources are called *knowledge resources*. In this context, immutable does not necessarily mean static, e.g., a stream of knowledge such as a feed from a sensor, may be considered an *observable* knowledge resource that is revealed over time. While an immutable knowledge source (i.e. a knowledge resource) has a specific structure, a mutable knowledge source has structure only indirectly through the structure of its state. In general, the structure of a mutable knowledge source's state changes arbitrarily over time, but could be restricted in order to emulate common dynamic patterns. Simple examples include state as a basic knowledge resource (linear history without caching), a key-value map with values that are basic knowledge resources (branching history without caching), or a sequence of basic knowledge resources (linear cached history).

In API4KP, a *structured knowledge resource* is a collection whose components are knowledge resources of the same level of abstraction, and knowledge operations will be provided to construct structured knowledge resources from other knowledge sources.

Structured Knowledge Expression: collection of knowledge expressions (either structured or basic), which are not necessarily in the same language and may themselves have structure.
Structured Knowledge Manifestation: collection of knowledge manifestations (either structured or basic), which are not necessarily in the same language or dialect and may themselves have structure.
Structured Knowledge Item: collection of knowledge items (either structured or basic), which are not necessarily in the same language, dialect, format or location, and may themselves have structure.
Structured Knowledge Asset: collection of knowledge assets (either structured or basic), which are not necessarily according to the same environment, but where there is a unique language that is the focus of the environment of each component.

To assist in defining operations on structured knowledge sources while still maintaining generality, the collection structure of a structured knowledge resource is required to arise from a monad functor. Monads of relevance to API4KP include Option,Try, Future, IO, Task, Observable, Key-Value Map, Heterogeneous List, State and M-Tree(see [2]). Collection structures that satisfy the above requirement include sets, bags and sequences, but other useful structures also meet these requirements.

Environments in API4KP are:

Categorical Environment: environment with an associative composition operation for mappings, that is closed under composition and contains an identity mapping for every member
Language Environment: environment whose members are languages
Focused Environment: nonempty environment which has a member F (called the focus or focus member) such that for every other member A, there is a mapping in the environment from A to F
Preserving Environment: environment where every mapping preserves a specified property
Asset Environment: focused, categorical, preserving language environment where the focus is a KRR language

Performatives in API4KP are modelled as *knowledge operations*. The KRR Languages covered by API4KP include ontology languages (e.g. OWL), query languages (e.g. SPARQL), languages that describe the results of queries, events and actions (e.g KR RuleML), and declarative executable languages (e.g. Prolog, Reaction RuleML). In the latter case, the languages typically includes syntactic constructs for performatives, e.g. *inform*, *query*, and the description of a knowledge resource may include a list of the performatives that are used within

it. Knowledge resources expressed in ontology languages may be considered as informative, e.g. when received from an untrusted source, or as providing an *assert*. Query resources may also be considered informative, e.g. when the sender is unauthorized, or as providing a *query* performative. Some languages (e.g. RuleML) have syntactic structures for assert, retract and query performatives, and are extensible to other actions.

In the API4KP framework, the building blocks for all knowledge operations are *actions* – unary functions, possibly with side-effects and possibly of higher-order. Actions are defined in terms of their possible knowledge events. To maintain a separation of concerns, side-effectful actions are assumed to be void, with no significant return value. Particular kinds of actions include, lifting, lowering and horizontal actions, whose output is a higher, lower or on the same knowledge source level than the input; idempotent action, where the output is equal to its composition with itself; and higher-order action, whose input or output (or both) is an action. Lifting and lowering are utility actions for changing the knowledge source level, e.g. parsing and IO. Horizontal actions are useful e.g. for constructing structured knowledge sources, while higher-order actions are needed to specify more complex operations e.g. querying. Additionally, two void actions, put and update, are defined, that have side-effects on the state of mutable knowledge resources.

1.4 API4B Metamodel Usage Examples and Implementation

1. The RDF stream of earthquake observations can be modelled using a Stream monad (also called Observable). A query registered against this RDF Stream will generate another Stream, with each item containing additions (if any) to the query result due to the assertion of the newly-arrived graph. Because RDF has monotonic semantics, the accumulated query results will always be equivalent to the result of the query applied to the accumulated graphs of the stream. Transformations that operate cumulatively, as this notion of query does, on a collection (e.g. set, list, stream, tree) are typically called "fold" or "traverse". (The "reduce" of map-reduce is a particular kind of fold.) Therefore, cumulative queries and other cumulative operations on Streams may be implemented through folding.
   If it is known that a query update will be independent of portions of the stream older than some fixed time interval, then the fold may incorporate windowing of the stream (e.g. deleting graphs older than some duration from cache) to optimize the computation without affecting entailments.
2. The connected-patient system uses a heterogeneous language environment to map the input XMPP data into a KRR language, employing terms from a vocabulary defined in a common ontology. The structure of this system may be modelled as a Set of Streams, since each device streams its output asynchronously.

3. State, Task and IO monads are appropriate to the use case of an active knowledge base where evaluation of an operation leads to side-effects; the choice of monad depends on the nature of the side-effects and the implementation. Equivalence of such knowledge resources requires not only the same entailments, but also side-effects that are in some sense equivalent. The smart home system may be modelled using a State monad, where changes to the state of the thermostat is a side-effect.
4. The legal knowledge source may be modelled as a mutable knowledge asset because of the possibility of correction of existing contents without a change of identifier. The modular nature of legal codes – chapters, sections, acts – is amenable to Set- or List-tree structures. Although some aspects, such as the addition of new rules, would fit with the Stream structure, queries are not expected to produce streaming results, and so the mutable asset model is a better fit than a Stream-based model.
5. The model of the emergency response system makes use of the Try monad, so that results can be reported as Success or Failure. A Success response has a value with the requested query results or confirmation of the update as a description. A Failure response includes information about the nature of the failure (e.g. timeout exception) so that the system can recover appropriately.

Our further use case implementations address, e.g., explicit state management and concurrency in sports competitions [8]. In a distributed, two-stage design a functional state transformer component works together with a typical stream processor composed of operators (event processing agents) using a functional message-passing style for the communication between the two. In the first stage the state transformer (ST) is responsible for explicit state management, concurrency control, reasoning (specifically, inference of state deltas), and state updates. In the second stage a stream processor (SP) is used for event pattern detection and emission of derived events, where the SP is responsible for augmenting the state deltas with detected event patterns that ST then includes in the updated state.

1.4.1 Rule-based Implementation: Prova API4KP mapping

Prova functional programming supports: single- and multi-valued functions, the latter offering direct support for non-determinism and backtracking; functional composition with the extended derive built-in; partial evaluation; lambda functions; monadic functions; monadic bind using a composition of map and join; maybe, list, state, fact, and tree monads as part of the provided library with easy extensibility; combination of monads; stream fusion capability.

Prova offers two mechanisms for composing functions: simple composition and monadic composition. In the case of simple composition, the functional pipeline is composed of functions operating on the totality of the values being passed between them. In the case of monadic composition, the functional pipeline passes around monadic data and is composed of functions operating on data in some form "contained" in the monadic data. Monadic composition in Prova is

done by pattern-matching the data passed between functions in the functional pipeline.

Monads supported are Maybe monad, List monad, State monad, Fact monad. Additionally, the stream fusion technique in Prova allows one to write functional pipelines the usual way but fuse the transformations and execute them iteratively, producing one final result at a time. A special built-in predicate *derive* is used for executing functional pipelines. The functional flavor of derive takes a single parameter, which is a Prova list with exactly three elements: functional pipeline, input, output.

1.5 Representational Analysis of the API4KP Metamodel

Following the methodology of [9] we use the Bunge-Wand-Weber representation model (BWW) [3] as a reference model for the evaluation of API4KPs metamodelling capabilities and its' related models - DOL and FRBR. For the analysis we select essential concepts, such as `Conceivable State Space`, `Lawful State Space`, `State Law`, `Stable State`, `Unstable State`, `History`, and further specializations of the top-level concepts such as event, thing, system, etc., from the BWW ontology, with respect to our representative examples and implementations from previous section 1.4. We analyze these results according to the ontological completeness and clarity of the metamodel using the following metrics: construct *deficit* (with respect to a construct in the BWW reference model), *redundancy* (multiple language constructs map to the same construct in the BWW reference model), *overload* (the same language construct maps to multiple (semantically different) constructs in the BWW reference mode), and *excess* (language constructs that do not have a mapping to the BWW reference model). Table 1.1 shows the results of the comparison with the select BWW categories.

We define the following correspondences between the API4KP metamodel and the BWW model:

- BWW Thing (a) API4KP communicating entities, (b) Mutable Knowledge Items with linear History, and (c) streamed Immutable Knowledge Items.
- BWW Property and Attributes = Properties and attributes of the individuals in the API4KP metamodel that correspond to BWW Things
- BWW Class and Kind (a) Roles and Kinds of communicating entities, (b) types of Mutable Knowledge Items and (c) types of Immutable Knowledge Items.
- BWW State (a) message queue of communicating entities (b) Knowledge Resources that are snapshots of Mutable Knowledge Sources and (c) cache of a streamed Immutable Knowledge Item
- BWW Stable State e.g. any snapshot of a non-reactive knowledge source is a stable state, according to the BWW defininition
- BWW Unstable State any intermediate state in a reactive execution chain of internal events/actions leading to a sequence of state transformations

Table 1.1. Comparison of API4KP to the Bunge-Wand-Weber Representation Model

BWW	*API4KP*	*DOL*	*FRBR*
Thing	+	-	+
Property and Attributes	+	-	+
Class and Kind	+	-	+
State	+	+	-
Stable State	+	-	-
Unstable State	+	-	-
Conceivable State Space	-	-	-
State Law	+	+	-
Lawful State Space	+	+	-
Event	+	-	-
Well-defined Event	+	-	-
Poorly-defined Event	+	-	-
External Event	+	-	-
Internal Event	+	-	-
Conceivable Event Space	-	-	-
Lawful Event Space	+	-	-
Transformation	+	+	-
Lawful Transformation	+	+	-
History	+	-	-
Acts on	+	-	-
System	+	-	-
System Composition	+	-	-
System Decomposition	+	-	-
System Environment	+	-	-
System Structure	+	-	-
Subsystem	+	-	-
Level Structure	+	-	-
Coupling	+	-	-

- BWW State Law (a) law which constrains communications, (b) law which constrains the operations that may be applied to a Mutable Knowledge Source and (c) law constraining the content of a streamed Knowledge Resource
- BWW Event (a) communication event, (b) subset of API4KP Events corresponding to an operation that has a snapshot of a Mutable Knowledge Source as input and whose output or side-effect is used to specify the next snapshot of that Mutable Knowledge Source and (c) retrieval of a stream item
- BWW Well-defined Event (b) API4KP Event as above, whose output is the next snapshot
- BWW Poorly-defined Event (b) API4KP Event as above with either i) the next snapshot obtained as a non-deterministic side-effect, or ii) the output is a function that requires additional input in order to specifiy a unique output
- BWW External Event (a) Client request (b) application of a transformation to a Mutable Knowledge Source
- BWW Internal Event (b) communication between internal communicating entities, e.g. Proxy and Knowledge Platform, (b) event/action in a reactive execution chain
- BWW Transformation (b) API4KP Operation
- BWW System and System Environment collections of internal and external API4KP communcating entities, resp. (note: system environment should not be confused with the API4KP notion of Logical Environment)

- BWW System Structure pattern of communication channels among a collection of internal API4KP communicating entities
- BWW Subsystem API4KP communicating entities that are Knowledge Platforms contain a subsystem of Mutable Knowledge Sources

In summary, API4KP shows high ontological completeness and clarity with respect to the BWW reference model, which is due to its close semantic similarity to the top level constructs in the API4KP metamodel. Not all of the BWW concepts are precisely related to API4KP concepts; this can be attributed to the purpose of BWW as a model for physical systems, which differs from API4KP's purpose as a metamodel of knowledge representation and reasoning systems addressing both physical and abstract information objects.

1.6 Conclusion and Future Work

This paper contributes a representational analysis of the API4KP metamodel [2] by an ontological comparison with the Bunge-Wand-Weber representation reference model. Future work may include a BWW-based comparison of the API4KP metamodel against other models of platform-specific KB APIs (e.g. Linked Open Data Platform, (Reaction) RuleML interfaces, Apache Metamodel, Kleisli Query System, OWL API, etc.), in order to show that the API4KP metamodel is a true conceptual abstraction and generalization of such specific API models.

References

1. : Extensible messaging and presence protocol (XMPP): Core. `http://xmpp.org/rfcs/rfc3920.html`
2. Athan, T., Bell, R., Kendall, E., Paschke, A., Sottara, D.: API4KP Metamodel: a Meta-API for Heterogeneous Knowledge Platforms. In: Proceedings of RuleML 2015. (2015)
3. Rosemann, M., Green, P.: Developing a Meta Model for the Bunge-Wand-Weber Ontological Constructs. Inf. Syst. **27**(2) (April 2002) 75–91
4. Mellor, S.J., Kendall, S., Uhl, A., Weise, D.: MDA Distilled. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (2004)
5. Object Management Group (OMG): API4KB request for proposal. `http://www.omg.org/techprocess/meetings/schedule/API4KB.html`
6. Object Management Group (OMG): OntoIOp request for proposal. `http://www.omg.org/cgi-bin/doc?ad/2013-12-02`
7. : The distributed ontology, model, and specication language (dol). `https://github.com/tillmo/DOL/blob/master/Standard/ebnf-OMG_OntoIOp_current.pdf`
8. Kozlenkov, A., Jeffery, D., Paschke, A.: State management and concurrency in event processing. In: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS 2009, Nashville, Tennessee, USA, July 6-9, 2009. (2009)
9. Rosemann, M., Indulska, M.: A Reference Methodology for Conducting Ontological Analyses. In: Proceedings of ER 2004 - 23rd International Conference on Conceptual Modeling, Springer (2004) 110–121